Tutorial 8: Graphs and Traversal I

October 17, 2022

Gu Zhenhao

** Partly adopted from tutorial slides by [Wang Zhi Jian](#).*

Midterm

How to do better... Is there still chance?

Common Mistakes

What is the time complexity for the following program?

```
void foo(int n){
  if (n < 1)
    return;
  for (int i = 0; i < n * n; i++)
    System.out.println("*");
  foo(n/2);
  foo(n/2);
}
```

Very close to our tutorial questions!

Common Mistakes

```
…
A is a linked list of length N;
A[i + K] += 1;
```

Linked lists have no random access!

```
while(true)
    // no break keyword inside
    do something
```

Infinite while loop, unable to end!

A Few Tips

1. **Focus on the basics!**
   - *What are the pros and cons of each data structure?*
   - *Which functions does the data structure support?*

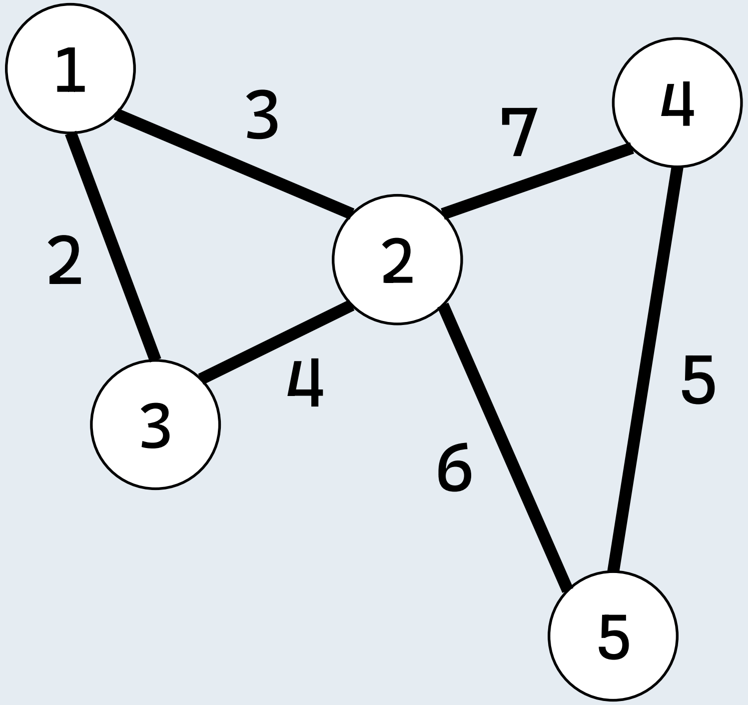
2. **Understand the problem-solving strategies in tutorials!**
   - *How to determine the runtime of an algorithm?*
   - *How to choose a good hash function?*
   - *Can we come up with a trivial answer first?*

Graph Representation

How to store a graph?

Graph Data Structure

- A set V of vertices,
- A set E of edges.
- The edge `[(u,v), w]` connects vertices u and v, and has a weight w (optional).

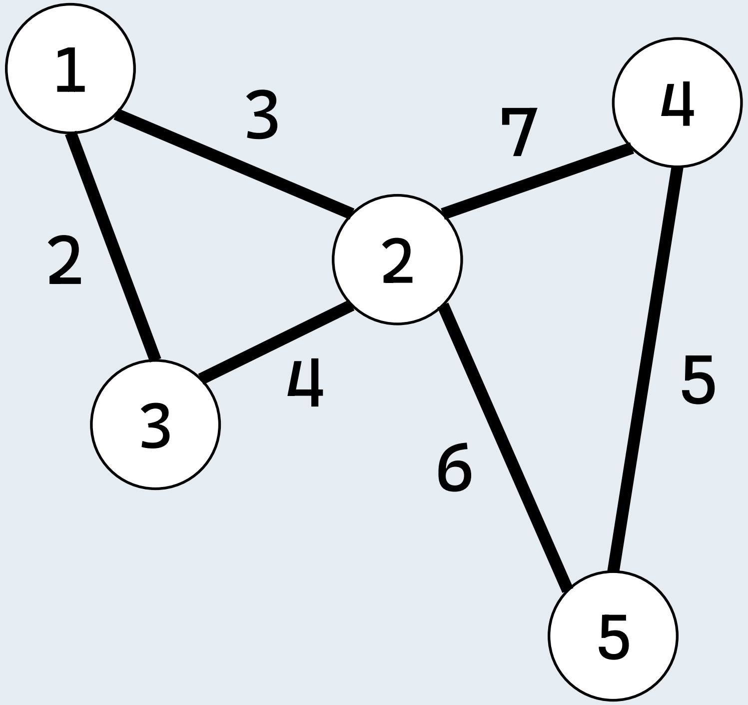


Representation of Graphs

As **edge list**:

- Simply store all edges in a list.

```
[(1,2),3], [(1,3),2], [(2,3),4],
[(2,4),7], [(2,5),6], [(4,5),5]
```



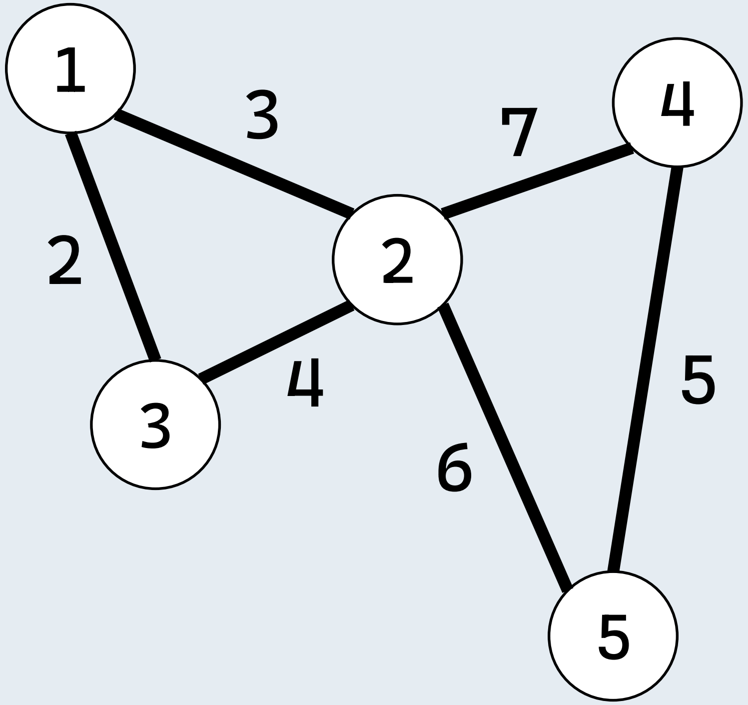
- **Space needed**: $O(|E|)$.
- **Time to query edge**: $O(|E|)$.

