DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 29

1. GRAPH EXPLORATION OVERVIEW
2. BREADTH–FIRST SEARCH (BFS)
3. DEPTH–FIRST SEARCH (DFS)
4. BONUS CONTENT: STRONGLY CONNECTED COMPONENTS

1. GRAPH EXPLORATION OVERVIEW

2. BREADTH-FIRST SEARCH (BFS)

3. DEPTH-FIRST SEARCH (DFS)

4. BONUS CONTENT: STRONGLY CONNECTED COMPONENTS

GRAPH EXPLORATION

Sample popular methods:

- breadth–first search (BFS)
  - uses a queue

GRAPH EXPLORATION

Sample popular methods:

- breadth–first search (BFS)
  - uses a queue
- depth–first search (DFS)
  - uses a stack

GRAPH EXPLORATION

Sample popular methods:

- breadth–first search (BFS)
  - uses a queue
- depth–first search (DFS)
  - uses a stack
- random walks
  - example: PageRank (see Homework 10)

USEFUL GRAPH SUBROUTINES

In [2]:

```
type Vertex = usize;
type ListOfEdges = Vec<(Vertex,Vertex)>;
type AdjacencyLists = Vec<Vec<Vertex>>;

#[derive(Debug)]
struct Graph {
    n: usize, // vertex labels in {0,...,n-1}
    outedges: AdjacencyLists,
}

// reverse direction of edges on a list
fn reverse_edges(list:&ListOfEdges)
        -> ListOfEdges {
    let mut new_list = vec![];
    for (u,v) in list {
        new_list.push((*v,*u));
    }
    new_list
}

reverse_edges(&vec![(3,2),(1,1),(0,100),(100,0)])
```

Out[2]:

```
[(2, 3), (1, 1), (100, 0), (0, 100)]
```

USEFUL GRAPH SUBROUTINES

In [2]:

```
type Vertex = usize;
type ListOfEdges = Vec<(Vertex,Vertex)>;
type AdjacencyLists = Vec<Vec<Vertex>>;

#[derive(Debug)]
struct Graph {
    n: usize, // vertex labels in {0,...,n-1}
    outedges: AdjacencyLists,
}

// reverse direction of edges on a list
fn reverse_edges(list:&ListOfEdges)
        -> ListOfEdges {
    let mut new_list = vec![];
    for (u,v) in list {
        new_list.push((*v,*u));
    }
    new_list
}

reverse_edges(&vec![(3,2),(1,1),(0,100),(100,0)])
```

Out[2]:

```
[(2, 3), (1, 1), (100, 0), (0, 100)]
```

In [3]:

```
impl Graph {
    fn add_directed_edges(&mut self,
                          edges:&ListOfEdges) {
        for (u,v) in edges {
            self.outedges[*u].push(*v);
        }
    }

    fn create_directed(n:usize,edges:&ListOfEdges)
                                            -> Graph {
        let mut g = Graph{n,outedges:vec![vec![];n]};
        g.add_directed_edges(edges);
        g
    }

    fn create_undirected(n:usize,edges:&ListOfEdges)
                                            -> Graph {
        let mut g = Self::create_directed(n,edges);
        g.add_directed_edges(&reverse_edges(edges));
        g
    }
}
```

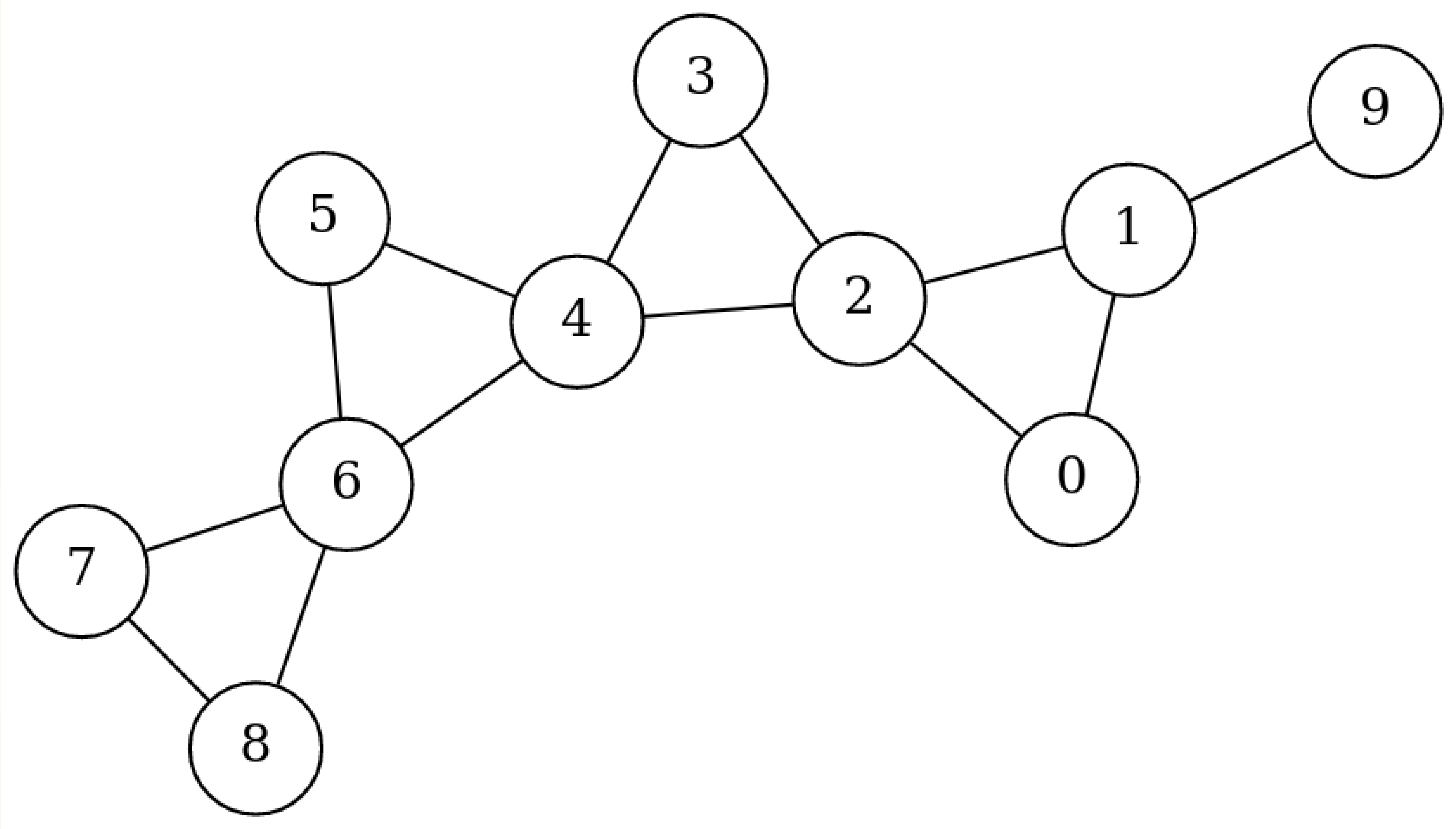
1. GRAPH EXPLORATION OVERVIEW

2. BREADTH-FIRST SEARCH (BFS)

3. DEPTH-FIRST SEARCH (DFS)

4. BONUS CONTENT: STRONGLY CONNECTED COMPONENTS

SAMPLE GRAPH



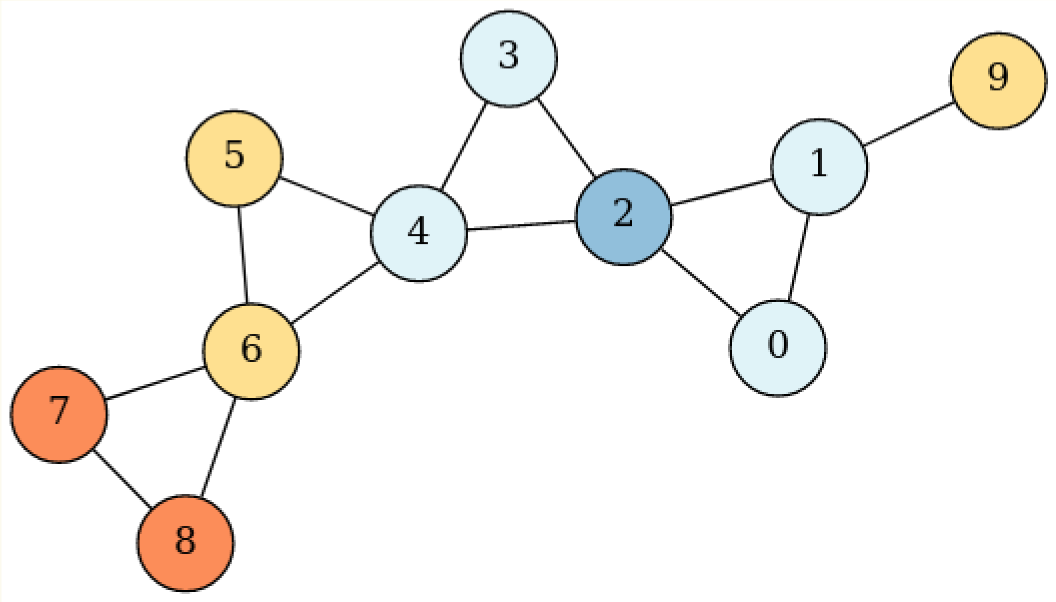

```
In [4]: let n: usize = 10;
        let edges: ListOfEdges = vec![(0,1),(0,2),(1,2),(2,4),(2,3),(4,3),(4,5),(5,6),(4,6),(6,8),(6,7),(8,7),(1,9)];
        let graph = Graph::create_undirected(n,&edges);
        graph

Out[4]: Graph { n: 10, outedges: [[1, 2], [2, 9, 0], [4, 3, 0, 1], [2, 4], [3, 5, 6, 2], [6, 4], [8, 7, 5, 4], [6, 8], [7, 6], [1]] }
```

BREADTH-FIRST SEARCH (BFS)

General idea:

- start from some vertex and explore its neighbors (distance 1)
- then explore neighbors of neighbors (distance 2)
- then explore neighbors of neighbors of neighbors (distance 3)
- ...

