Graph Algorithms

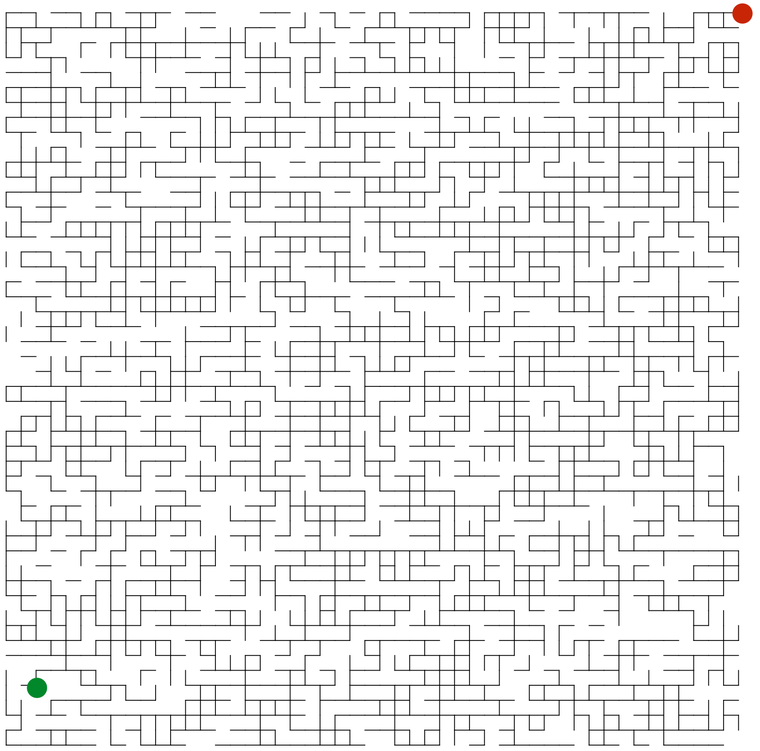
Graph Algorithms

For the next 3 weeks we will talk about computing properties of graphs.

There are 5 main problems we will discuss:

- Finding connected components (depth/breadth first search)
- Finding a minimum spanning tree
- Finding shortest paths between vertices
- Finding negative cycles in a graph with negative edge weights
- Finding a cycle in a directed graph and topological sorting.

Depth/Breadth First Search



Depth-first search is how you might find your way out of a maze with a ball of string and a piece of chalk.

It can be used to determine if there is a path between two vertices, and more generally all the vertices that can be reached from a source vertex.

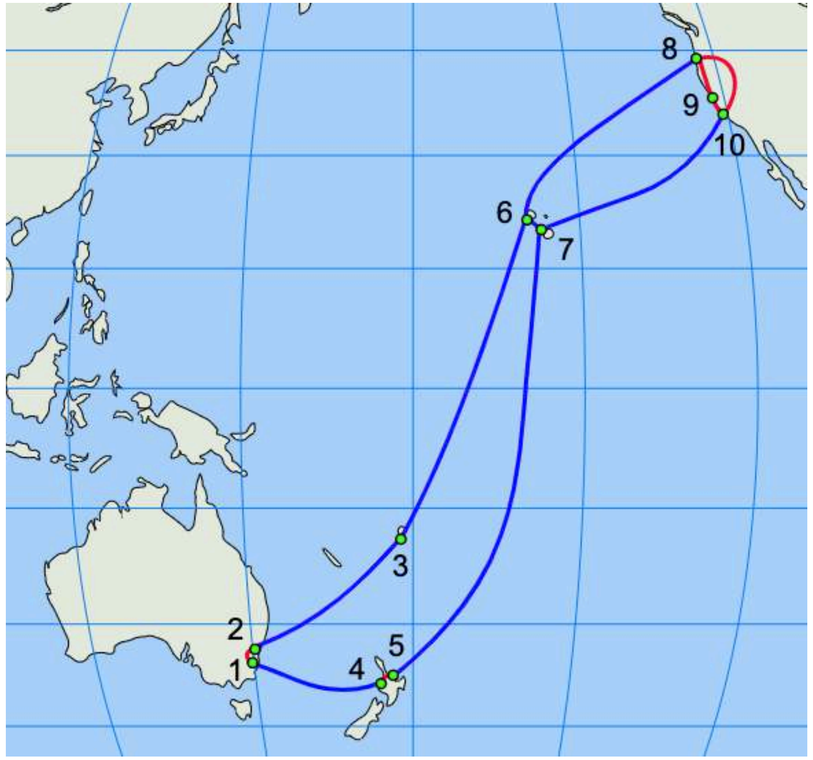
Depth- and Breadth-first search are simple examples of a more general graph searching paradigm that can be used for several different problems like...

Minimum Spanning Tree

Given a connected graph find a tree connecting all vertices of minimum total edge weight.

An algorithm for this problem was developed as early as 1926 from a practical application: find the least amount of wire to connect a set of cities.

At right the southern cross cable route.
~$50K USD per km to construct.



By J.P.Lon, Mysid - Manually vectorized in Inkscape by User:Mysid on a PNG originally by User:J.P.Lon., CC BY-SA 3.0

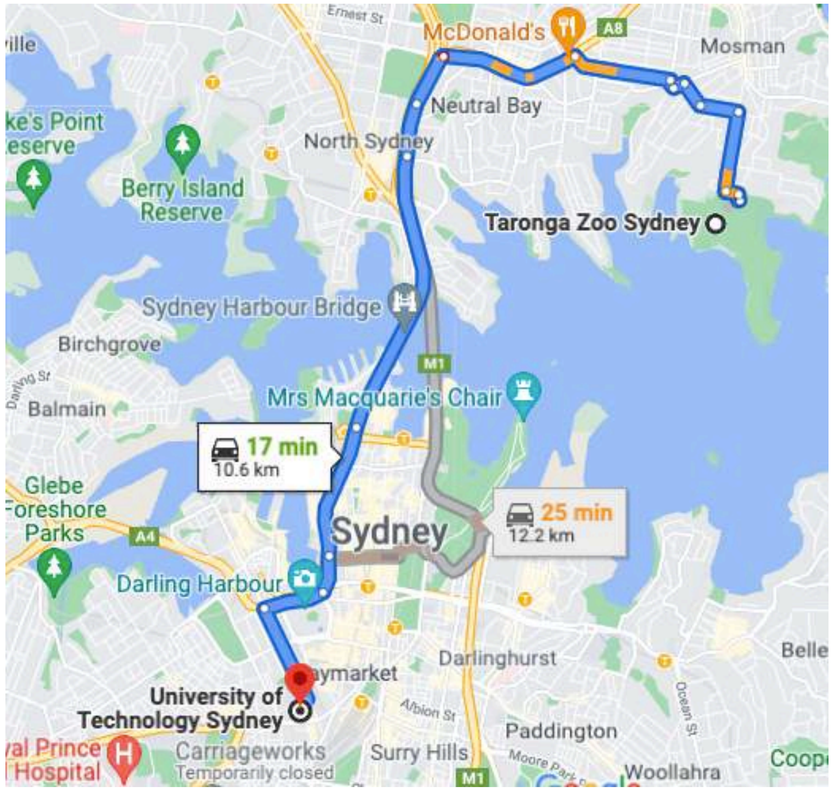
Shortest paths

Finding a shortest path between two points is a problem you probably solve every day.

Shortest could mean least distance, time, or money.

That is an advantage of abstracting the problem to a weighted graph.

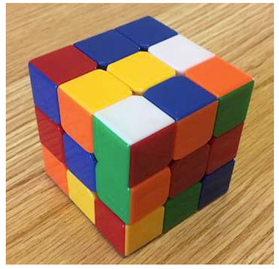
The same algorithm can be used no matter the notion of "distance".