Representation of Graphs

As **adjacency list**:

- For each vertex, store its neighbours and weight.

```
1: [2,3], [3,2]
2: [1,3], [3,4], [4,7], [5,6]
3: [1,2], [2,4]
4: [2,7], [5,5]
5: [2,6], [4,5]
```



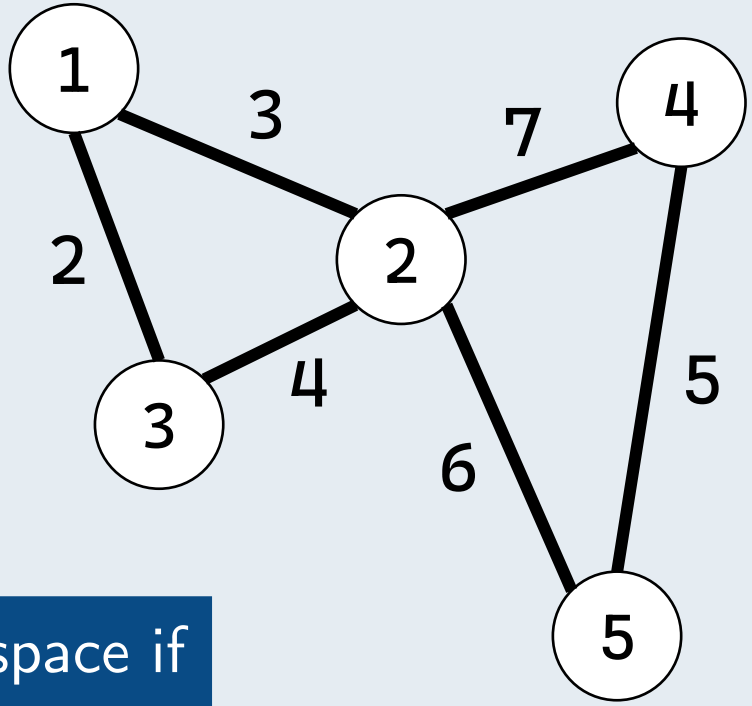
- **Space needed**: $O(|V| + |E|)$.
- **Time to query edge**: $O(d)$, where d is maximum degree.

Representation of Graphs

As **adjacency matrix**:

- Store the weight of (u, v) on u-th row and v-th column.

	1	2	3	4	5
1		3	2		
2	3		4	7	6
3	2	4			
4		7			5
5		6		5	



- **Space needed**: $O(|V|^2)$.
- **Time to query edge**: $O(1)$.

waste a lot of space if graph is sparse.

Graph Traversal

Two ways of graph traversal:

- **Depth-First Search (DFS)**: Explore the closest nodes to the last visited node first. (LIFO)
- **Breadth-First Search (BFS)**: Explore the closest nodes to the first visited node first. (FIFO)

Both costs $O(|V| + |E|)$ time.

Graph Traversal

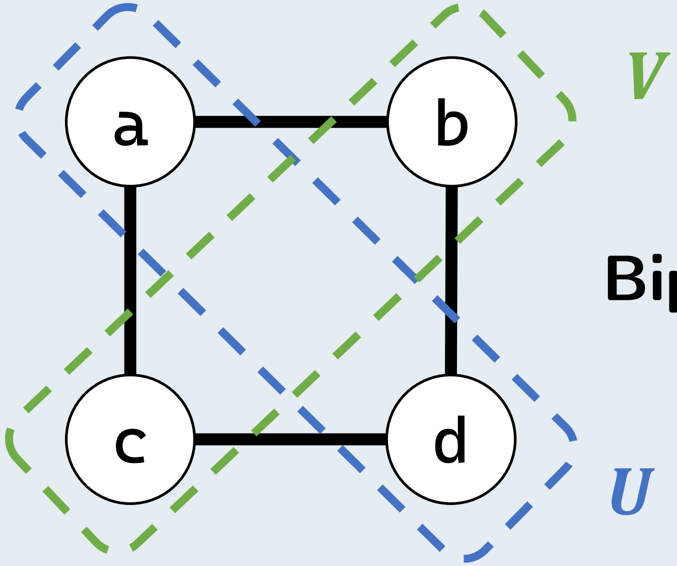
```
traversal(G, v):
    D.push(v);
    while D is not empty do
        v = D.pop();
        if v is not visited then
            visit(v);
            D.push(all unvisited neighbours of v);
```

For DFS: D is a stack
For BFS: D is a queue

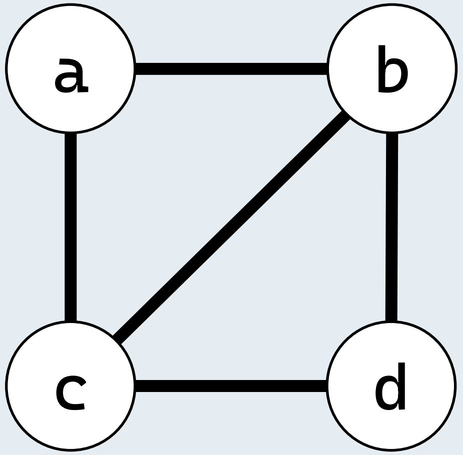
Problem 1

*A **bipartite graph** is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V; but there is no edge between vertices in U and also no edge between vertices in V.*

Goal: check if an undirected graph is bipartite.



Bipartite!

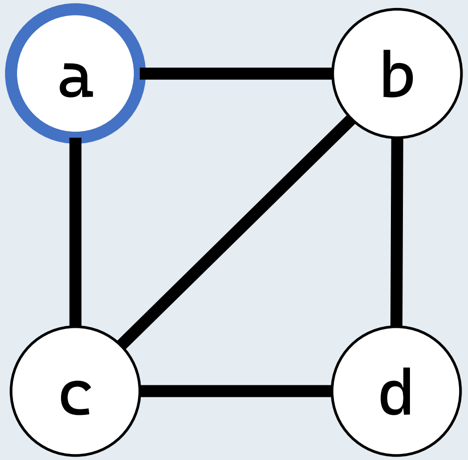


Is it bipartite?

Problem 1

Suppose this graph is bipartite.

- Suppose vertex **a** is in set U. **What do we know about other nodes?**

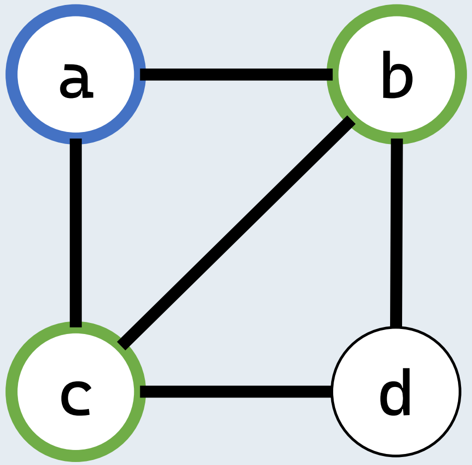


Problem 1

Suppose this graph is bipartite.