Our example: start from vertex 2

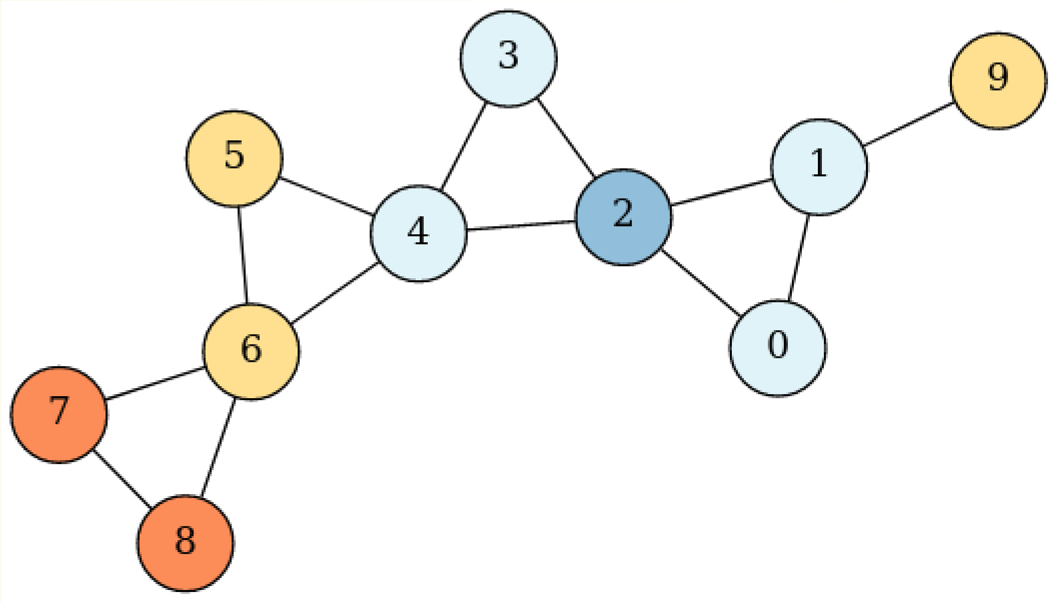


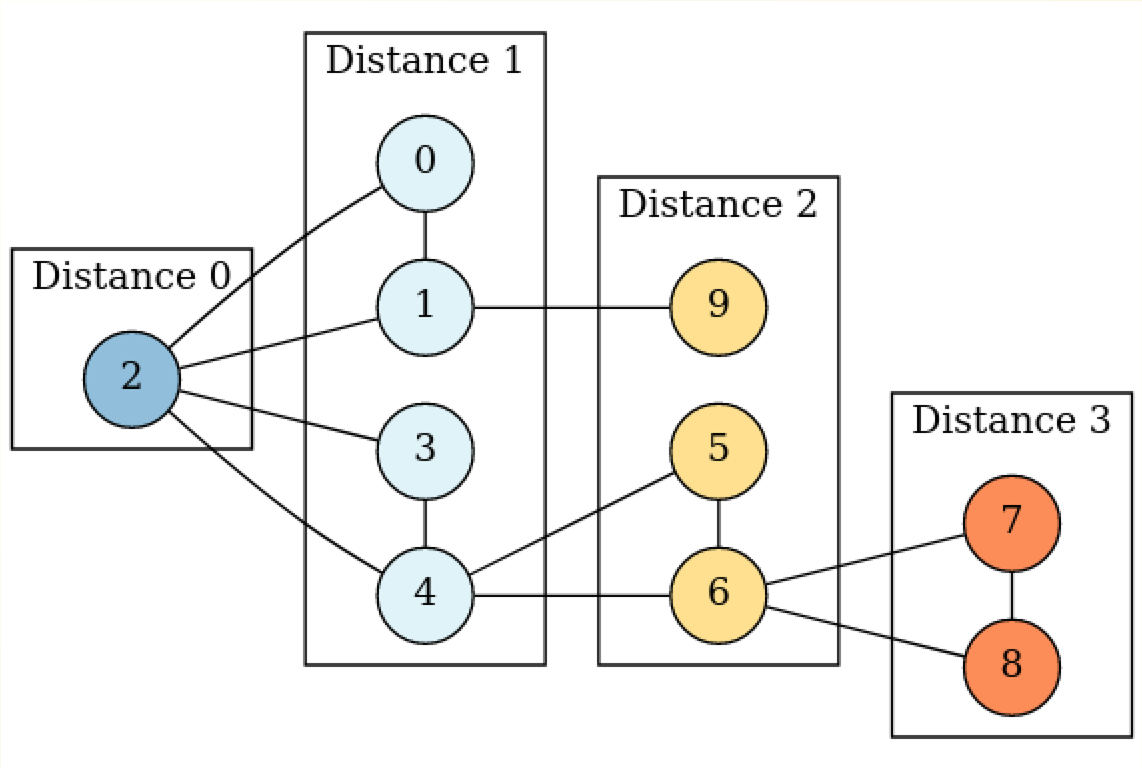
BREADTH-FIRST SEARCH (BFS)

General idea:

- start from some vertex and explore its neighbors (distance 1)
- then explore neighbors of neighbors (distance 2)
- then explore neighbors of neighbors of neighbors (distance 3)
- ...

Our example: start from vertex 2





IMPLEMENTATION: COMPUTE DISTANCES FROM VERTEX 2 VIA BFS

`distance[v]`: distance of `v` from vertex 2 (`None` is unknown)

In [5]:
```
let start: Vertex = 2; // <= we'll start from this vertex

let mut distance: Vec<Option<u32>> = vec![None;graph.n];
distance[start] = Some(0); // <= we know this distance
distance
```

Out[5]:
```
[None, None, Some(0), None, None, None, None, None, None, None]
```

IMPLEMENTATION: COMPUTE DISTANCES FROM VERTEX 2 VIA BFS

`distance[v]`: distance of `v` from vertex 2 (`None` is unknown)

In [5]:
```
let start: Vertex = 2; // <= we'll start from this vertex

let mut distance: Vec<Option<u32>> = vec![None;graph.n];
distance[start] = Some(0); // <= we know this distance
distance
```

Out[5]:
```
[None, None, Some(0), None, None, None, None, None, None, None]
```

`queue`: vertices to consider, they will arrive layer by layer

In [6]:
```
use std::collections::VecDeque;
let mut queue: VecDeque<Vertex> = VecDeque::new();
queue.push_back(start);
queue
```

Out[6]:
```
[2]
```

IMPLEMENTATION: COMPUTE DISTANCES FROM VERTEX 2 VIA BFS

Main loop:

- consider vertices one by one
- add their new neighbors to the processing queue

IMPLEMENTATION: COMPUTE DISTANCES FROM VERTEX 2 VIA BFS

Main loop:

- consider vertices one by one
- add their new neighbors to the processing queue

In [7]:
```
println!("{:?}",queue);
while let Some(v) = queue.pop_front() { // new unprocessed vertex
    println!("{:?}",queue);
    for u in graph.outedges[v].iter() {
        if let None = distance[*u] { // consider all unprocessed neighbors of v
            distance[*u] = Some(distance[v].unwrap() + 1);
            queue.push_back(*u);
            println!("{:?}",queue);
        }
    }
};
```

```
[2]
[]
[4]
[4, 3]
[4, 3, 0]
[4, 3, 0, 1]
[3, 0, 1]
[3, 0, 1, 5]
[3, 0, 1, 5, 6]
[0, 1, 5, 6]
[1, 5, 6]
```

IMPLEMENTATION: COMPUTE DISTANCES FROM VERTEX 2 VIA BFS

Main loop:

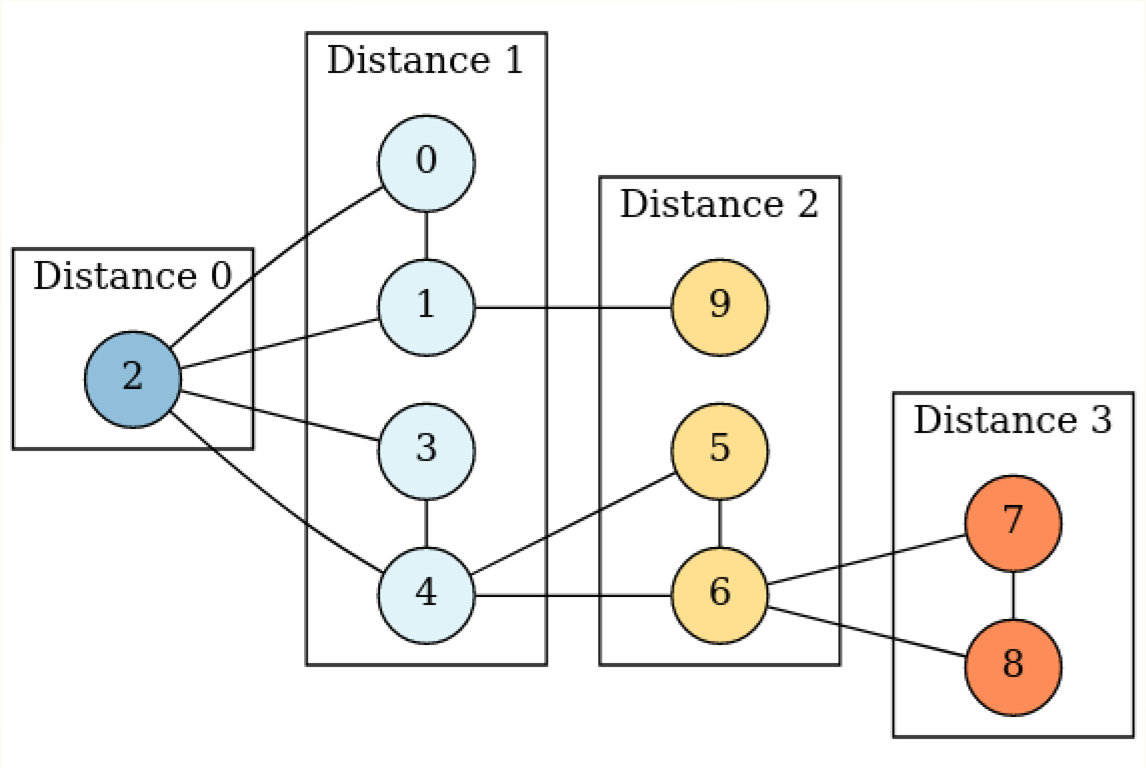
- consider vertices one by one
- add their new neighbors to the processing queue

In [7]:
```
println!("{:?}",queue);
while let Some(v) = queue.pop_front() { // new unprocessed vertex
    println!("{:?}",queue);
    for u in graph.outedges[v].iter() {
        if let None = distance[*u] { // consider all unprocessed neighbors of v
            distance[*u] = Some(distance[v].unwrap() + 1);
            queue.push_back(*u);
            println!("{:?}",queue);
        }
    }
};
```

```
[2]
[]
[4]
[4, 3]
[4, 3, 0]
[4, 3, 0, 1]
[3, 0, 1]
[3, 0, 1, 5]
[3, 0, 1, 5, 6]
[0, 1, 5, 6]
[1, 5, 6]
[5, 6]
[5, 6, 9]
[6, 9]
[9]
[9, 8]
[9, 8, 7]
[8, 7]
[7]
[]
```

IMPLEMENTATION: COMPUTE DISTANCES FROM VERTEX 2 VIA BFS

Compare results:



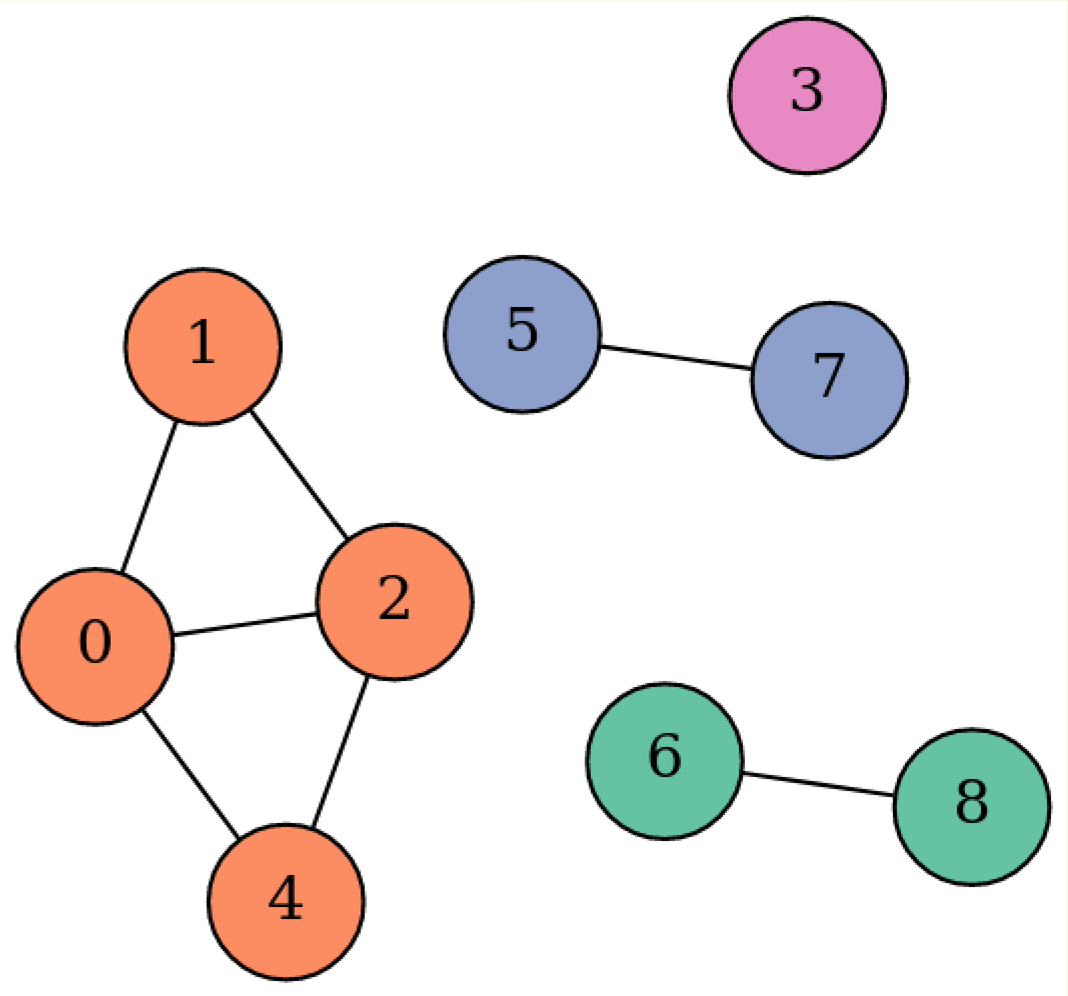
In [8]:
```
print!("vertex:distance");
for v in 0..graph.n {
    print!("   {}:{}",v,distance[v].unwrap());
}
println!();
```

```
vertex:distance   0:1   1:1   2:0   3:1   4:1   5:2   6:2   7:3   8:3   9:2
```