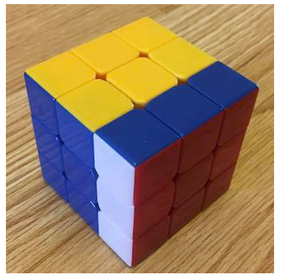


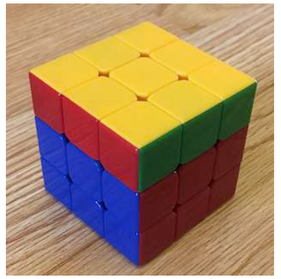
Shortest paths

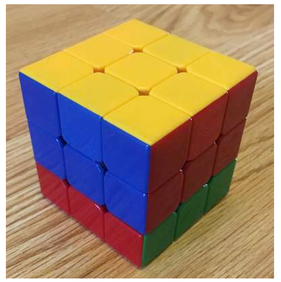
Solving a Rubik's cube also traces a path through a graph.

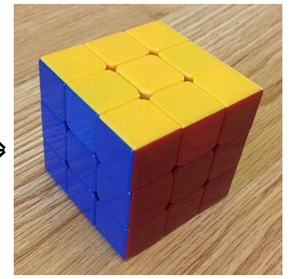
This graph has ~ $4.3 \cdot 10^{19}$ vertices.









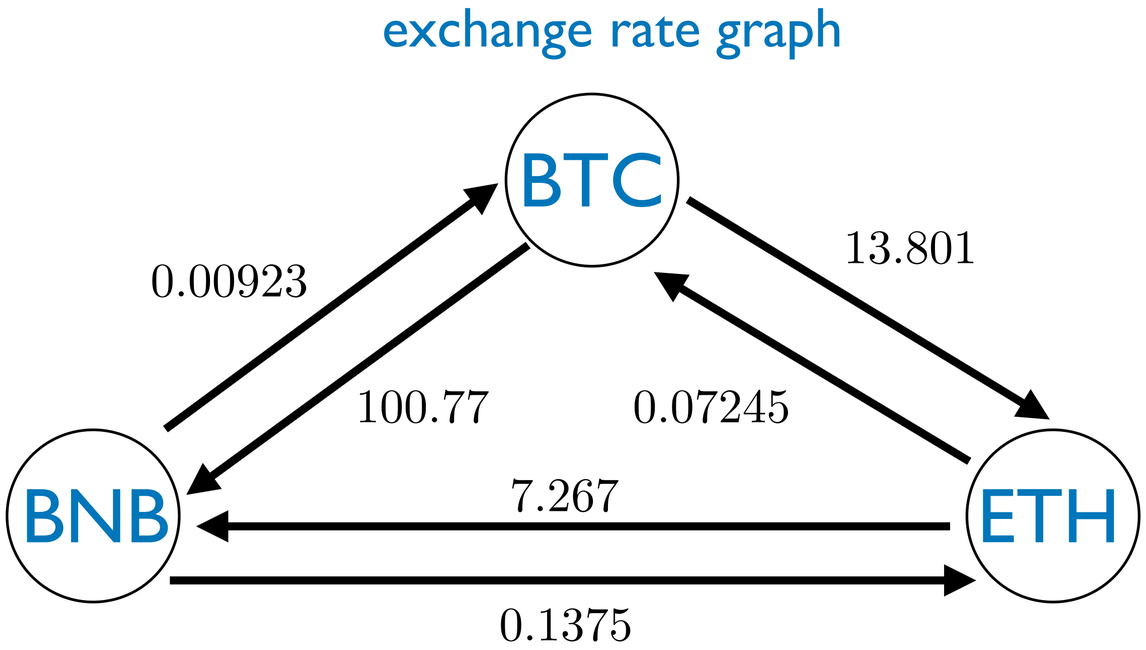


There is a path of length at most 20 between any two vertices in the "half turn metric."

Negative Cycles

Usually we think about the weight of an edge as a positive number representing distance, time, or money.

But negative weights also arise in applications!



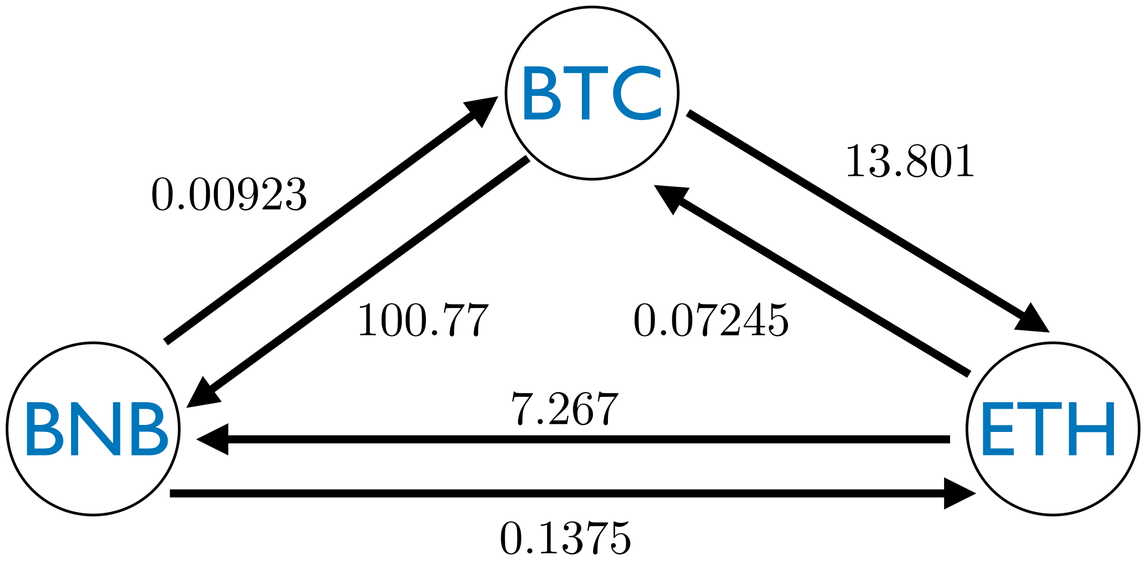
Binance Feb 3, 2022

Negative Cycles

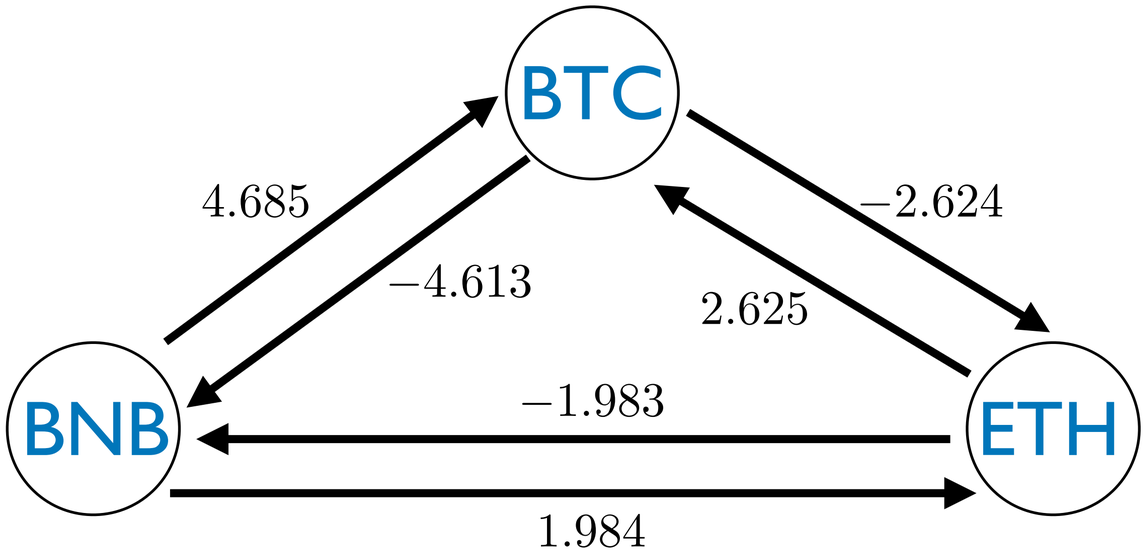
Usually we think about the weight of an edge as a positive number representing distance, time, or money.