- Suppose vertex **a** is in set U.
- Vertices **b** and **c** must be in set V!
- But there's an edge between **b** and **c**... Contradiction!

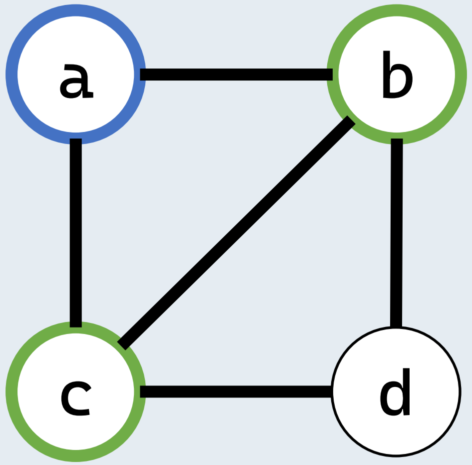
No way to divide nodes into 2 sets... **Not bipartite!**



Problem 1

Idea:

- Start from a vertex, assign it to set U.
- Traverse through the graph.
- As we visit a vertex, try to assign all neighbours to a different set.
- If a neighbour is already assigned the same set, it's not bipartite!



Problem 1

Algorithm 1 DFS for Bipartite Graph Detection

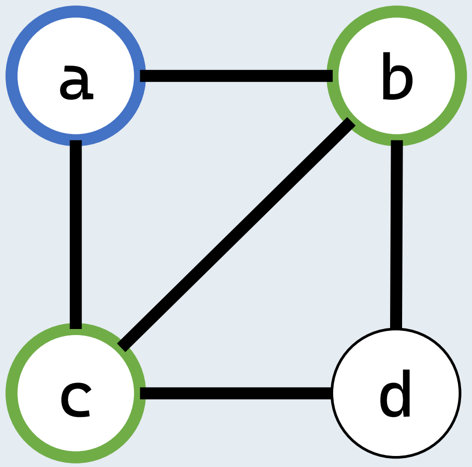
```
colour[1...n] ← WHITE                              ▷ Unvisited nodes are white
isBipartite ← true                                 ▷ Flag to indicate if graph is bipartite
procedure DFS(u, c)                                ▷ u is the current vertex, c is the colour to be assigned to u
    if colour[u] ≠ white then                      ▷ This node has been visited before
        if colour[u] ≠ c then                      ▷ Colour of this node is different from colour to be assigned
            isBipartite ← false
        end if
        return
    end if
    colour[u] ← c
    for each neighbour v of u do
        if c = BLUE then
            DFS(v, RED)
        else
            DFS(v, BLUE)
        end if
    end for
end procedure
```

- **Time complexity**: the same as DFS, $O(|V| + |E|)$.

Problem 1

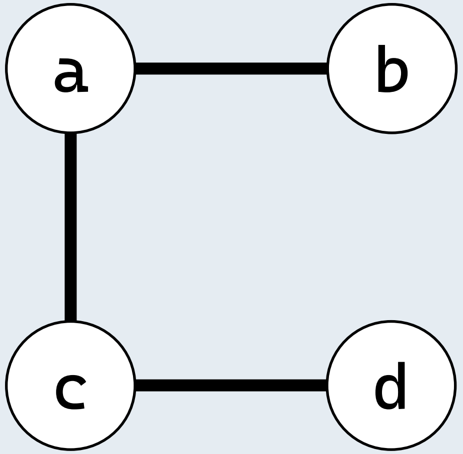
Note: Checking if a graph is ***bipartite*** is equivalent to

- Checking if the graph is ***2-colorable*** (use only 2 colours to colour the nodes, with all neighbouring nodes in different colours).
- Checking if there exists an ***odd-length cycle***.

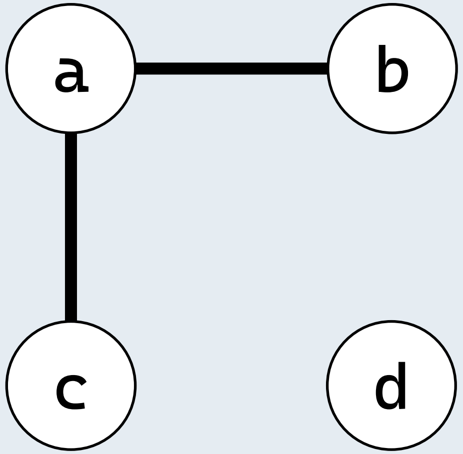


Problem 2.a

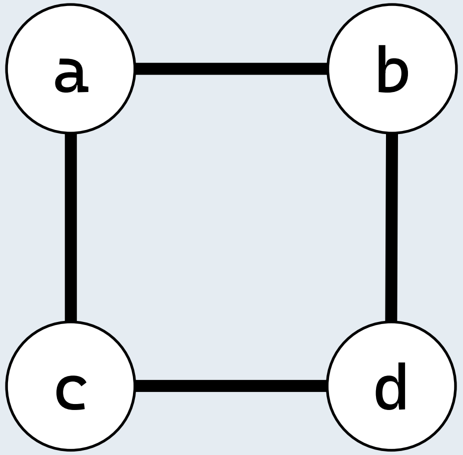
Check if an undirected graph contains cycles.



No cycle.



No cycle.

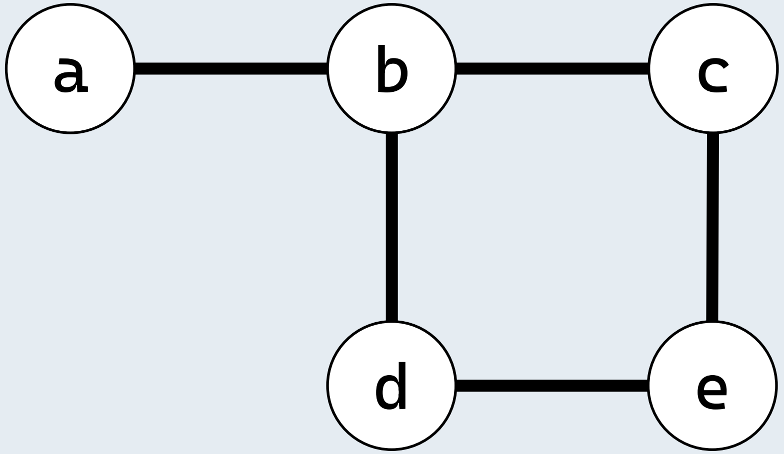


Has cycle!

Problem 2.a

If a graph has no cycles, then there's only one path connecting any pair of vertices.

If we **find 2 paths connecting a pair of vertices**, we are done!