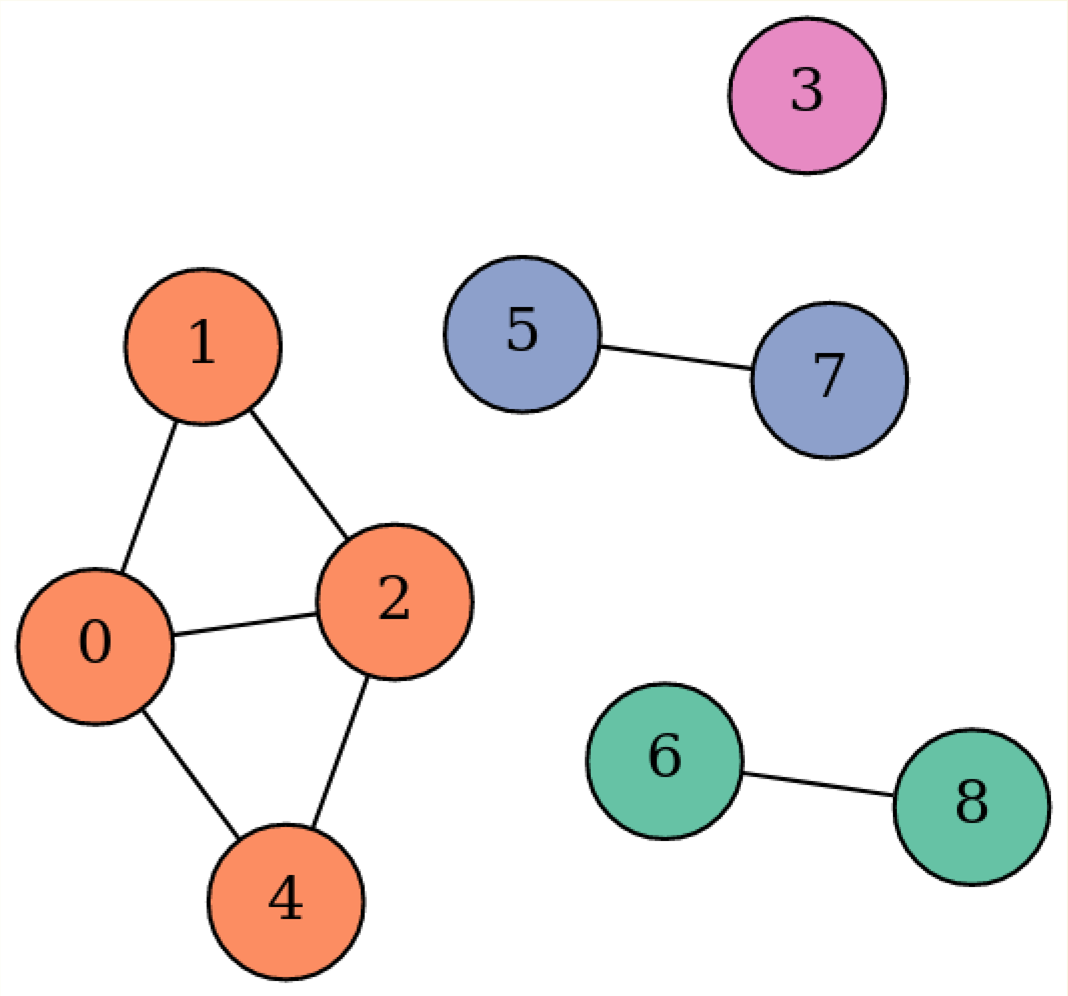
CONNECTED COMPONENTS VIA BFS

Connected component (in an undirected graph):

a maximal set of vertices that are connected



CONNECTED COMPONENTS VIA BFS



Connected component (in an undirected graph):

> a maximal set of vertices that are connected

Sample graph:

In [9]:
```
let n: usize = 9;
let edges: Vec<(Vertex,Vertex)> = vec![(0,1),(0,2),(1,2),(2,4),(0,4),(5,7),(6,8)];
let graph = Graph::create_undirected(n, &edges);
```

DISCOVERING VERTICES OF A CONNECTED COMPONENT VIA BFS

`component[v]`: `v`'s component's number (`None` ≡ not assigned yet)

In [10]:
```
type Component = usize;

fn mark_component_bfs(vertex:Vertex, graph:&Graph, component:&mut Vec<Option<Component>>, component_no:Component) {
    component[vertex] = Some(component_no);

    let mut queue = std::collections::VecDeque::new();
    queue.push_back(vertex);

    while let Some(v) = queue.pop_front() {
        for w in graph.outedges[v].iter() {
            if let None = component[*w] {
                component[*w] = Some(component_no);
                queue.push_back(*w);
            }
        }
    }
}
```

MARKING ALL CONNECTED COMPONENTS

Loop over all unassigned vertices and assign component numbers

In [11]:
```
let mut component: Vec<Option<Component>> = vec![None;n];
let mut component_count = 0;
for v in 0..n {
    if let None = component[v] {
        component_count += 1;
        mark_component_bfs(v, &graph, &mut component, component_count);
    }
};
```

MARKING ALL CONNECTED COMPONENTS

Loop over all unassigned vertices and assign component numbers

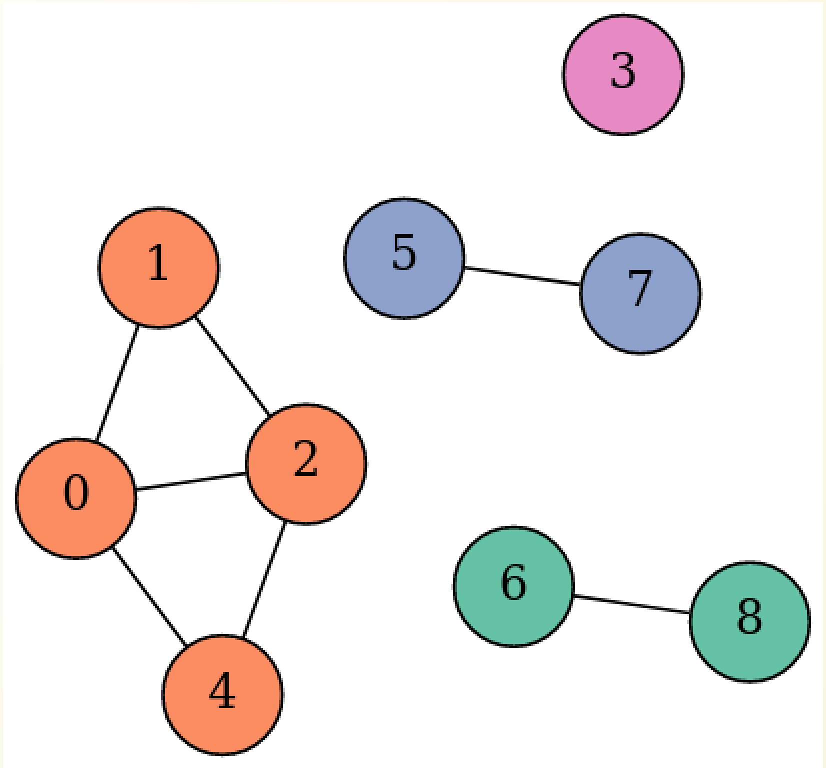
In [11]:

```
let mut component: Vec<Option<Component>> = vec![None;n];
let mut component_count = 0;
for v in 0..n {
    if let None = component[v] {
        component_count += 1;
        mark_component_bfs(v, &graph, &mut component, component_count);
    }
};
```

In [12]:

```
// Let's verify the assignment!
print!("{} components:\n[  ",component_count);
for v in 0..n {
    print!("{}:{}  ",v,component[v].unwrap());
}
println!("]\n");
```

```
4 components:
[  0:1  1:1  2:1  3:2  4:1  5:3  6:4  7:3  8:4  ]
```



1. GRAPH EXPLORATION OVERVIEW

2. BREADTH-FIRST SEARCH (BFS)

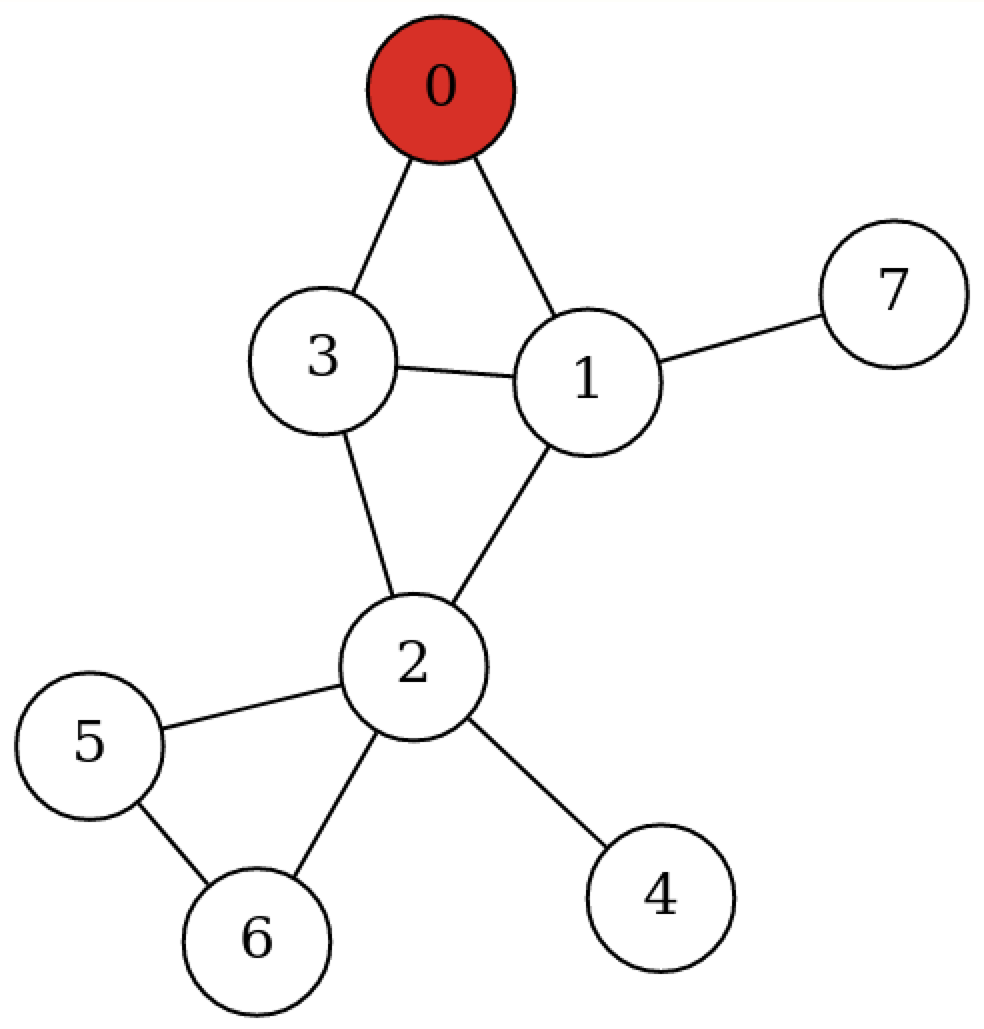
3. DEPTH-FIRST SEARCH (DFS)