But negative weights also arise in applications!

exchange rate graph



negative log of exchange rates



Binance Feb 3, 2022

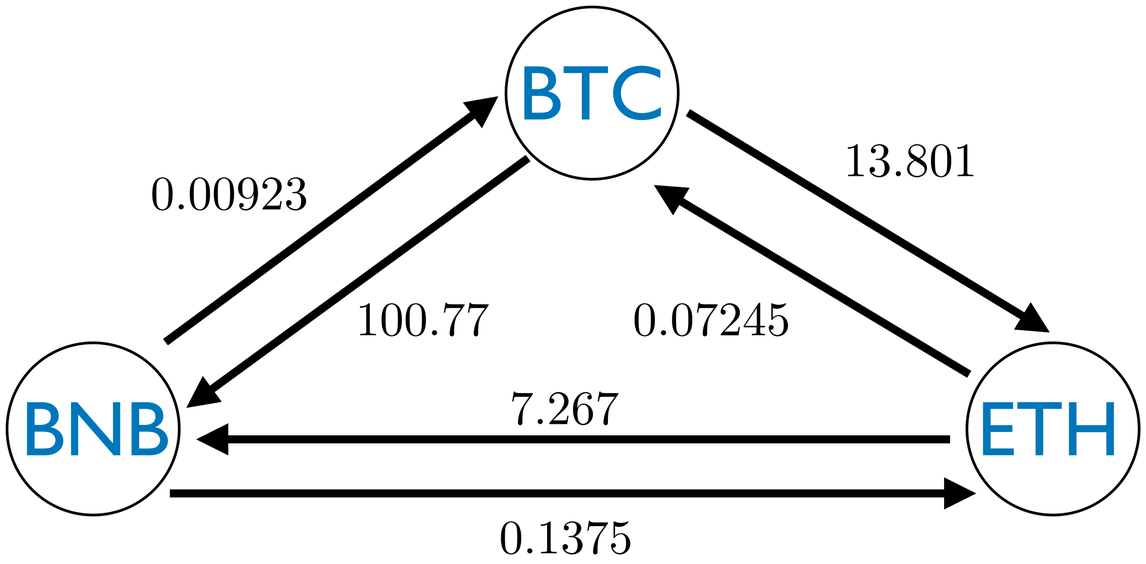
Negative Cycles

Usually we think about the weight of an edge as a positive number representing distance, time, or money.

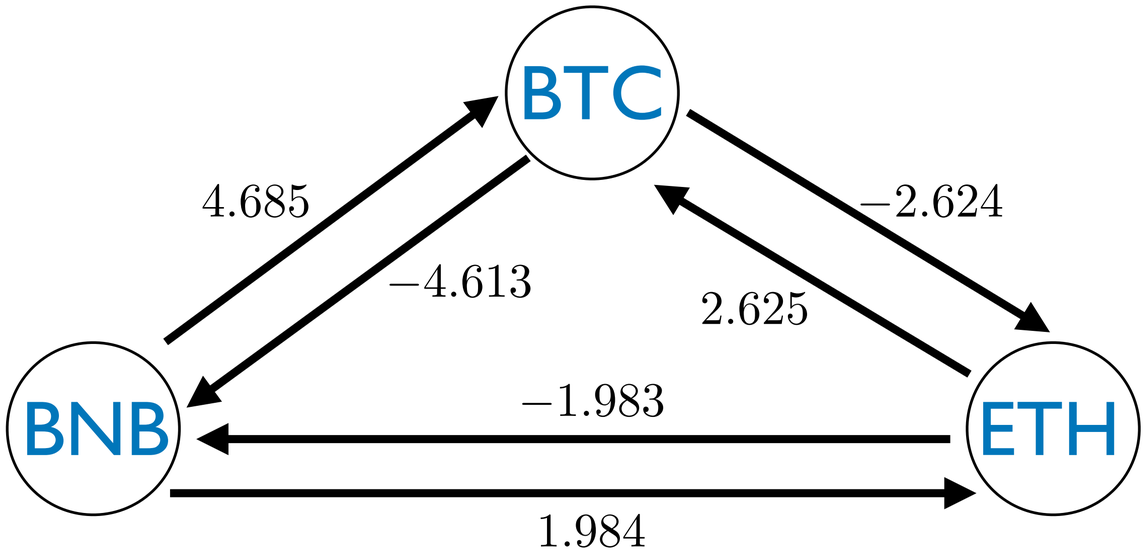
But negative weights also arise in applications!

Bellman-Ford algorithm

exchange rate graph



negative log of exchange rates



Binance Feb 3, 2022

Types of graphs

These examples have shown several different kinds of graphs.

Edges can be

- Directed or undirected
- Weighted or simply present/absent.

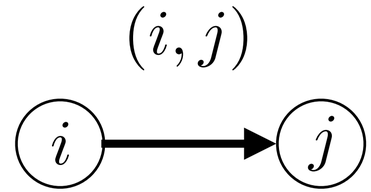
weighted & directed	unweighted & directed	weighted & undirected	unweighted & undirected
exchange rates	street atlas	distance between cities	Rubik's cube graph

Edge Notation

An edge must have a non-zero weight.

A directed edge is an ordered pair of vertices (i, j).

This means an edge leaving i directed toward j.

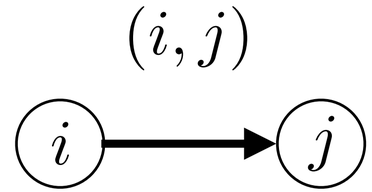


Edge Notation

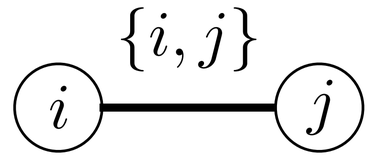
An edge must have a non-zero weight.

A directed edge is an ordered pair of vertices (i, j).

This means an edge leaving i directed toward j.



An undirected edge is a set of 2 vertices $\{i, j\}$.



Graph Terminology

Undirected graph:

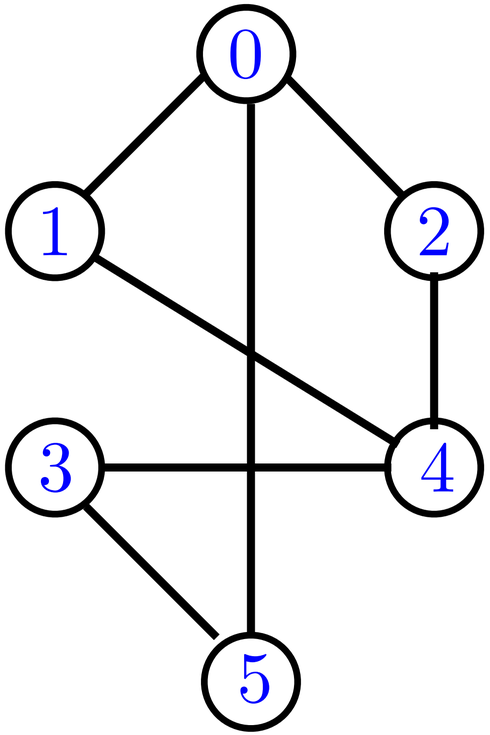
Two vertices are adjacent if and only if there is an edge between them.

The endpoints of the edge $\{u, v\}$ are the vertices u and v.

An edge is incident to a vertex v iff v is one of its endpoints.

The degree of v is the number of edges incident to v.

Example



In an n-vertex graph we will identify the vertices with the numbers $0, 1, \ldots, n-1$.

In this graph vertex 2 and 4 are adjacent.

Vertex 1 and 5 are not adjacent.

The endpoints of $\{3, 5\}$ are vertex 3 and 5.

The degree of vertex 0 is 3.

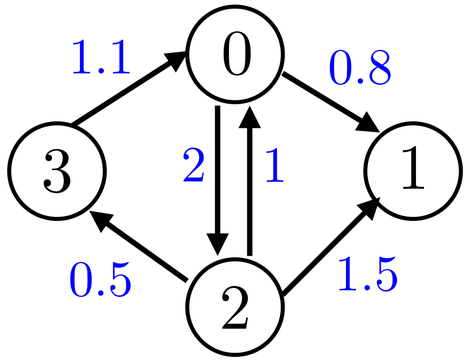
Graph Terminology

Directed graph:

For a directed edge (u, v) we say v is out-adjacent to u , and u is in-adjacent to v .