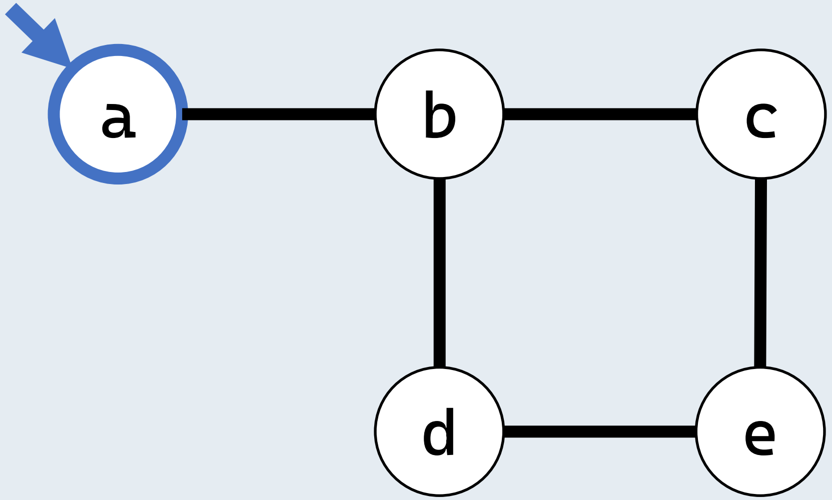


Use which graph traversal to try finding paths?

Problem 2.a

Let's try using DFS:

- Start from `a`, mark `a` as visited.

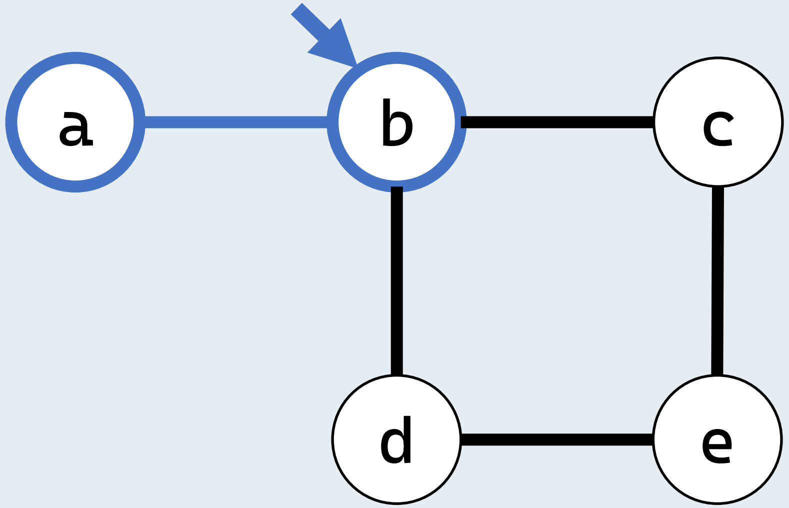


`stack: b`

Problem 2.a

Let's try using DFS:

- Start from **a**, mark **a** as visited.

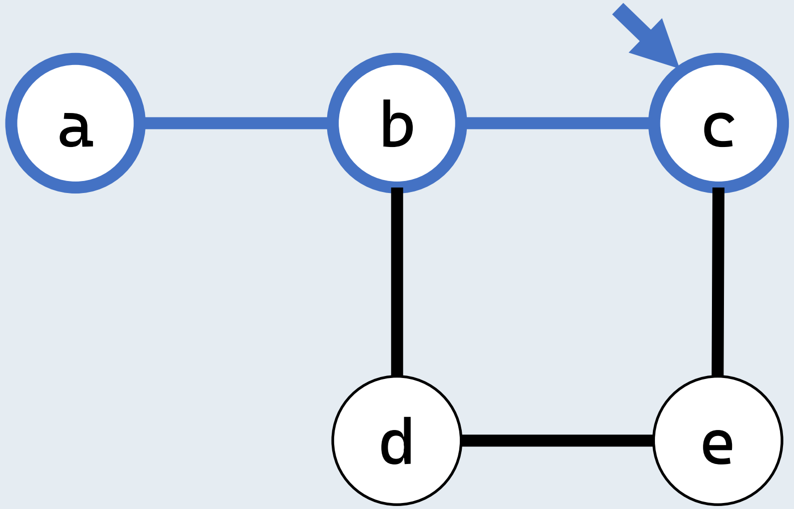


`stack: d, c`

Problem 2.a

Let's try using DFS:

- Start from `a`, mark `a` as visited.

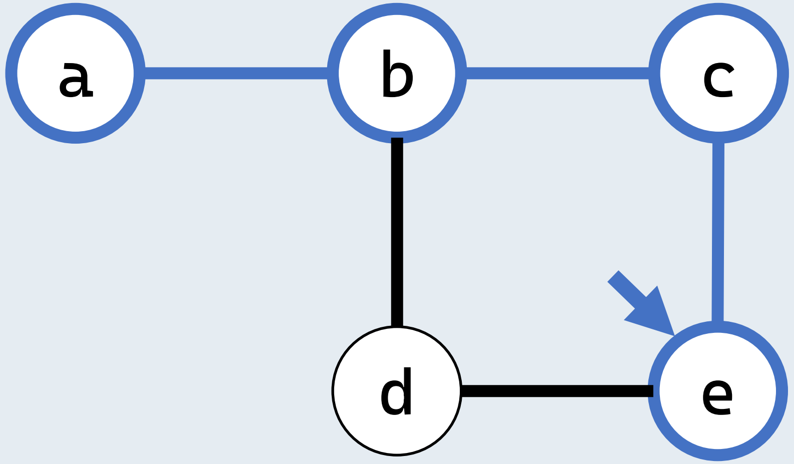


`stack: d, e`

Problem 2.a

Let's try using DFS:

- Start from `a`, mark `a` as visited.



`stack: d, d`

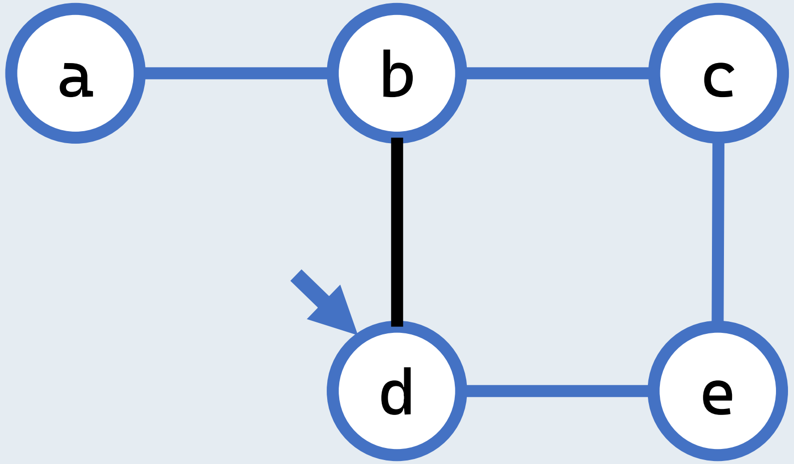
Problem 2.a

Let's try using DFS:

- Start from **a**, mark **a** as visited.
- Nothing to push to the stack... **b** is already visited, and **e** is where we came from. **What does this imply?**

There must be 2 paths from **b** to **d**... one passing through **e** and the other not!