4. BONUS CONTENT: STRONGLY CONNECTED COMPONENTS

DEPTH–FIRST SEARCH (DFS)

General idea:

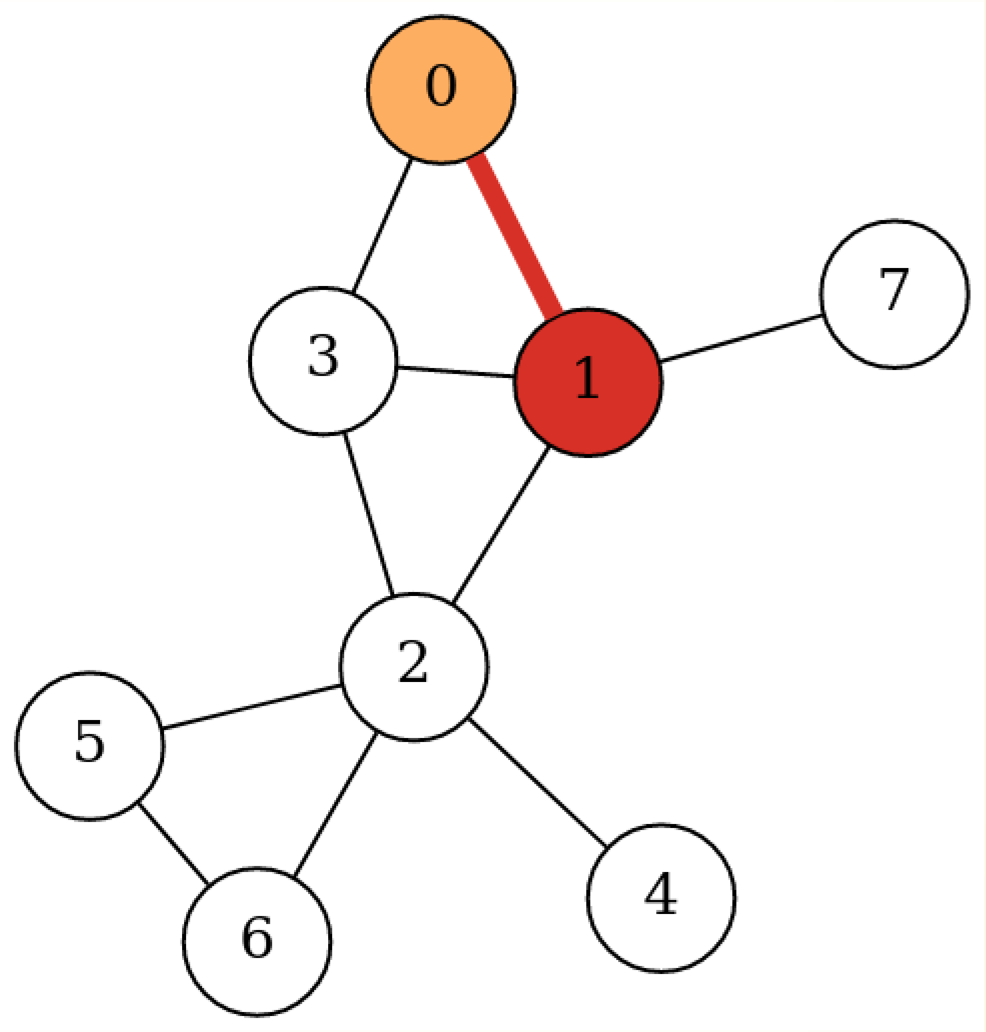
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH–FIRST SEARCH (DFS)

General idea:

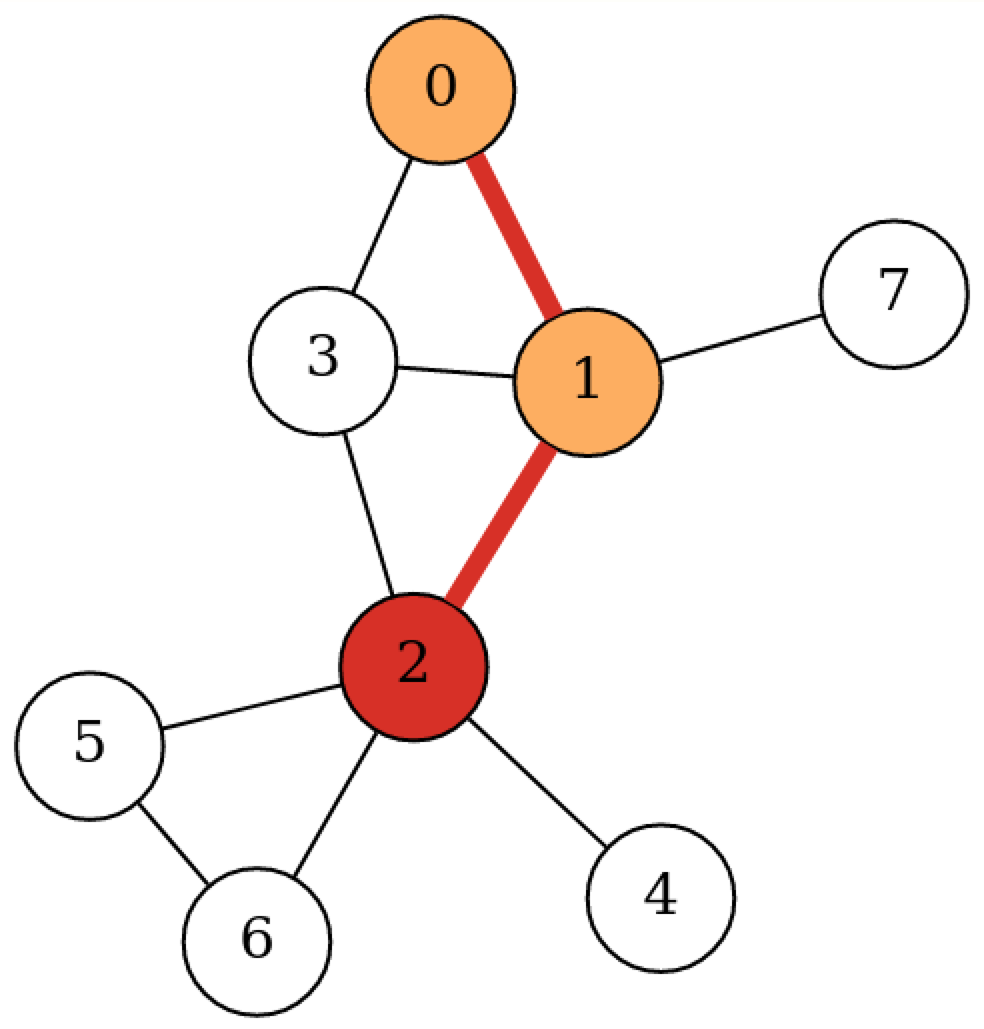
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH-FIRST SEARCH (DFS)

General idea:

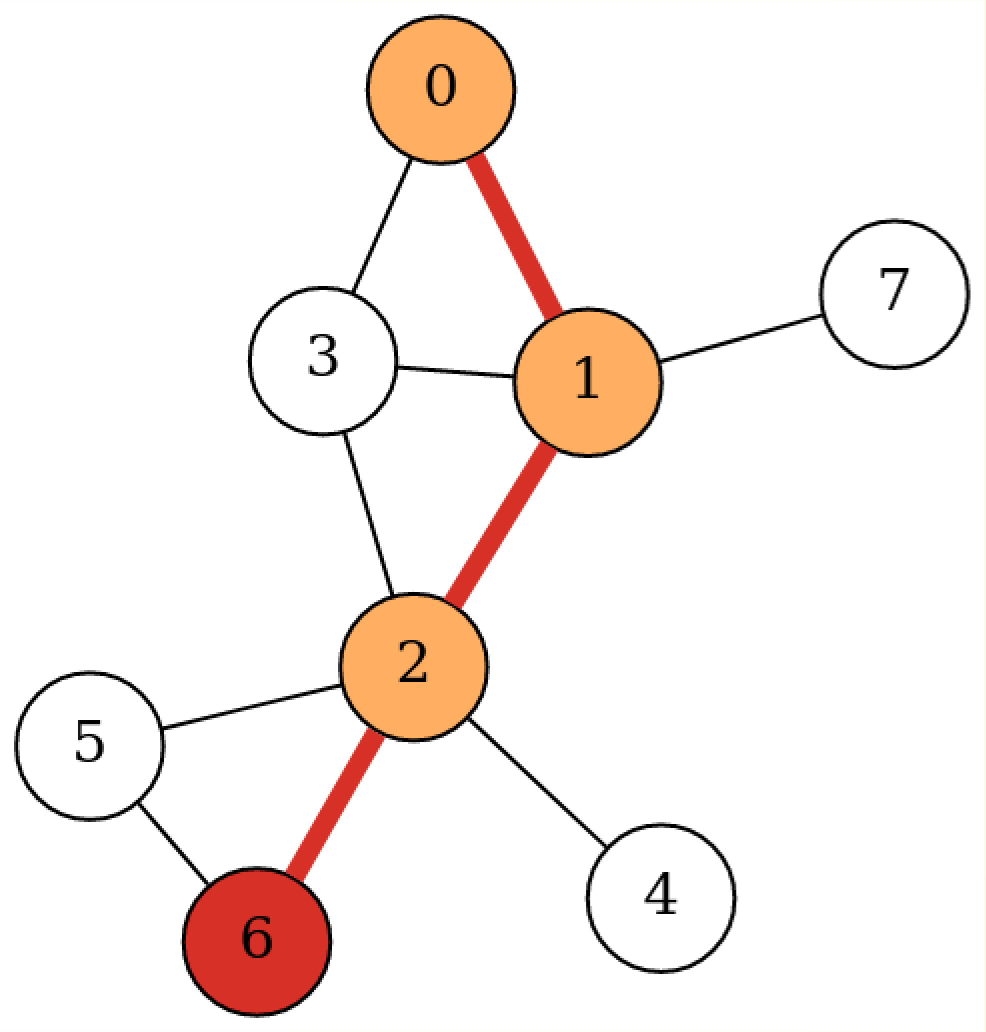
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH–FIRST SEARCH (DFS)

General idea:

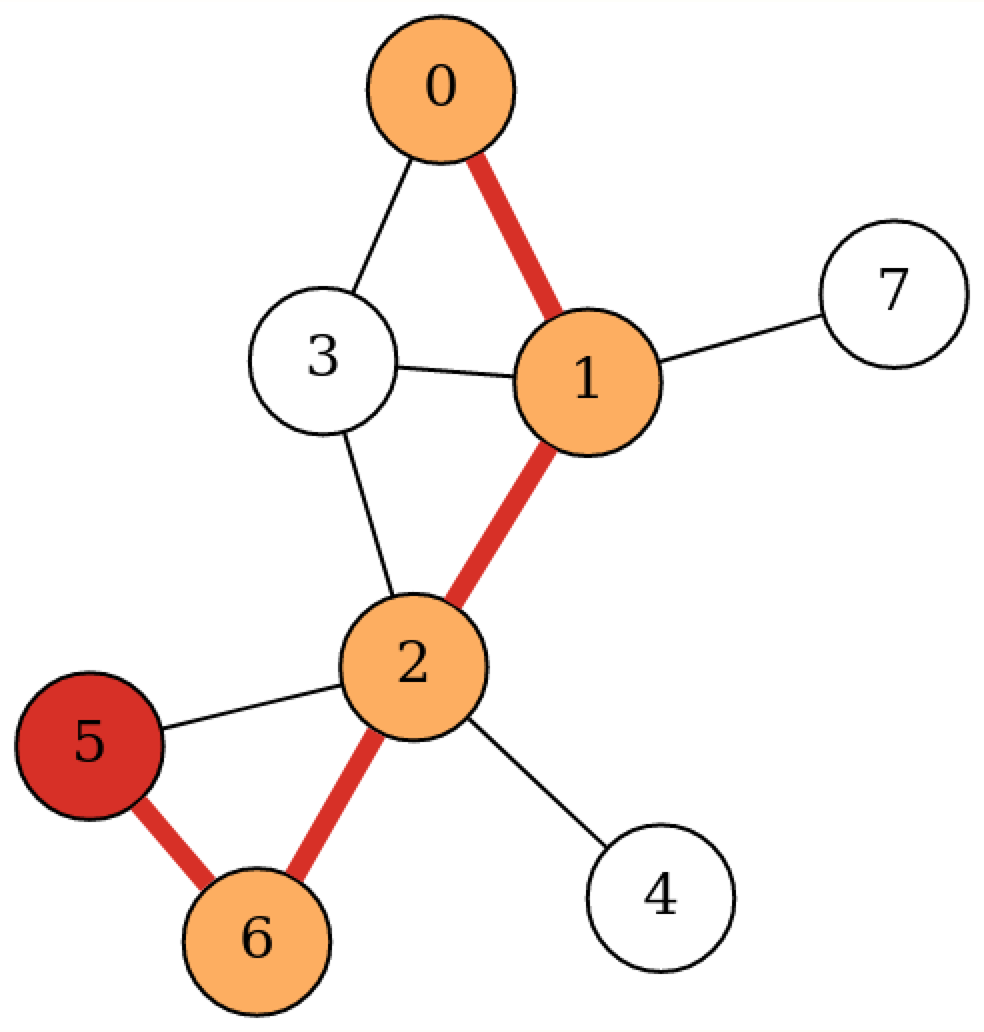
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH–FIRST SEARCH (DFS)

General idea:

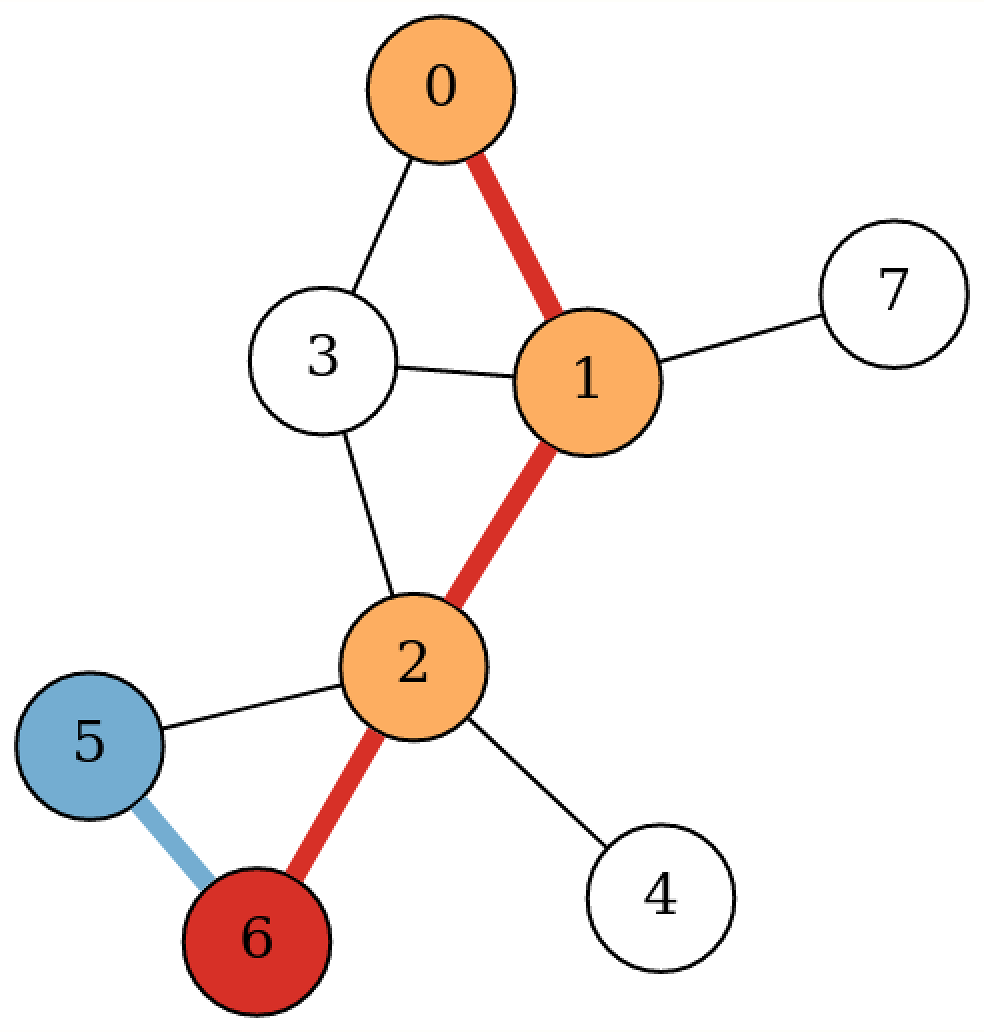
- keep moving to an unvisited neighbor
- when stuck make a step back and try again



DEPTH–FIRST SEARCH (DFS)

General idea:

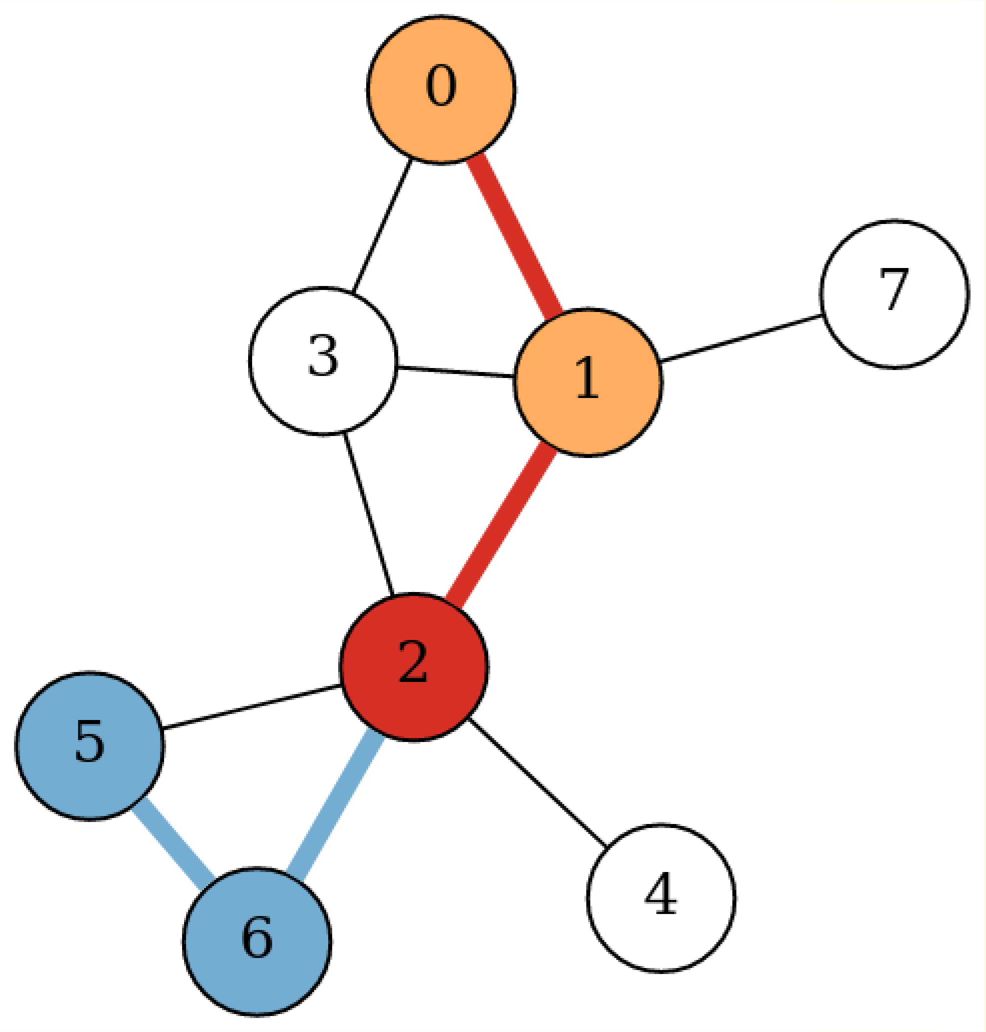
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH-FIRST SEARCH (DFS)

General idea:

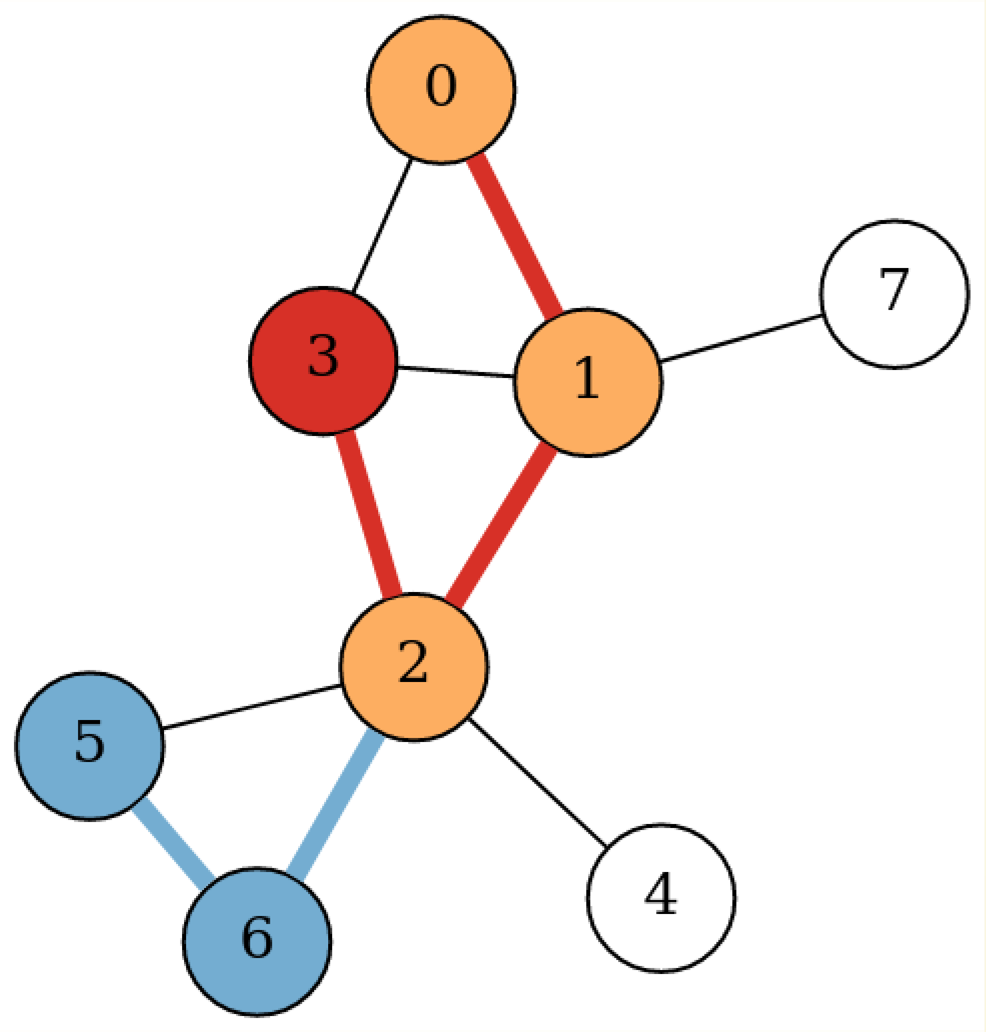
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH–FIRST SEARCH (DFS)