The out-degree of v is the number of edges directed out of it.

The in-degree of v is the number of edges directed into it.



The out-degree of vertex 2 is 3 and the in-degree is 1.

Graph Notation

Both directed and undirected:

We write an unweighted graph as $G = (V, E)$ and a weighted one as $G = (V, E, w)$ where $w : E \to \mathbb{R}$.

Here E is the set of edges.

In the directed case E will be a set of ordered pairs of V, and in the undirected case a set of 2-element subsets of V.

Data Structures

Graph data structures

What operations do we expect from a graph data structure?

Graph data structures

What operations do we expect from a graph data structure?

- add an edge.

Graph data structures

What operations do we expect from a graph data structure?

- add an edge.
- is there an edge between vertices u and v? What is its weight?

Graph data structures

What operations do we expect from a graph data structure?

- add an edge.
- is there an edge between vertices u and v? What is its weight?
- iterate through the vertices (out)-adjacent to v.

Graph data structures

What operations do we expect from a graph data structure?

- add an edge.
- is there an edge between vertices u and v? What is its weight?
- iterate through the vertices (out)-adjacent to v.

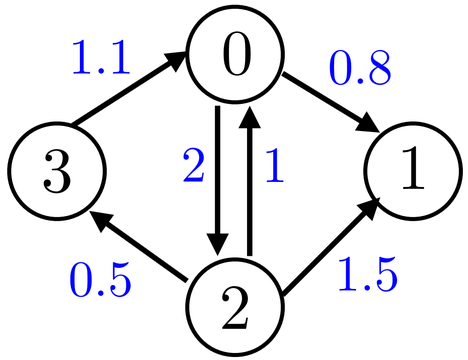
Graph data structures

What operations do we expect from a graph data structure?

- add an edge.
- is there an edge between vertices u and v? What is its weight?
- iterate through the vertices (out)-adjacent to v.
- remove an edge.
- return the (out)-degree of a vertex.
- add/remove a vertex.

Graph Data Structures

There are two standard representations of a graph $G = (V, E, w)$:

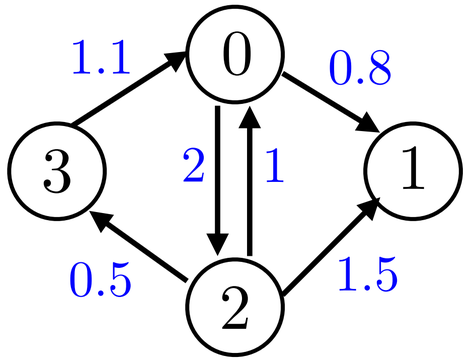


Graph Data Structures

There are two standard representations of a graph $G = (V, E, w)$:

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0 & 0 \end{bmatrix}$$



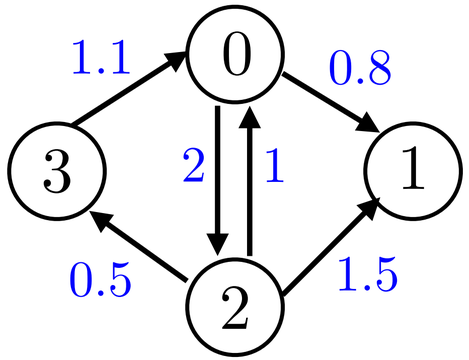
$$A[i][j] = \begin{cases} 0 & \text{if } (i,j) \notin E \\ w((i,j)) & \text{otherwise} \end{cases}$$

Graph Data Structures

There are two standard representations of a graph $G = (V, E, w)$:

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0 & 0 \end{bmatrix}$$



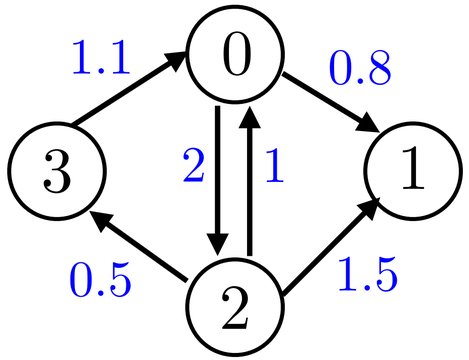
$$A[i][j] = \begin{cases} 0 & \text{if } (i,j) \notin E \\ w((i,j)) & \text{otherwise} \end{cases}$$

Graph Data Structures

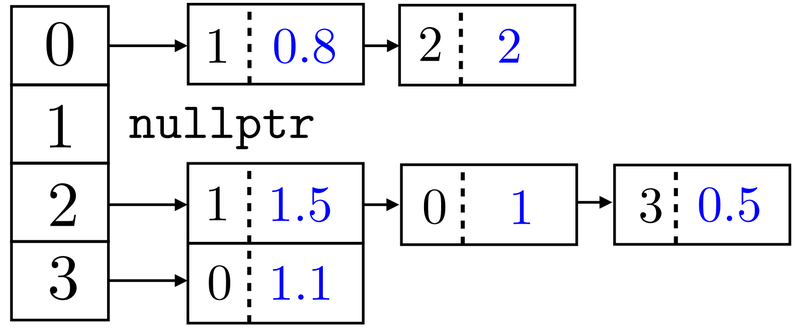
There are two standard representations of a graph $G = (V, E, w)$:

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0 & 0 \end{bmatrix}$$



Adjacency list



$$A[i][j] = \begin{cases} 0 & \text{if } (i,j) \notin E \\ w((i,j)) & \text{otherwise} \end{cases}$$

For each vertex we have a singly linked list of its out-adjacent vertices and assoc. weight.

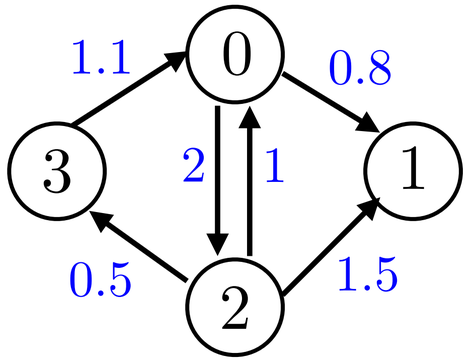
Ordering is arbitrary.

Graph Data Structures

There are two standard representations of a graph $G = (V, E, w)$:

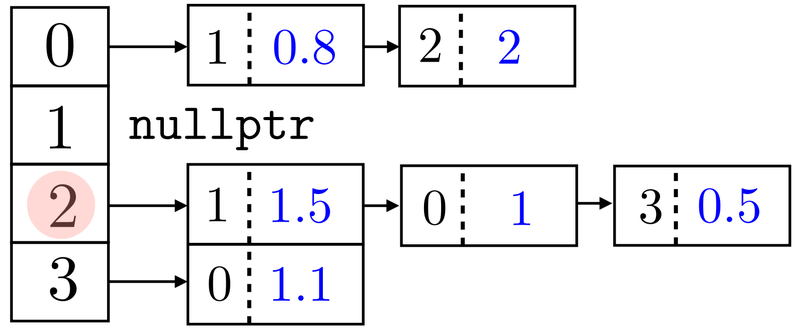
Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0 & 0 \end{bmatrix}$$



$$A[i][j] = \begin{cases} 0 & \text{if } (i,j) \notin E \\ w((i,j)) & \text{otherwise} \end{cases}$$

Adjacency list



For each vertex we have a singly linked list of its out-adjacent vertices and assoc. weight.

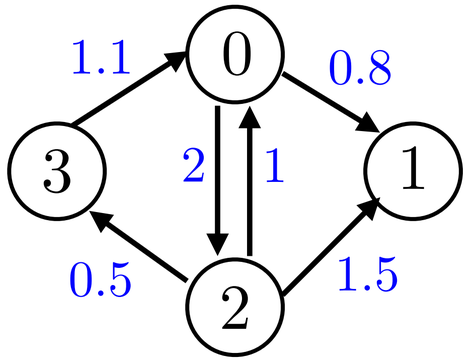
Ordering is arbitrary.