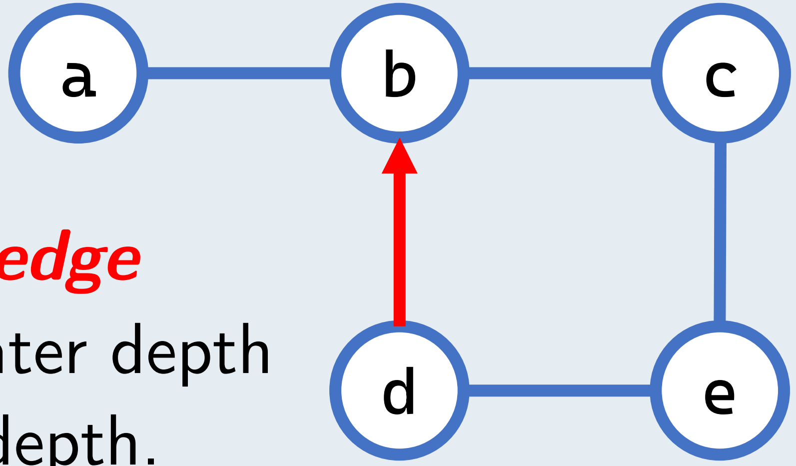
`stack: d`



Problem 2.a

Idea:

- Use DFS to traverse through the graph,
- If we find a neighbour is visited and it's not where we directly came from, claim that there's a cycle.



We call this a ***back edge*** from a vertex of greater depth to another at lower depth.

Note: DFS gives us a **spanning tree** connecting all vertices.

Problem 2.a

Algorithm 2 DFS for Cycle Detection in Undirected Graphs

```
visited[1 ... v] ← false
hasCycle ← false
procedure DFS(u, p)        ▷ u is the current vertex, p is the predecessor of u in the DFS tree
    visited[u] ← true
    for each neighbour v of u do
        if v ≠ p and visited[v] then
            hasCycle ← true
        end if
        DFS(v, u)
    end for
end procedure
```

Note: Need to run this on every connected component.

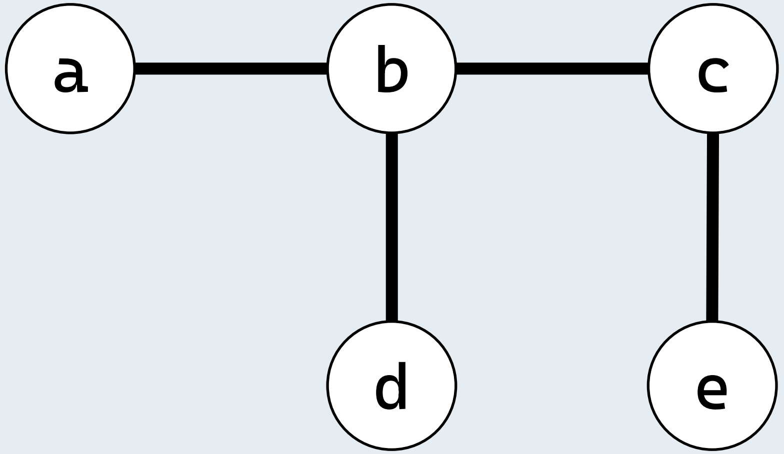
Pause and ponder: Can we use BFS instead of DFS?

Problem 2.a: Another idea

If an undirected graph has no cycles, then it must be a ***forest*** (graph containing one or several trees).

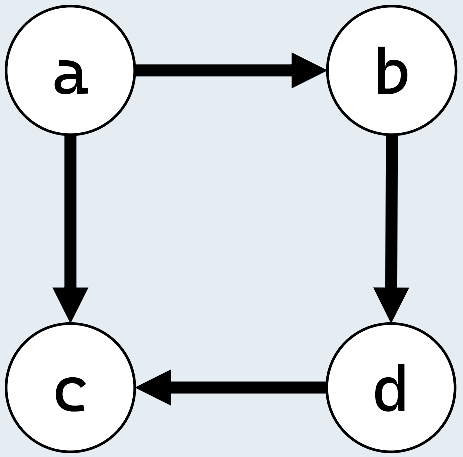
How to check if a connected graph is a tree?

In trees, $|E| = |V| - 1$!

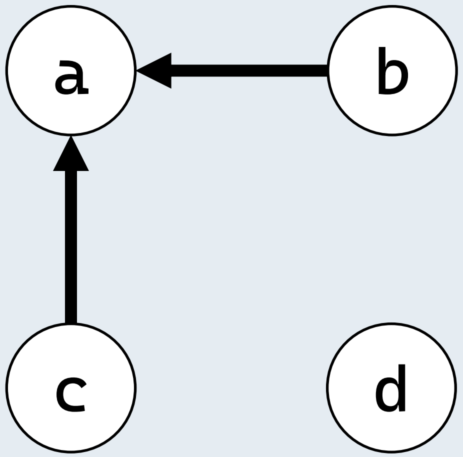


Problem 2.b

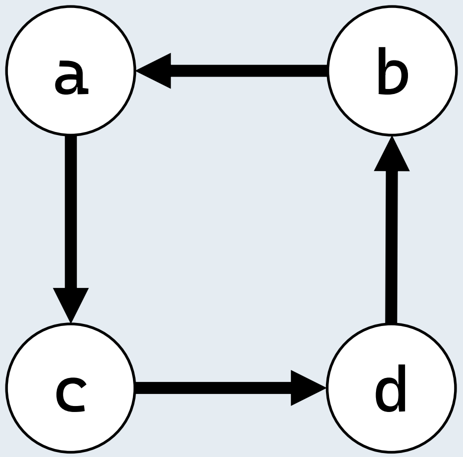
Check if a directed graph contains cycles.



No cycle.



No cycle.

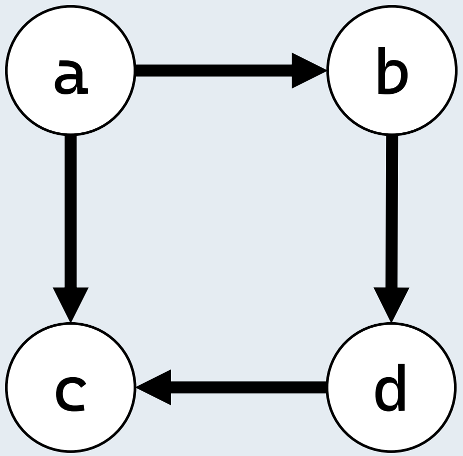


Has cycle!