General idea:

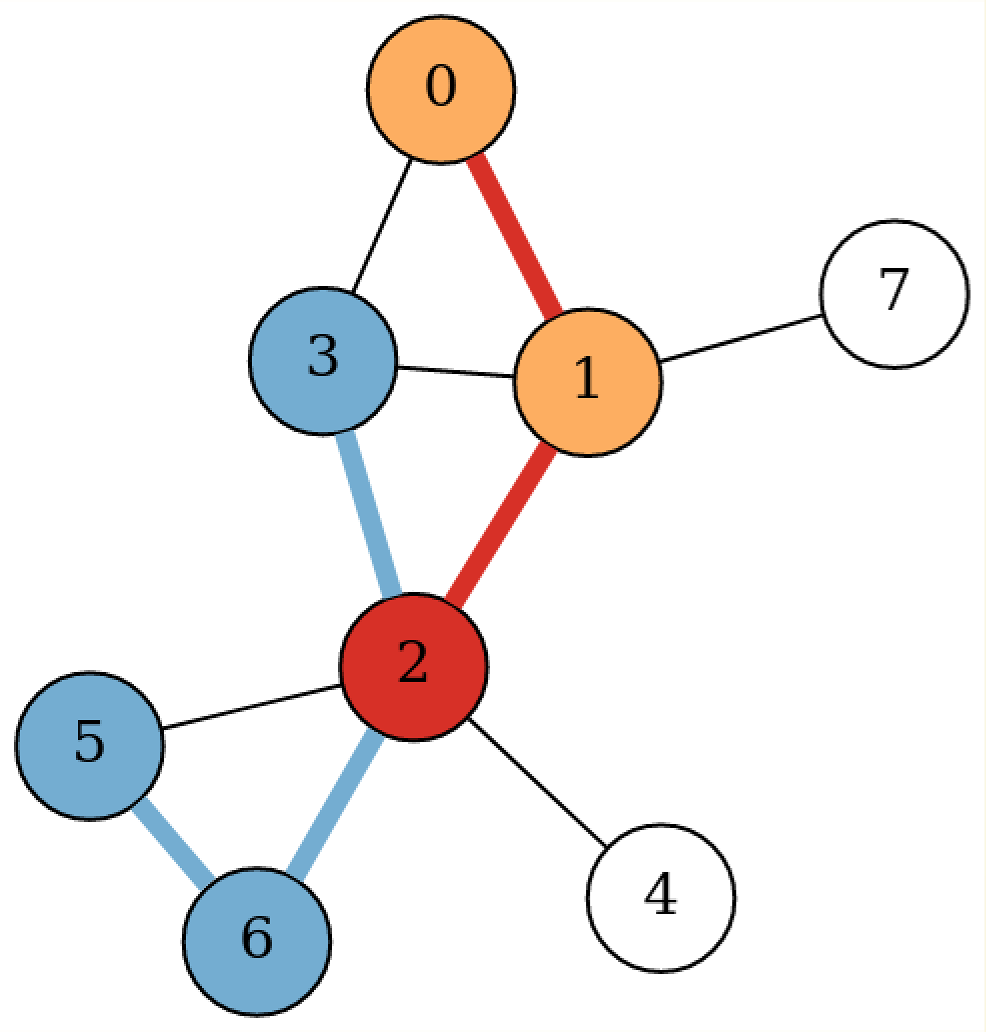
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH–FIRST SEARCH (DFS)

General idea:

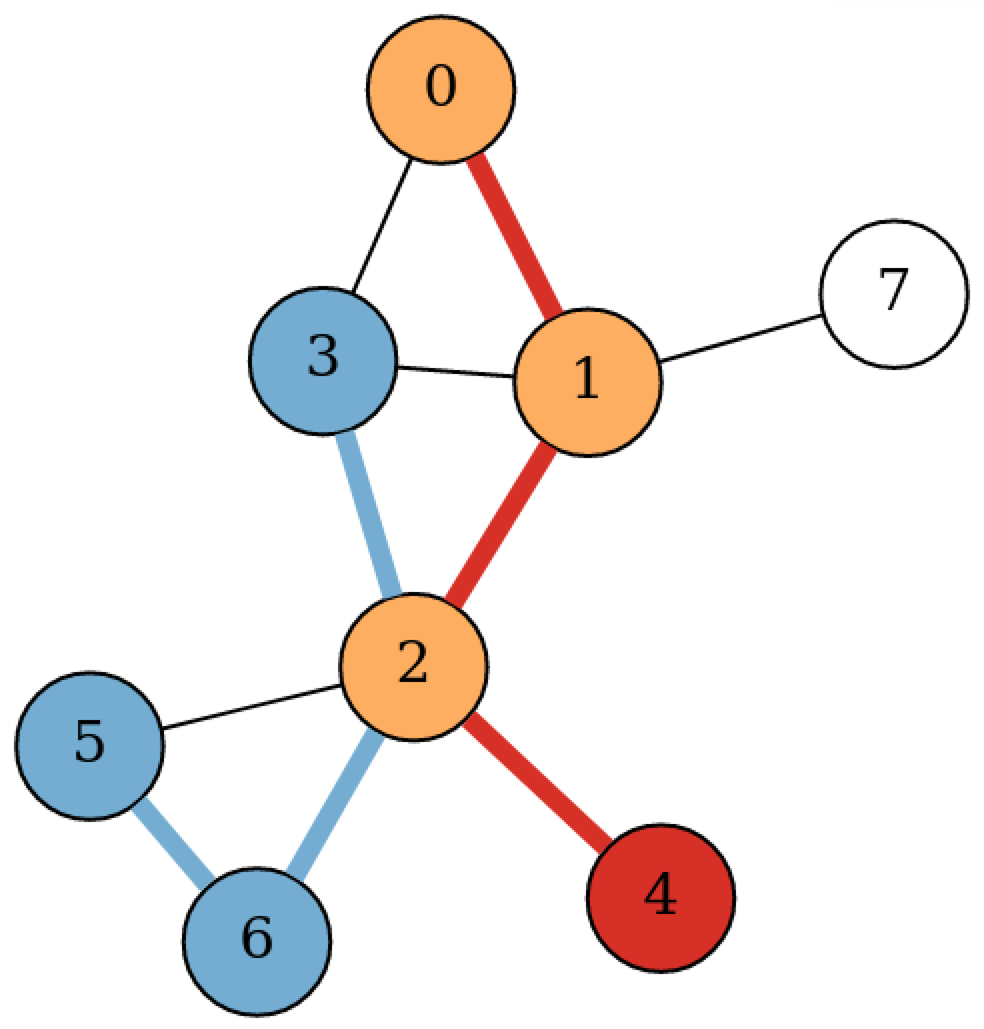
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH–FIRST SEARCH (DFS)

General idea:

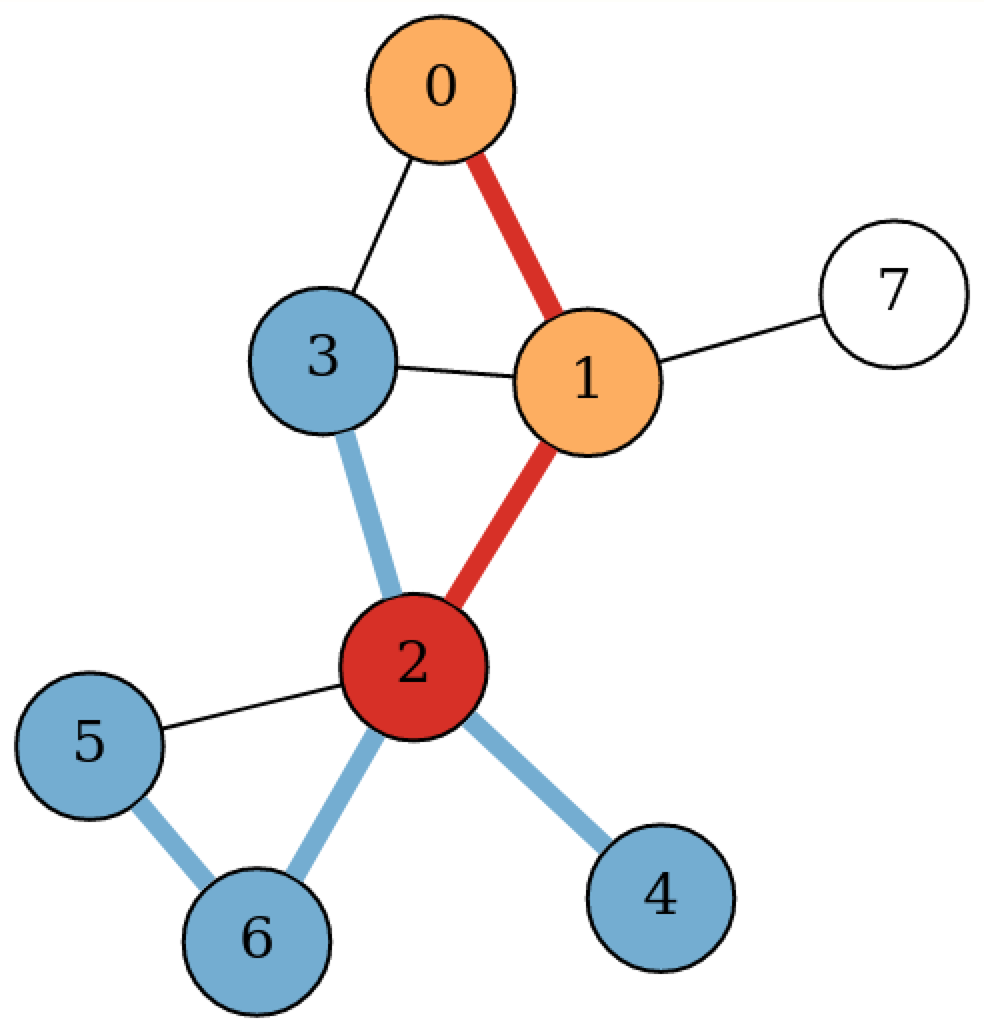
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH-FIRST SEARCH (DFS)

General idea:

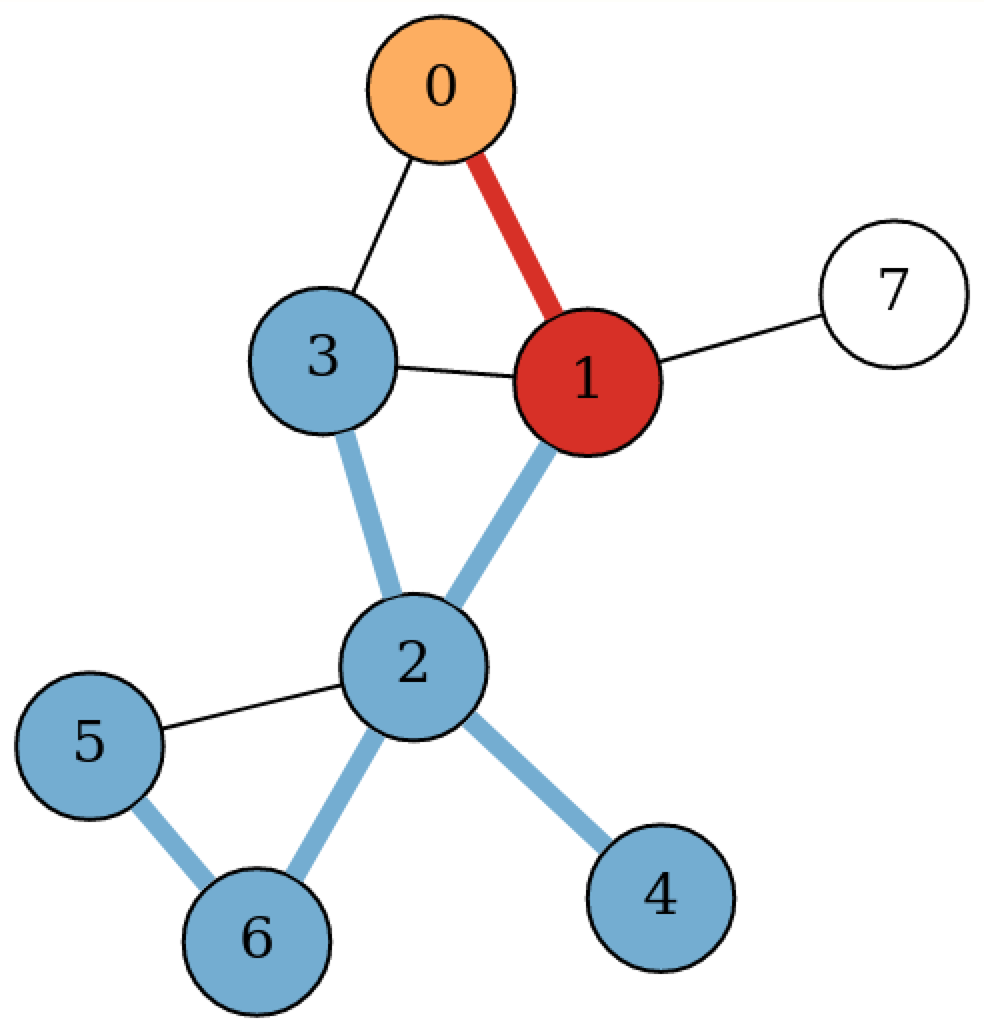
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH-FIRST SEARCH (DFS)

General idea:

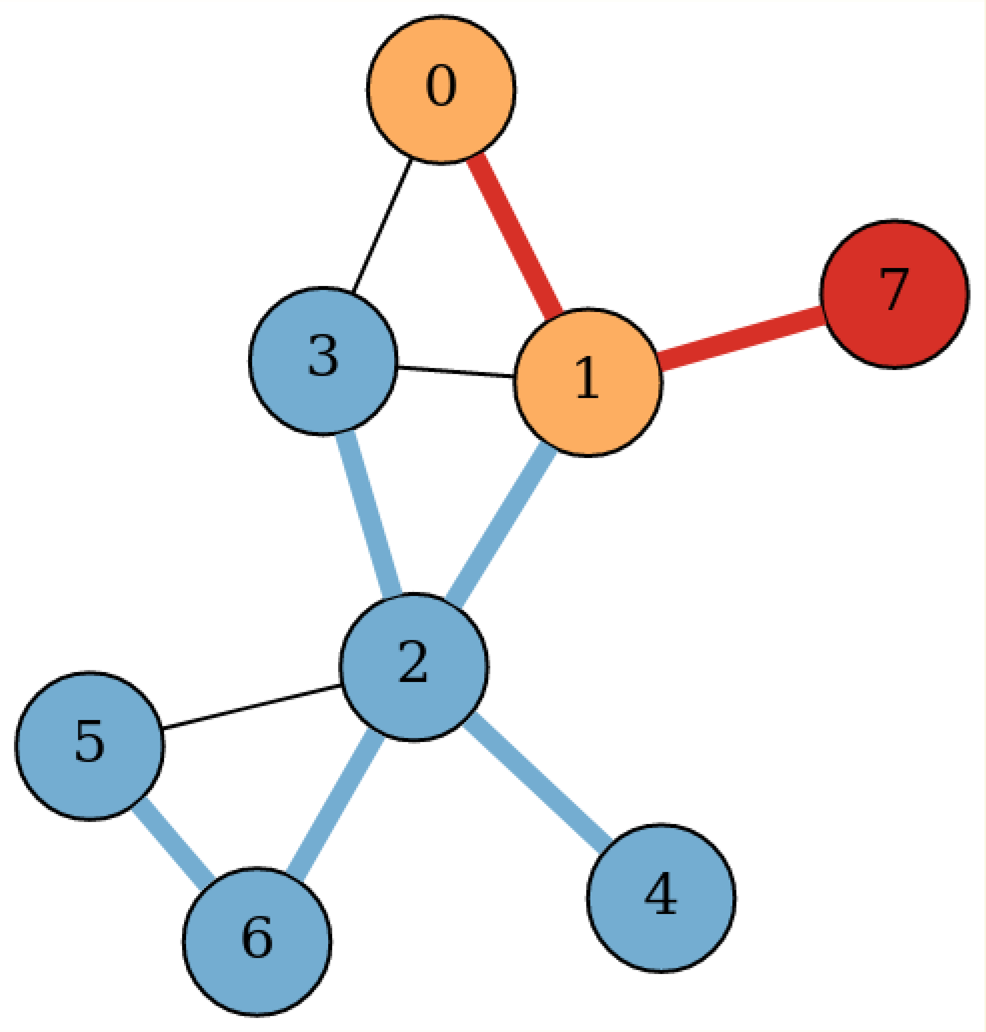
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH-FIRST SEARCH (DFS)

General idea:

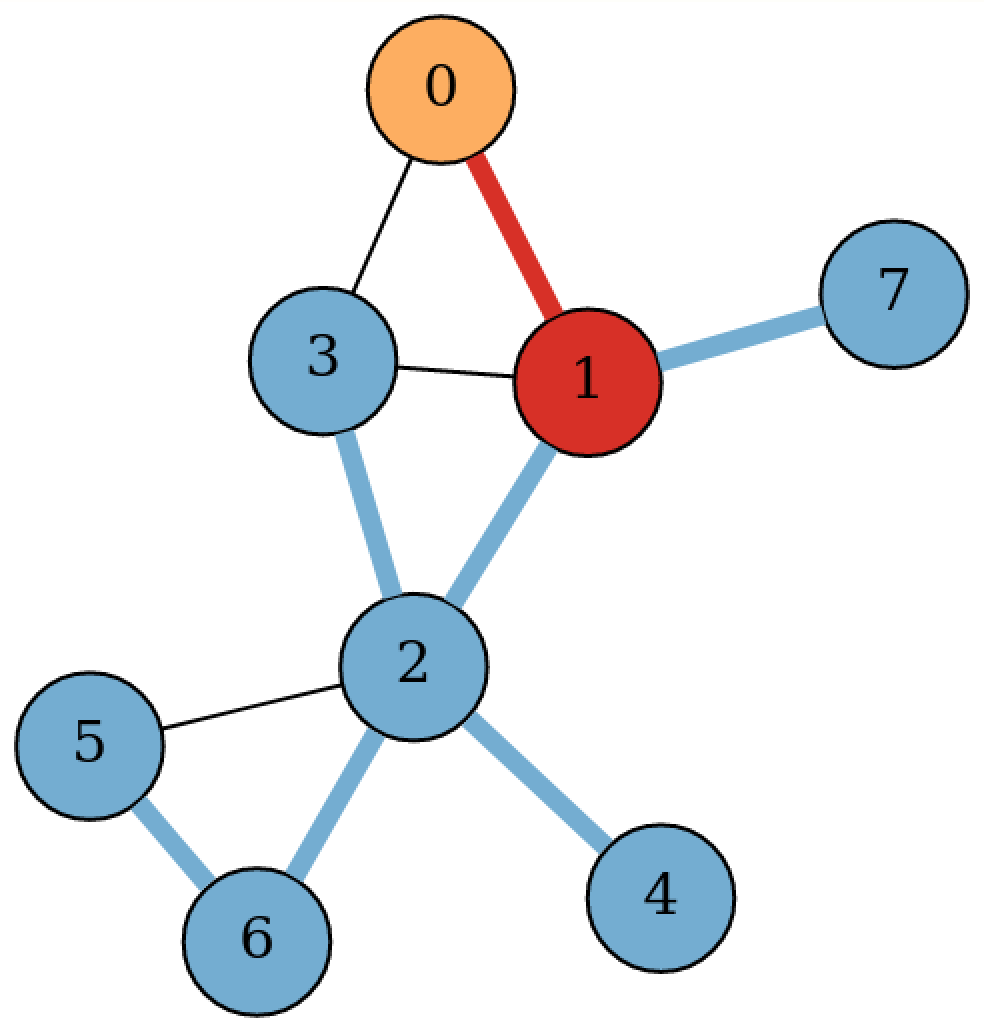
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH-FIRST SEARCH (DFS)

General idea:

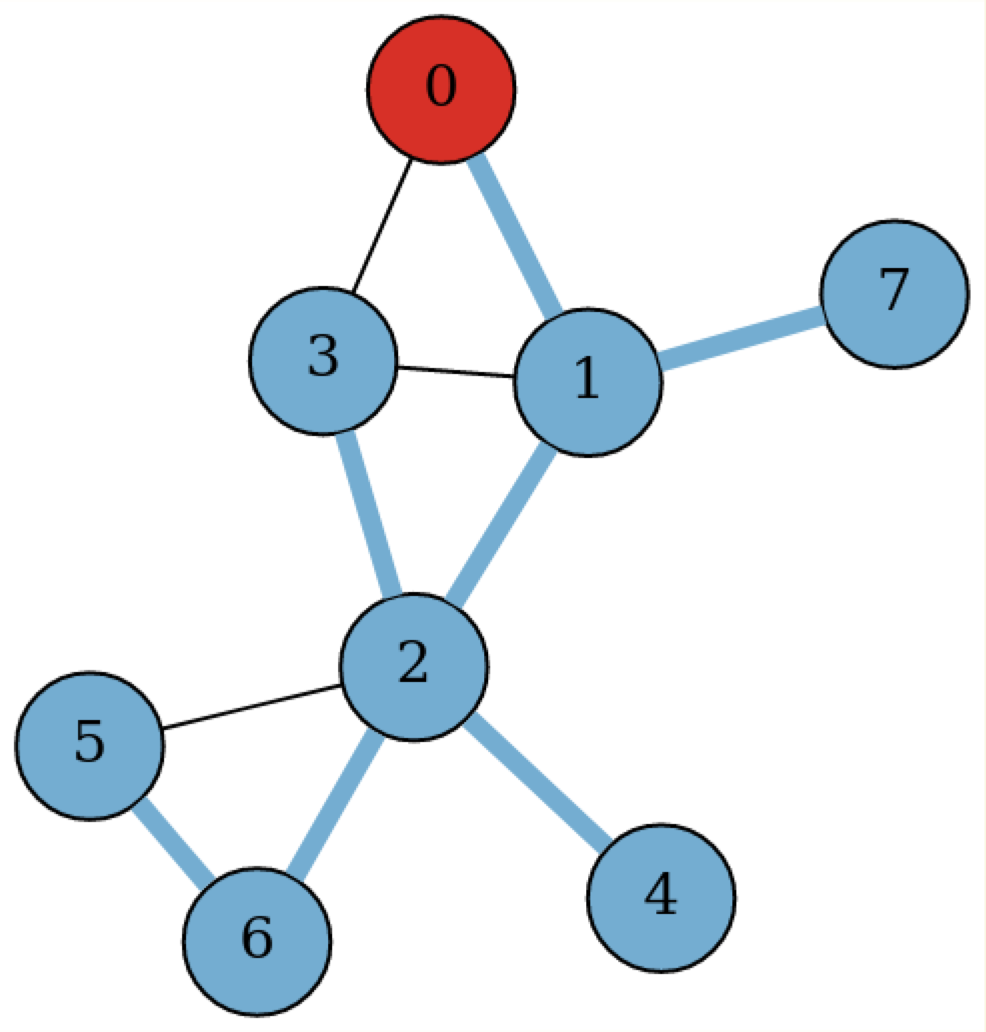
- keep going to an unvisited vertex
- when stuck make a step back and try again