Graph Data Structures

There are two standard representations of a graph $G = (V, E, w)$:

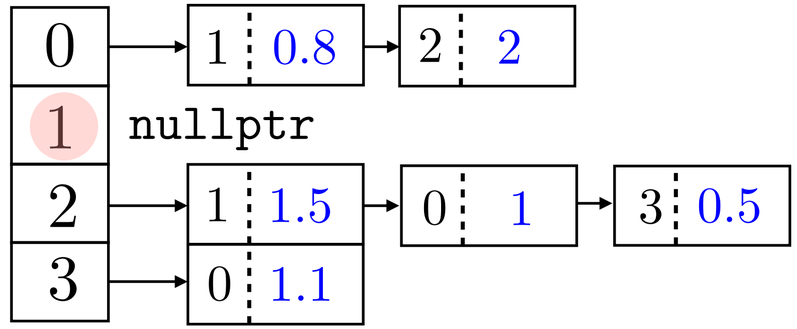
Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0 & 0 \end{bmatrix}$$



$$A[i][j] = \begin{cases} 0 & \text{if } (i,j) \notin E \\ w((i,j)) & \text{otherwise} \end{cases}$$

Adjacency list



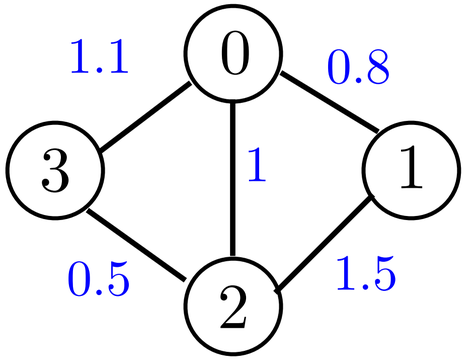
For each vertex we have a singly linked list of its out-adjacent vertices and assoc. weight.

Ordering is arbitrary.

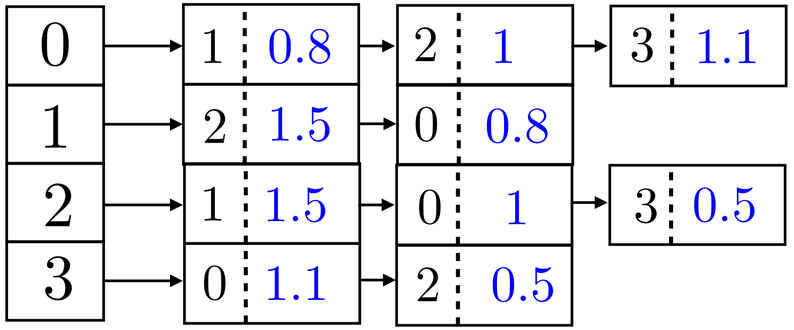
Undirected case

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 1 & 1.1 \\ 0.8 & 0 & 1.5 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0.5 & 0 \end{bmatrix}$$



Adjacency list



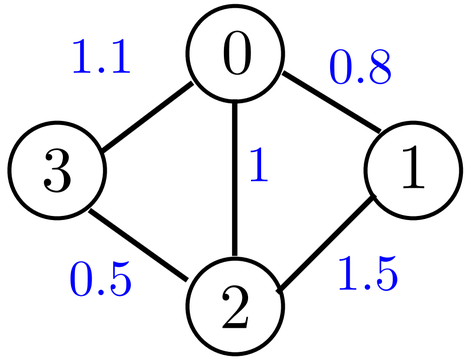
The matrix is symmetric.

The edge $\{i, j\}$ appears in the list for i and j.

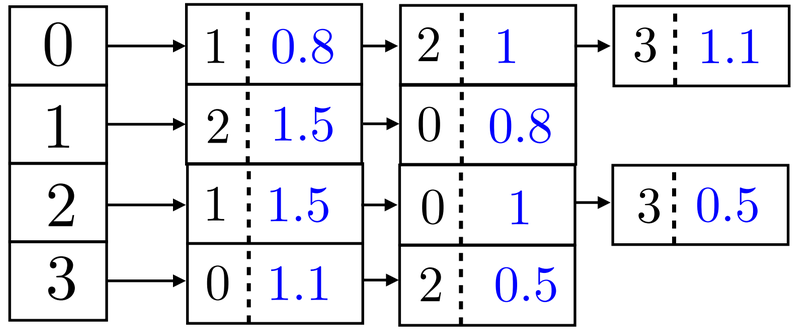
Undirected case

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 1 & 1.1 \\ 0.8 & 0 & 1.5 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0.5 & 0 \end{bmatrix}$$



Adjacency list



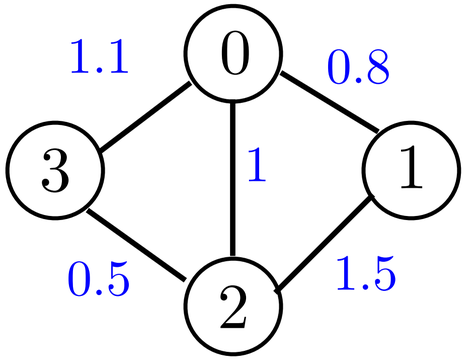
The matrix is symmetric.

The edge $\{i, j\}$ appears in the list for i and j.

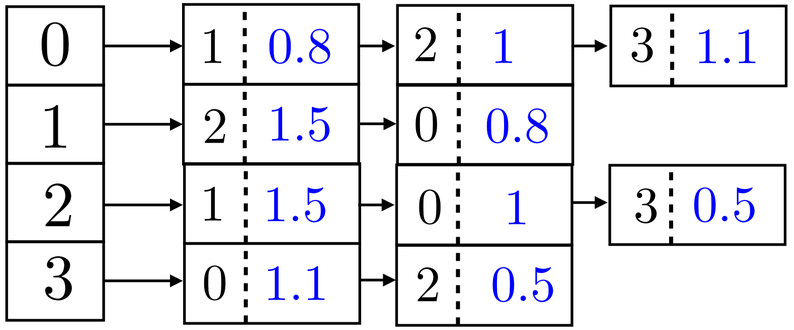
Undirected case

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 1 & 1.1 \\ 0.8 & 0 & 1.5 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0.5 & 0 \end{bmatrix}$$



Adjacency list



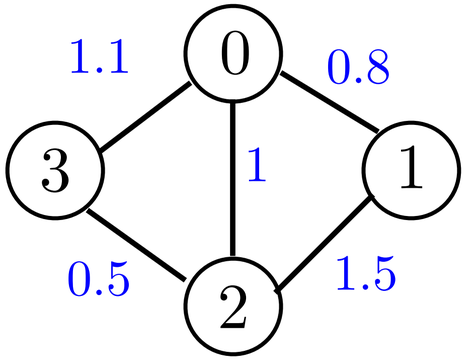
The matrix is symmetric.

The edge $\{i, j\}$ appears in the list for i and j.

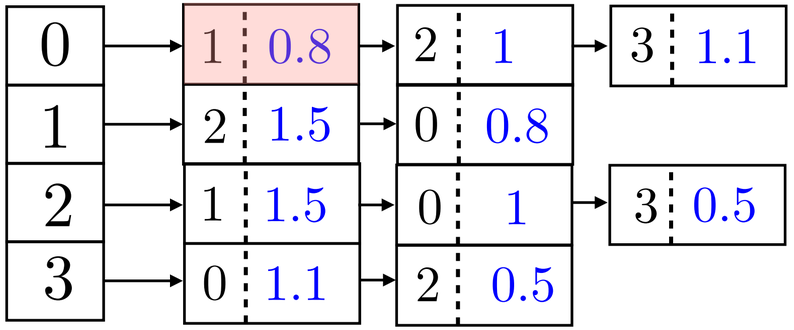
Undirected case

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 1 & 1.1 \\ 0.8 & 0 & 1.5 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0.5 & 0 \end{bmatrix}$$



Adjacency list



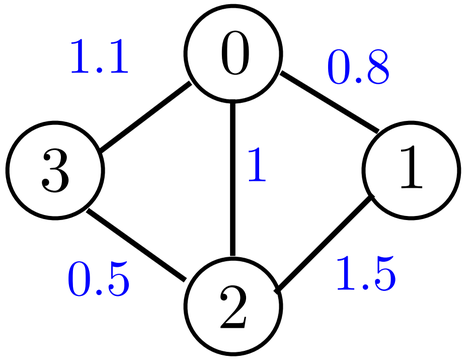
The matrix is symmetric.