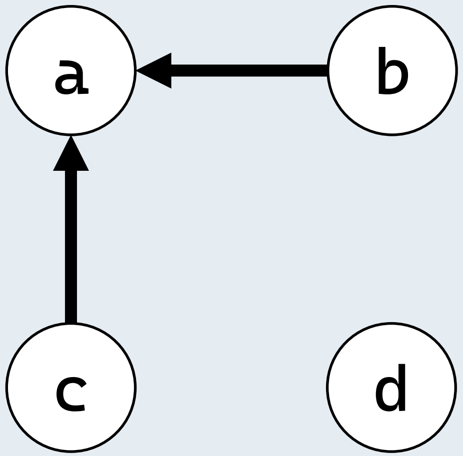
Problem 2.b

Question: Can we simply apply the previous traversal algorithm?

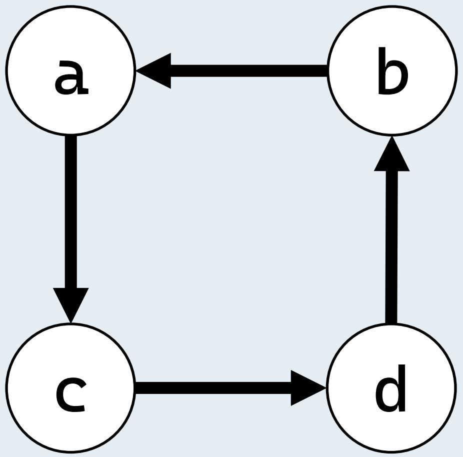
We have 2 paths from **a** to **c**... but this is not a cycle!



No cycle.



No cycle.

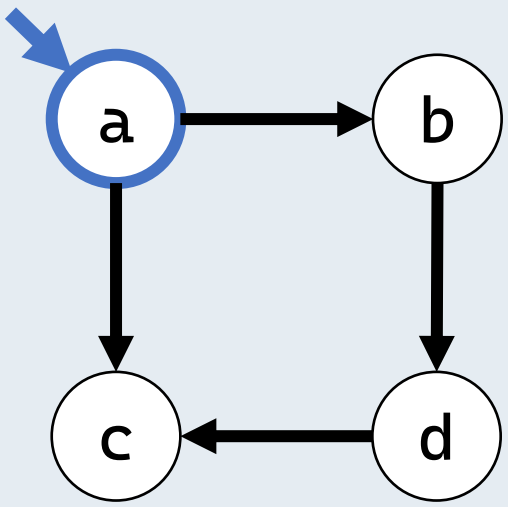


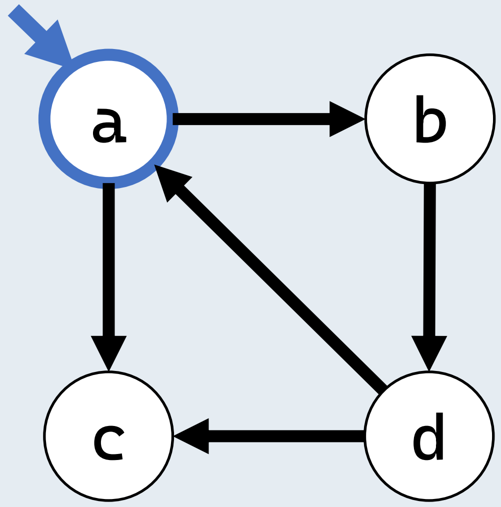
Has cycle!

Problem 2.b

Idea: Find a path that starts and ends on the same vertex!

Let's try DFS again...





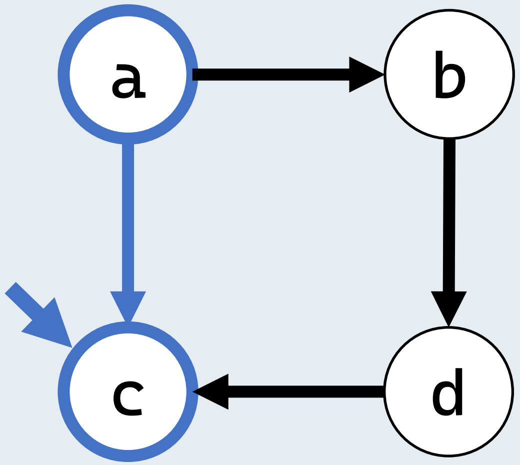
Problem 2.b

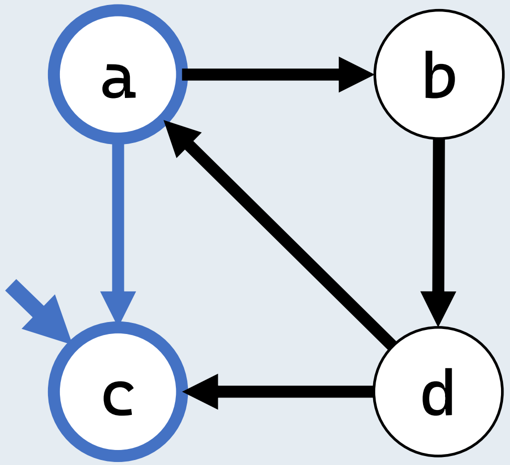
Idea: Find a path that starts and ends on the same vertex!

Let's try DFS again...

No more neighbors! **What do this imply?**

This path won't form a cycle!



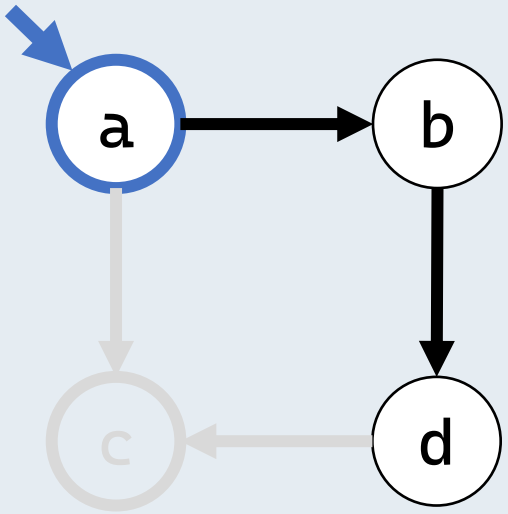


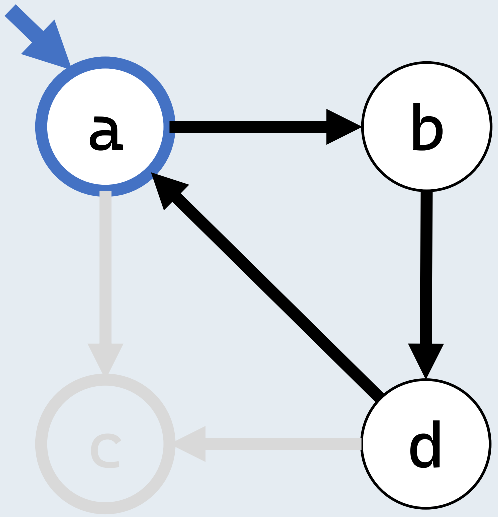
Problem 2.b

Idea: Find a path that starts and ends on the same vertex!