DEPTH–FIRST SEARCH (DFS)

General idea:

- keep going to an unvisited vertex
- when stuck make a step back and try again



CONNECTED COMPONENTS VIA DFS

Recursive DFS exploration:

In [13]:
```
fn mark_component_dfs(vertex:Vertex, graph:&Graph, component:&mut Vec<Option<Component>>, component_no:Component) {
    component[vertex] = Some(component_no);
    for w in graph.outedges[vertex].iter() {
        if let None = component[*w] {
            mark_component_dfs(*w,graph,component,component_no);
        }
    }
}
```

CONNECTED COMPONENTS VIA DFS

Recursive DFS exploration:

In [13]:
```
fn mark_component_dfs(vertex:Vertex, graph:&Graph, component:&mut Vec<Option<Component>>, component_no:Component) {
    component[vertex] = Some(component_no);
    for w in graph.outedges[vertex].iter() {
        if let None = component[*w] {
            mark_component_dfs(*w,graph,component,component_no);
        }
    }
}
```

Going over all components and assigning vertices:

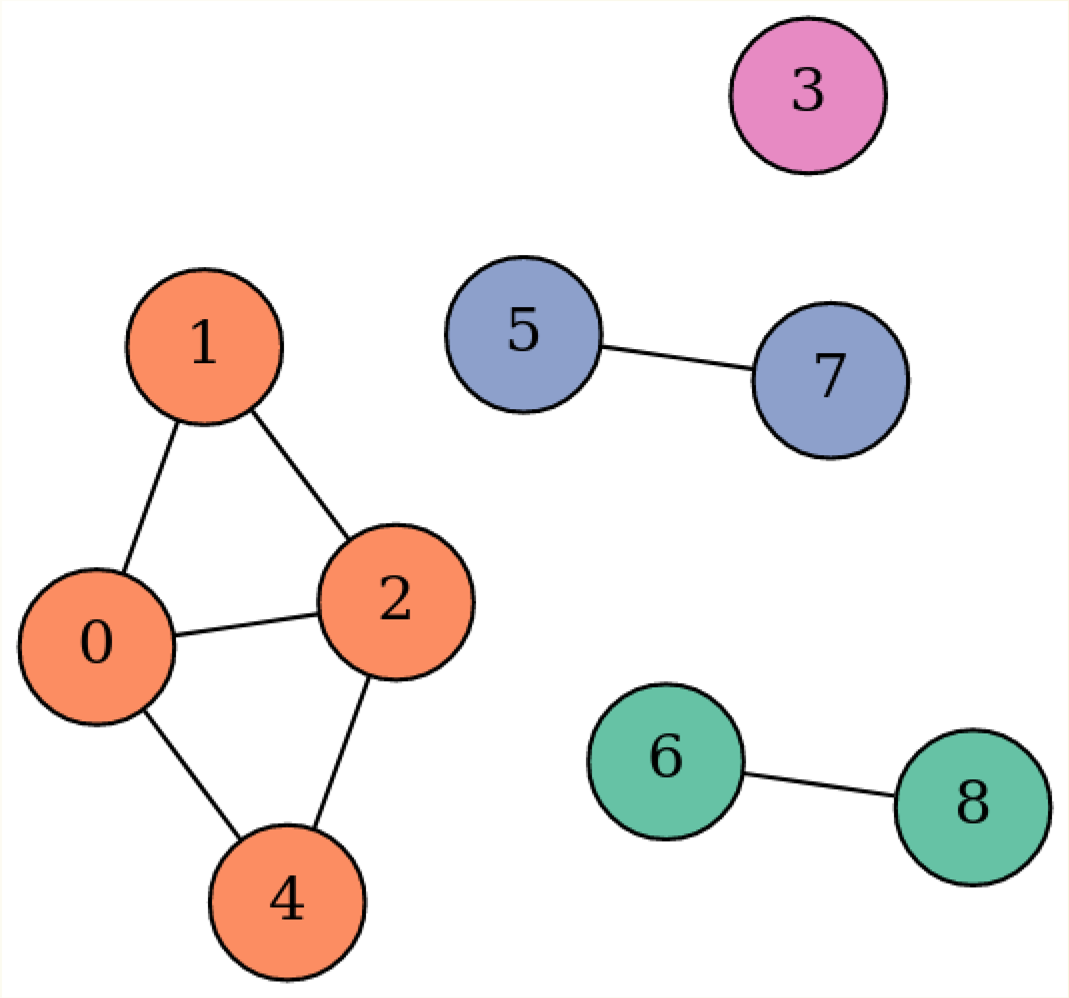
In [14]:
```
let mut component = vec![None;graph.n];
let mut component_count = 0;

for v in 0..graph.n {
    if let None = component[v] {
        component_count += 1;
        mark_component_dfs(v,&graph,&mut component,component_count);
    }
};
```

CONNECTED COMPONENTS VIA DFS

Let's verify the results:

In [15]:
```
print!("{} components:\n[   ",component_count);
for v in 0..n {
    print!("{}:{}  ",v,component[v].unwrap());
}
println!("]\n");
```

```
4 components:
[   0:1   1:1   2:1   3:2   4:1   5:3   6:4   7:3   8:4   ]
```



BFS VS. DFS

BFS

- gives graph distances between vertices (fundamental problem!)
- connectivity

BFS VS. DFS

BFS

- gives graph distances between vertices (fundamental problem!)
- connectivity

DFS

- What is it good for?

BFS VS. DFS

BFS

- gives graph distances between vertices (fundamental problem!)
- connectivity

DFS

- What is it good for?

LOTS OF THINGS!

Examples:

- find edges/vertices crucial for connectivity
- orient edges of a graph so it is still connected
- strongly connected components in directed graphs

1. GRAPH EXPLORATION OVERVIEW

2. BREADTH-FIRST SEARCH (BFS)

3. DEPTH-FIRST SEARCH (DFS)

4. BONUS CONTENT: STRONGLY CONNECTED COMPONENTS

STRONG CONNECTIVITY

What does connectivity mean in directed graphs?
What if you can get from v to w, but not from w to v?

STRONG CONNECTIVITY

What does connectivity mean in directed graphs?
What if you can get from v to w, but not from w to v?

Strongly connected component:
a maximal set of vertices such that you can get from any of them to any other one

STRONG CONNECTIVITY

What does connectivity mean in directed graphs?
What if you can get from v to w, but not from w to v?

Strongly connected component:
a maximal set of vertices such that you can get from any of them to any other one

Fact: There is a unique decomposition

FIND THE UNIQUE DECOMPOSITION VIA TWO DFS RUNS

GENERAL IDEA

First DFS:

- maintain auxiliary stack S
- visit all vertices, starting DFS multiple times from unvisited vertices as needed
- put each vertex, when done going over its neighbors, on the stack

FIND THE UNIQUE DECOMPOSITION VIA TWO DFS RUNS

GENERAL IDEA

First DFS:

- maintain auxiliary stack S
- visit all vertices, starting DFS multiple times from unvisited vertices as needed
- put each vertex, when done going over its neighbors, on the stack

Second DFS:

- **reverse edges of the graph!!!**
- consider vertices in order from the stack
- for each unvisited vertex, start DFS: it will visit a new strongly connected component

IMPLEMENTATION

In [16]:
```
let n: usize = 7;
let edges: ListOfEdges = vec![(0,1),(1,2),(2,0),(3,4),(4,5),(5,3),(2,3),(6,5)];
let graph = Graph::create_directed(n, &edges);
let graph_reverse = Graph::create_directed(n,&reverse_edges(&edges));
println!("{:?}\n{:?}",graph,graph_reverse);
```

```
Graph { n: 7, outedges: [[1], [2], [0, 3], [4], [5], [3], [5]] }
Graph { n: 7, outedges: [[2], [0], [1], [5, 2], [3], [4, 6], []] }
```

IMPLEMENTATION (FIRST DFS)

In [17]:
```
let mut stack: Vec<Vertex> = Vec::new();
let mut visited = vec![false;graph.n];
```

IMPLEMENTATION (FIRST DFS)

In [17]:
```
let mut stack: Vec<Vertex> = Vec::new();
let mut visited = vec![false;graph.n];
```

In [18]:
```
fn dfs_collect_stack(v:Vertex, graph:&Graph, stack:&mut Vec<Vertex>, visited:&mut Vec<bool>) {
    if !visited[v] {
        visited[v] = true;
        for w in graph.outedges[v].iter() {
            dfs_collect_stack(*w, graph, stack, visited);
        }
        stack.push(v);
    }
}
```

IMPLEMENTATION (FIRST DFS)

In [17]:
```
let mut stack: Vec<Vertex> = Vec::new();
let mut visited = vec![false;graph.n];
```

In [18]:
```
fn dfs_collect_stack(v:Vertex, graph:&Graph, stack:&mut Vec<Vertex>, visited:&mut Vec<bool>) {
    if !visited[v] {
        visited[v] = true;
        for w in graph.outedges[v].iter() {
            dfs_collect_stack(*w, graph, stack, visited);
        }
        stack.push(v);
    }
}
```

In [19]:
```
for v in 0..graph.n {
    dfs_collect_stack(v,&graph,&mut stack,&mut visited);
};
stack
```

Out[19]: [5, 4, 3, 2, 1, 0, 6]

IMPLEMENTATION (SECOND DFS, REVERSED GRAPH)

In [20]:
```
let mut component: Vec<Option<Component>> = vec![None;graph.n];
let mut component_count = 0;

while let Some(v) = stack.pop() {
    if let None = component[v] {
        component_count += 1;
        mark_component_dfs(v, &graph_reverse, &mut component, component_count);
    }
};
```

IMPLEMENTATION (SECOND DFS, REVERSED GRAPH)

In [20]:
```
let mut component: Vec<Option<Component>> = vec![None;graph.n];
let mut component_count = 0;

while let Some(v) = stack.pop() {
    if let None = component[v] {
        component_count += 1;
        mark_component_dfs(v, &graph_reverse, &mut component, component_count);
    }
};
```

In [21]:
```
print!("{} components:\n[  ",component_count);
for v in 0..n {
    print!("{}:{}  ",v,component[v].unwrap());
}
println!("]\n");
```

```
3 components:
[  0:2  1:2  2:2  3:3  4:3  5:3  6:1  ]
```