The edge $\{i, j\}$ appears in the list for i and j.

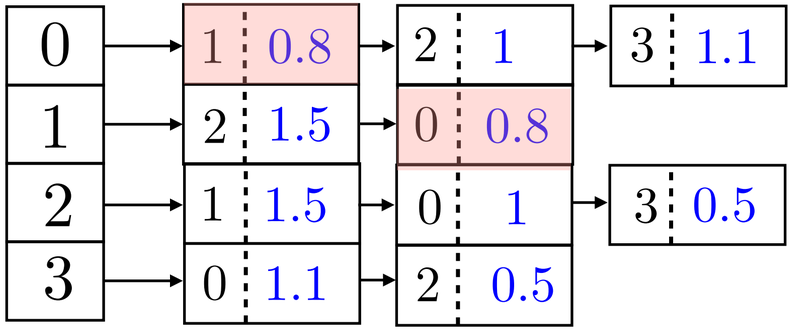
Undirected case

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 1 & 1.1 \\ 0.8 & 0 & 1.5 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0.5 & 0 \end{bmatrix}$$



Adjacency list



The matrix is symmetric.

The edge $\{i, j\}$ appears in the list for i and j.

Adjacency Matrix Implementation

```
template <unsigned N>
class Graph_adj_matrix
{
    bool matrix[N][N] {};
    bool is_directed = 0;

public:
    Graph_adj_matrix(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        matrix[i][j] = true;
        if(!is_directed)
        {
            matrix[j][i] = true;
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        return matrix[i][j];
    }
};
```

https://godbolt.org/z/5vv1zd4nh

Adjacency Matrix Implementation

```
template <unsigned N>
class Graph_adj_matrix
{
    bool matrix[N][N] {};
    bool is_directed = 0;

public:
    Graph_adj_matrix(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        matrix[i][j] = true;
        if(!is_directed)
        {
            matrix[j][i] = true;
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        return matrix[i][j];
    }
};
```

https://godbolt.org/z/5vv1zd4nh

Adjacency Matrix Implementation

```
template <unsigned N>
class Graph_adj_matrix
{
    bool matrix[N][N] {};
    bool is_directed = 0;

public:
    Graph_adj_matrix(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        matrix[i][j] = true;
        if(!is_directed)
        {
            matrix[j][i] = true;
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        return matrix[i][j];
    }
};
```

https://godbolt.org/z/5vv1zd4nh

Adjacency Matrix Implementation

```
template <unsigned N>
class Graph_adj_matrix
{
    bool matrix[N][N] {};
    bool is_directed = 0;

public:
    Graph_adj_matrix(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        matrix[i][j] = true;
        if(!is_directed)
        {
            matrix[j][i] = true;
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        return matrix[i][j];
    }
};
```

https://godbolt.org/z/5vv1zd4nh

Adjacency Matrix Implementation

```
template <unsigned N>
class Graph_adj_matrix
{
    bool matrix[N][N] {};
    bool is_directed = 0;

public:
    Graph_adj_matrix(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        matrix[i][j] = true;
        if(!is_directed)
        {
            matrix[j][i] = true;
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        return matrix[i][j];
    }
};
```

https://godbolt.org/z/5vv1zd4nh

Adjacency Matrix Implementation

```
template <unsigned N>
class Graph_adj_matrix
{
    bool matrix[N][N] {};
    bool is_directed = 0;

public:
    Graph_adj_matrix(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        matrix[i][j] = true;
        if(!is_directed)
        {
            matrix[j][i] = true;
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        return matrix[i][j];
    }
};
```

https://godbolt.org/z/5vv1zd4nh

Adjacency Matrix Implementation

```
template <unsigned N>
class Graph_adj_matrix
{
    bool matrix[N][N] {};
    bool is_directed = 0;

public:
    Graph_adj_matrix(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        matrix[i][j] = true;
        if(!is_directed)
        {
            matrix[j][i] = true;
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        return matrix[i][j];
    }
};
```

https://godbolt.org/z/5vv1zd4nh

Adjacency Matrix Implementation

```
template <unsigned N>
class Graph_adj_matrix
{
    bool matrix[N][N] {};
    bool is_directed = 0;

public:
    Graph_adj_matrix(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        matrix[i][j] = true;
        if(!is_directed)
        {
            matrix[j][i] = true;
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        return matrix[i][j];
    }
};
```

`add_edge` and `is_edge` both take time $O(1)$.

https://godbolt.org/z/5vv1zd4nh

Adjacency List Implementation

```
template <unsigned N>
class Graph_adj_list
{
    std::forward_list<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_list(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].push_front(j);
        if(!is_directed)
        {
            arr[j].push_front(i);
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        bool found = false;
        for(auto x: arr[i])
        {
            if(x == j)
            {
                found = true;
                break;
            }
        }
        return found;
    }
};
```

https://godbolt.org/z/x65s7vc91

Adjacency List Implementation

```
template <unsigned N>
class Graph_adj_list
{
    std::forward_list<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_list(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].push_front(j);
        if(!is_directed)
        {
            arr[j].push_front(i);
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        bool found = false;
        for(auto x: arr[i])
        {
            if(x == j)
            {
                found = true;
                break;
            }
        }
        return found;
    }
};
```

https://godbolt.org/z/x65s7vc91

Adjacency List Implementation

```
template <unsigned N>
class Graph_adj_list
{
    std::forward_list<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_list(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].push_front(j);
        if(!is_directed)
        {
            arr[j].push_front(i);
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        bool found = false;
        for(auto x: arr[i])
        {
            if(x == j)
            {
                found = true;
                break;
            }
        }
        return found;
    }
};
```

https://godbolt.org/z/x65s7vc91

Adjacency List Implementation

```
template <unsigned N>
class Graph_adj_list
{
    std::forward_list<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_list(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].push_front(j);
        if(!is_directed)
        {
            arr[j].push_front(i);
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        bool found = false;
        for(auto x: arr[i])
        {
            if(x == j)
            {
                found = true;
                break;
            }
        }
        return found;
    }
};
```

https://godbolt.org/z/x65s7vc91

Adjacency List Implementation

```
template <unsigned N>
class Graph_adj_list
{
    std::forward_list<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_list(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].push_front(j);
        if(!is_directed)
        {
            arr[j].push_front(i);
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        bool found = false;
        for(auto x: arr[i])
        {
            if(x == j)
            {
                found = true;
                break;
            }
        }
        return found;
    }
};
```

https://godbolt.org/z/x65s7vc91

Adjacency List Implementation

```
template <unsigned N>
class Graph_adj_list
{
    std::forward_list<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_list(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].push_front(j);
        if(!is_directed)
        {
            arr[j].push_front(i);
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        bool found = false;
        for(auto x: arr[i])
        {
            if(x == j)
            {
                found = true;
                break;
            }
        }
        return found;
    }
};
```

https://godbolt.org/z/x65s7vc91

Adjacency List Implementation

```
template <unsigned N>
class Graph_adj_list
{
    std::forward_list<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_list(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].push_front(j);
        if(!is_directed)
        {
            arr[j].push_front(i);
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        bool found = false;
        for(auto x: arr[i])
        {
            if(x == j)
            {
                found = true;
                break;
            }
        }
        return found;
    }
};
```

add_edge$(i, j) : O(1)$

https://godbolt.org/z/x65s7vc91

Adjacency List Implementation

```
template <unsigned N>
class Graph_adj_list
{
    std::forward_list<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_list(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].push_front(j);
        if(!is_directed)
        {
            arr[j].push_front(i);
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        bool found = false;
        for(auto x: arr[i])
        {
            if(x == j)
            {
                found = true;
                break;
            }
        }
        return found;
    }
};
```

add_edge$(i, j) : O(1)$

https://godbolt.org/z/x65s7vc91

Adjacency List Implementation

```
template <unsigned N>
class Graph_adj_list
{
    std::forward_list<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_list(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].push_front(j);
        if(!is_directed)
        {
            arr[j].push_front(i);
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        bool found = false;
        for(auto x: arr[i])
        {
            if(x == j)
            {
                found = true;
                break;
            }
        }
        return found;
    }
};
```

add_edge$(i, j) : O(1)$

https://godbolt.org/z/x65s7vc91

Adjacency List Implementation

```
template <unsigned N>
class Graph_adj_list
{
    std::forward_list<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_list(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].push_front(j);
        if(!is_directed)
        {
            arr[j].push_front(i);
        }
    }

    bool is_edge(unsigned i, unsigned j)
    {
        bool found = false;
        for(auto x: arr[i])
        {
            if(x == j)
            {
                found = true;
                break;
            }
        }
        return found;
    }
};
```