Let's try DFS again...

No more neighbors! **Might as well ignore the nodes on this path!**



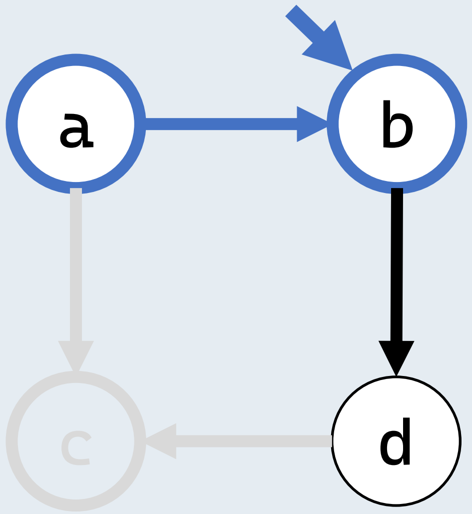


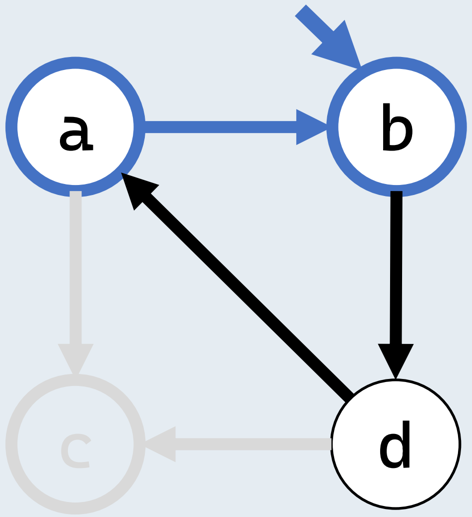
Problem 2.b

Idea: Find a path that starts and ends on the same vertex!

Let's try DFS again...

No more neighbors! **Might as well ignore the nodes on this path!**





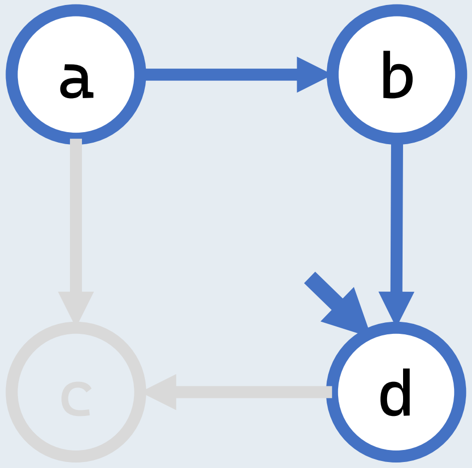
Problem 2.b

Idea: Find a path that starts and ends on the same vertex!

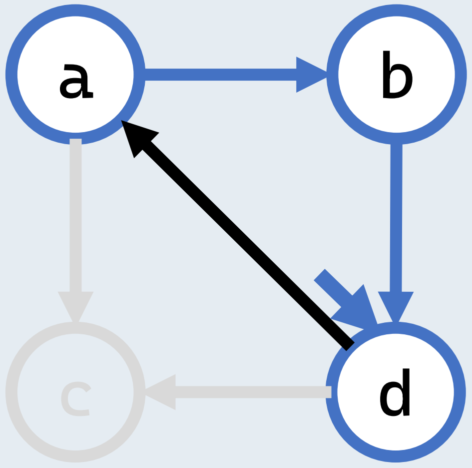
Let's try DFS again...

No more neighbors! **Might as well ignore the nodes on this path!**

Detect a visited neighbor **a**, cycle found!



No neighbors.



A visited neighbor a!

Problem 2.b

Algorithm 3 DFS for Cycle Detection in Directed Graphs

$status[1 \ldots v] \leftarrow NOT\_VISITED$
$hasCycle \leftarrow false$
procedure DFS(u, p) ▷ u is the current vertex, p is the predecessor of u in the DFS tree
    $status[u] \leftarrow VISITING$
    for each neighbour v of u **do**
        if $v \neq p$ **and** $status[v] = VISITING$ **then**
            $hasCycle \leftarrow true$
        end if
        DFS(v, u)
    end for
    $status[u] \leftarrow VISITED$
end procedure

Note: Need to ensure that in the end every node is visited.

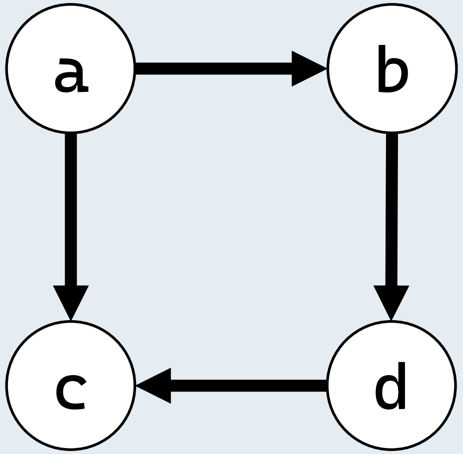
Pause and ponder: Can we use BFS instead of DFS?

Problem 2.b: Another idea

If a directed graph has no cycles, then it must be a ***directed acyclic graph (DAG)***.

How to check if graph is a DAG?

A DAG has a topological ordering!



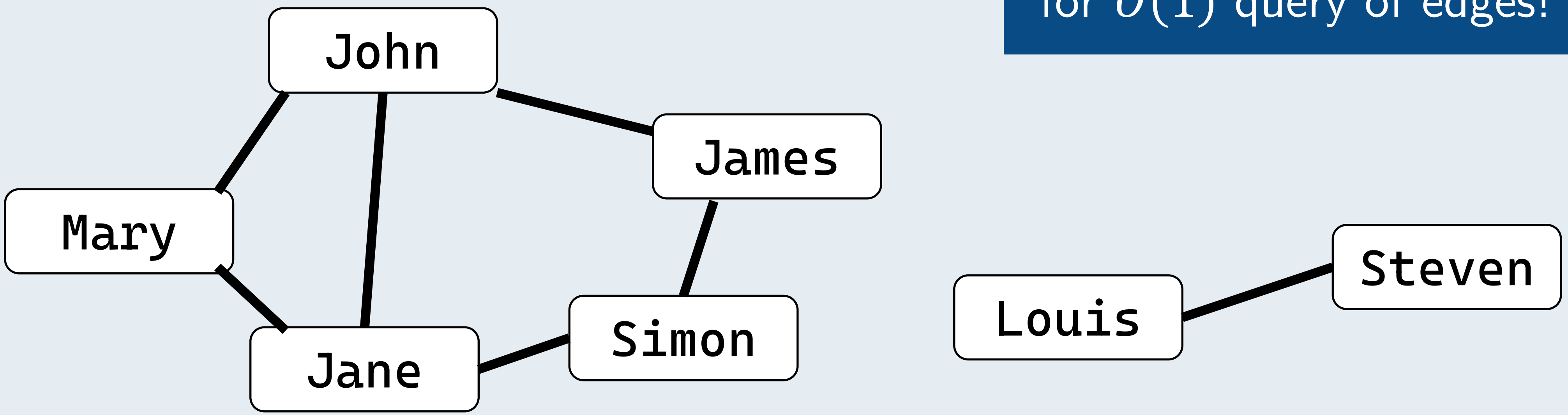
topological ordering: `a, b, d, c`

Problem 3.a,b

- We have an undirected graph with n vertices and m edges.