$\texttt{add\_edge}(i, j) : O(1)$

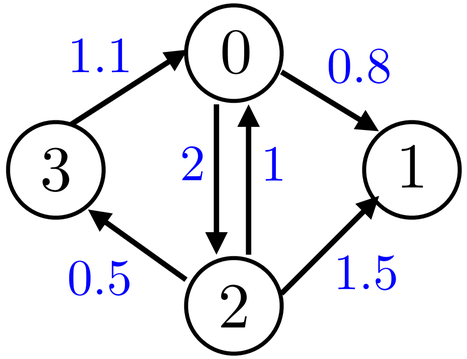
$\texttt{is\_edge}(i, j) : \Theta(\deg(i))$

https://godbolt.org/z/x65s7vc91

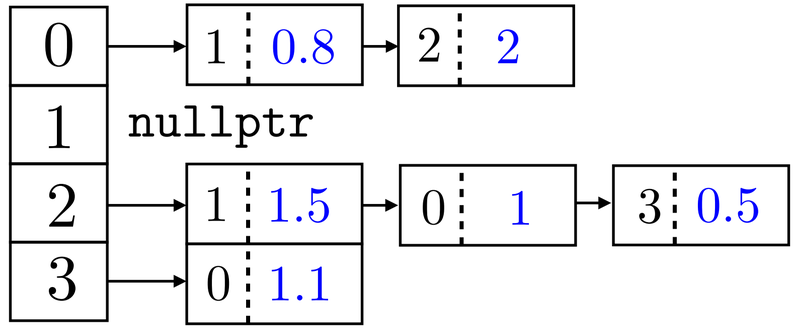
Comparison

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0 & 0 \end{bmatrix}$$



Adjacency list



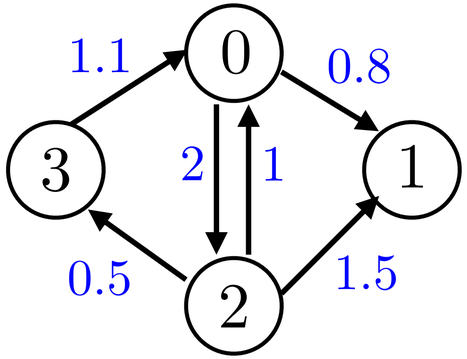
	Adj. matrix	Adj. list
size	$\Theta(n^2)$	$\Theta(n+m)$
add edge	$\Theta(1)$	$\Theta(1)$
edge (i,j) ?	$\Theta(1)$	$\Theta(\deg(i))$
list out-adjacent to v	$\Theta(n)$	$\Theta(\deg(v))$

Here m is the number of edges.

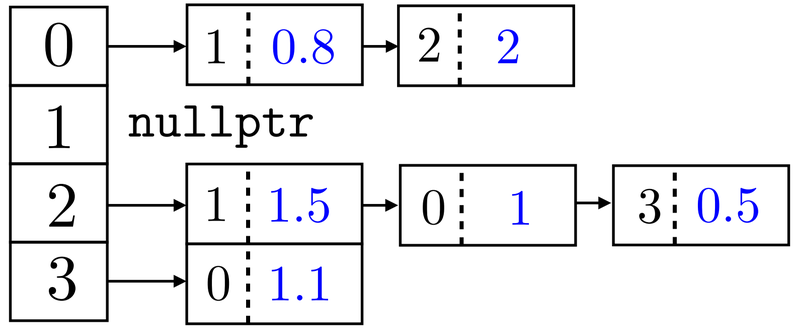
Comparison

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0 & 0 \end{bmatrix}$$



Adjacency list



	Adj. matrix	Adj. list
size	$\Theta(n^2)$	$\Theta(n+m)$
add edge	$\Theta(1)$	$\Theta(1)$
edge (i,j) ?	$\Theta(1)$	$\Theta(\deg(i))$
list out-adjacent to v	$\Theta(n)$	$\Theta(\deg(v))$

Here m is the number of edges.

We will use the adjacency list model for all our algorithms.

Comparison

An adjacency matrix of an n vertex graph always takes memory $\Theta(n^2)$, no matter how many edges the graph has.

We typically only use an adjacency matrix when n is small or when the graph is dense, i.e. the number of edges is $\Omega(n^2)$.

Positive features of an adjacency matrix:

- Contiguous memory layout.
- Constant time to check for the presence/absence of a particular edge.

Be Flexible

There are a lot of design choices here. What features does your application need?

	Adj. matrix	Adj. list
size	$\Theta(n^2)$	$\Theta(n+m)$
add edge	$\Theta(1)$	$\Theta(1)$
edge (i,j) ?	$\Theta(1)$	$\Theta(\deg(i))$
list out-adjacent to v	$\Theta(n)$	$\Theta(\deg(v))$

Be Flexible

There are a lot of design choices here. What features does your application need?

	Adj. matrix	Adj. list
size	$\Theta(n^2)$	$\Theta(n+m)$
add edge	$\Theta(1)$	$\Theta(1)$
edge (i,j) ?	$\Theta(1)$	$\Theta(\deg(i))$
list out-adjacent to v	$\Theta(n)$	$\Theta(\deg(v))$

What if we took an array of `std::set` instead of linked lists?

"Adjacency Set"

What if we took an array of `std :: set` instead of linked lists?

```
template <unsigned N>
class Graph_adj_set
{
    std::set<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_set(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].insert(j);
        if(!is_directed)
        {
            arr[j].insert(i);
        }
    }
    bool is_edge(unsigned i, unsigned j)
    {
        return arr[i].contains(j);
    }
};
```

https://godbolt.org/z/e9M9W7f9o

"Adjacency Set"

What if we took an array of `std :: set` instead of linked lists?

```
template <unsigned N>
class Graph_adj_set
{
    std::set<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_set(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].insert(j);
        if(!is_directed)
        {
            arr[j].insert(i);
        }
    }
    bool is_edge(unsigned i, unsigned j)
    {
        return arr[i].contains(j);
    }
};
```

https://godbolt.org/z/e9M9W7f9o

"Adjacency Set"

What if we took an array of `std :: set` instead of linked lists?

```
template <unsigned N>
class Graph_adj_set
{
    std::set<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_set(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].insert(j);
        if(!is_directed)
        {
            arr[j].insert(i);
        }
    }
    bool is_edge(unsigned i, unsigned j)
    {
        return arr[i].contains(j);
    }
};
```

https://godbolt.org/z/e9M9W7f9o

"Adjacency Set"

What if we took an array of `std :: set` instead of linked lists?

```
template <unsigned N>
class Graph_adj_set
{
    std::set<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_set(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].insert(j);
        if(!is_directed)
        {
            arr[j].insert(i);
        }
    }
    bool is_edge(unsigned i, unsigned j)
    {
        return arr[i].contains(j);
    }
};
```

$\text{add\_edge}(i, j) : O(\log(n))$

https://godbolt.org/z/e9M9W7f9o

"Adjacency Set"