Goal: to quickly check if vertex X is connected to vertex Y.

Use **adjacency matrix** for $O(1)$ query of edges!

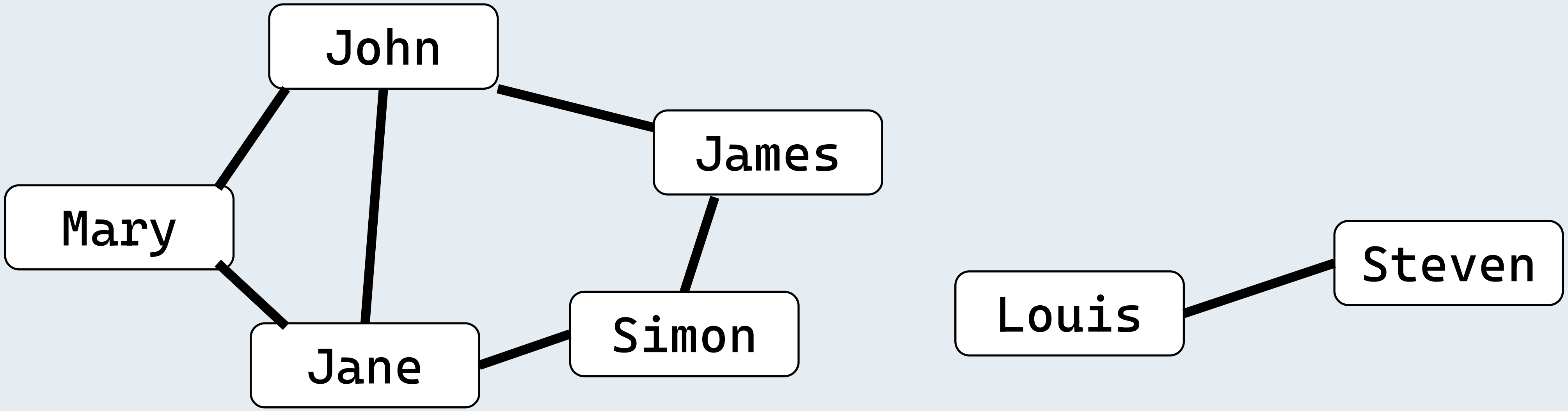


Problem 3.c,d

- We have an undirected graph with n vertices and m edges.

Goal: to quickly check **if there's a path** from vertex X to vertex Y.

Idea: start from X, do BFS/DFS and see if Y is reachable!



Graph Traversal

```
traversal(G, v):
    D.push(v);
    while D is not empty do
        v = D.pop();
        if v is not visited then
            visit(v);
            D.push(all unvisited neighbours of v);
```

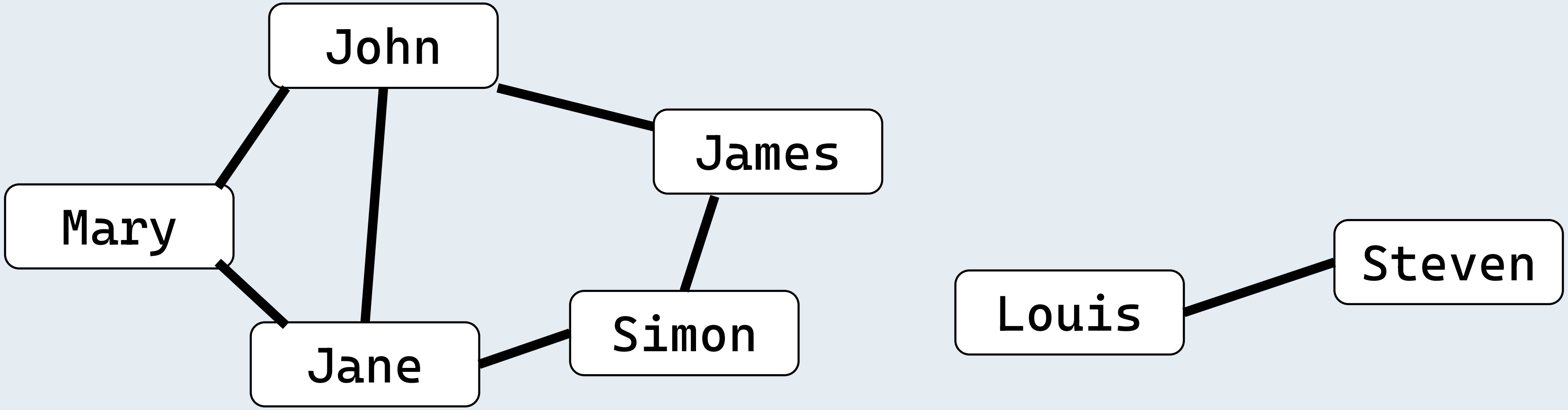
Use **adjacency list** to find the neighbors quickly!

Problem 3.e

- We have an undirected graph with n vertices and m edges.

Goal: answer k queries of whether there's a path from vertex X to vertex Y.

Observation: If 2 vertices are in the same connected component, then there's a path between the two!



Problem 3.e

- We have an undirected graph with n vertices and m edges.

Goal: answer k queries of whether there's a path from vertex X to vertex Y.

Observation: <u>If 2 vertices are in the same connected component, then there's a path between the two!</u>

Idea: use UFDS, store each connected component in a set.

```
{John, James, Mary, Jane, Simon}, {Louis, Steven}
```

Pre-processing: $O\big(m\alpha(n)\big)$ time, **Query**: $O\big(K\alpha(n)\big)$ time.

Problem 3.e

- We have an undirected graph with n vertices and m edges.

Goal: answer k queries of whether there's a path from vertex X to vertex Y.

Observation: <u>If 2 vertices are in the same connected component, then there's a path between the two!</u>

Another Idea: Give vertices in a connected component the same label.

```
John:1, James:1, Mary:1, Jane:1, Simon:1, Louis:2, Steven:2
```

Pre-processing: $O(m+n)$ time, **Query**: $O(K)$ time.

End of File

Thank you very much for your attention :-)