What if we took an array of `std :: set` instead of linked lists?

```
template <unsigned N>
class Graph_adj_set
{
    std::set<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_set(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].insert(j);
        if(!is_directed)
        {
            arr[j].insert(i);
        }
    }
    bool is_edge(unsigned i, unsigned j)
    {
        return arr[i].contains(j);
    }
};
```

$\text{add\_edge}(i, j) : O(\log(n))$

https://godbolt.org/z/e9M9W7f9o

"Adjacency Set"

What if we took an array of `std :: set` instead of linked lists?

```
template <unsigned N>
class Graph_adj_set
{
    std::set<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_set(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].insert(j);
        if(!is_directed)
        {
            arr[j].insert(i);
        }
    }
    bool is_edge(unsigned i, unsigned j)
    {
        return arr[i].contains(j);
    }
};
```

$\texttt{add\_edge}(i, j) : O(\log(n))$

https://godbolt.org/z/e9M9W7f9o

"Adjacency Set"

What if we took an array of `std :: set` instead of linked lists?

```
template <unsigned N>
class Graph_adj_set
{
    std::set<unsigned> arr[N] {};
    bool is_directed = 0;
public:
    Graph_adj_set(bool type): is_directed(type) {}

    void add_edge(unsigned i, unsigned j)
    {
        arr[i].insert(j);
        if(!is_directed)
        {
            arr[j].insert(i);
        }
    }
    bool is_edge(unsigned i, unsigned j)
    {
        return arr[i].contains(j);
    }
};
```

$\texttt{add\_edge}(i, j) : O(\log(n))$

$\texttt{is\_edge}(i, j) : O(\log \deg(i))$

https://godbolt.org/z/e9M9W7f9o

Be Flexible

	Adj. matrix	Adj. list	Adj. set
size	$\Theta(n^2)$	$\Theta(n+m)$	$\Theta(n+m)$
add edge	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$
edge (i,j) ?	$\Theta(1)$	$\Theta(\deg(i))$	$\Theta(\log \deg(i))$
list out-adjacent to v	$\Theta(n)$	$\Theta(\deg(v))$	$\Theta(\deg(v))$

With an "adjacency set" we have memory proportional to the number of edges but also pretty fast times for adding and checking for an edge.

You could use `std::unordered_set` instead to get these down to average case amoritised constant time.

Why use Adjacency List?

Considering an "Adjacency Set" begs the question: why use an Adjacency List at all?

Why use Adjacency List?

Considering an "Adjacency Set" begs the question: why use an Adjacency List at all?

Part of the reason is no doubt historical.

Why use Adjacency List?

Considering an "Adjacency Set" begs the question: why use an Adjacency List at all?

Part of the reason is no doubt historical.

But it is also interesting to state our algorithms in the weakest model needed.

This can allow a greater range of application.

All our algorithms work in the adjacency list model!

Rubik's cube graph

What representation shall we use for the Rubik's cube graph?

We don't want to explicitly store this graph at all!

But we can simulate an adjacency list representation.

In particular, we can

Iterate through all one-move neighbors of a cube position.

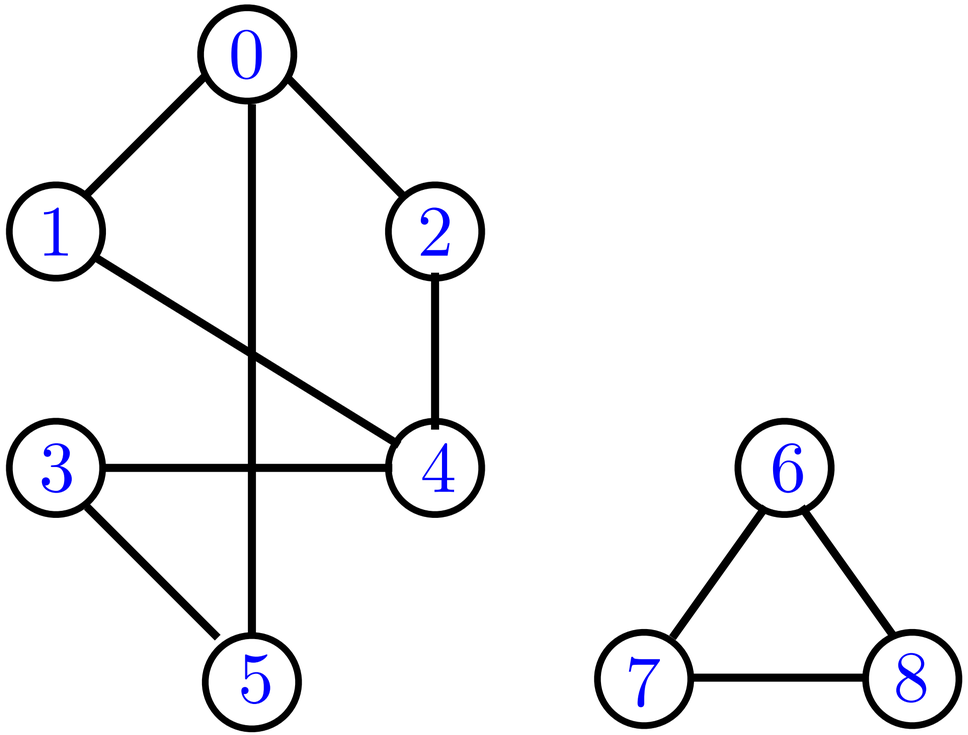
Remarkably, the algorithms we discuss only need this primitive.

Depth/Breadth First Search

Connectivity

Some of the most basic questions about a graph are about connectivity.

For this segment, we will just talk about undirected graphs.



Is the graph connected?

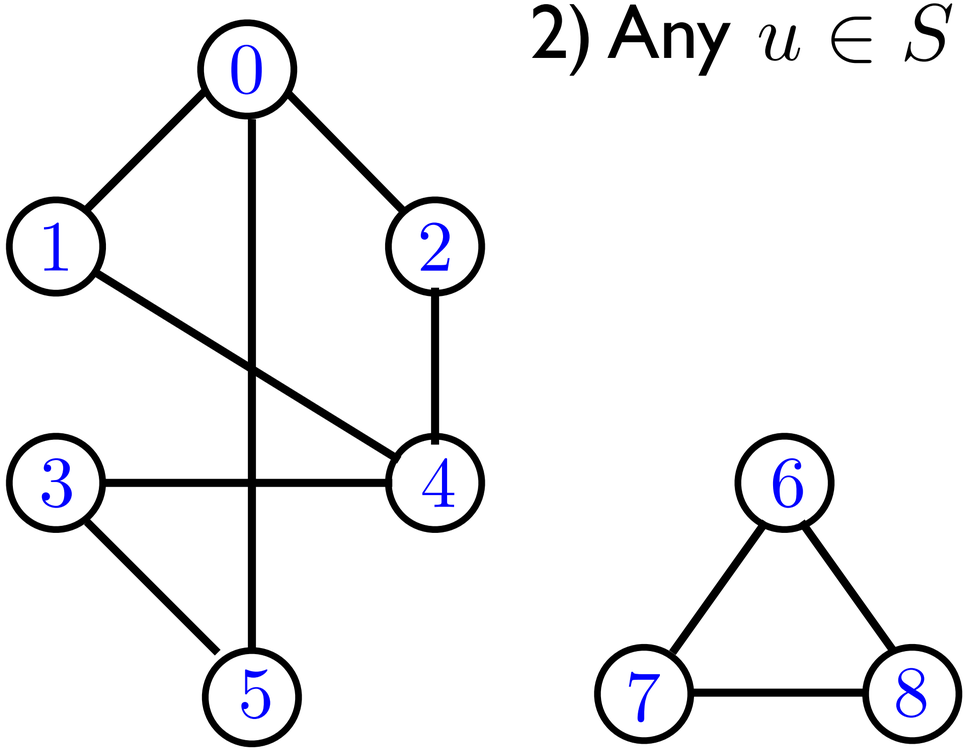
What are the vertices that can be reached from vertex 0?

Connected Component

A connected component of a graph is a subset S of vertices that

1) is connected, i.e. there is a path between every $u, v \in S$.

2) Any $u \in S$ is not connected to any $v \notin S$.

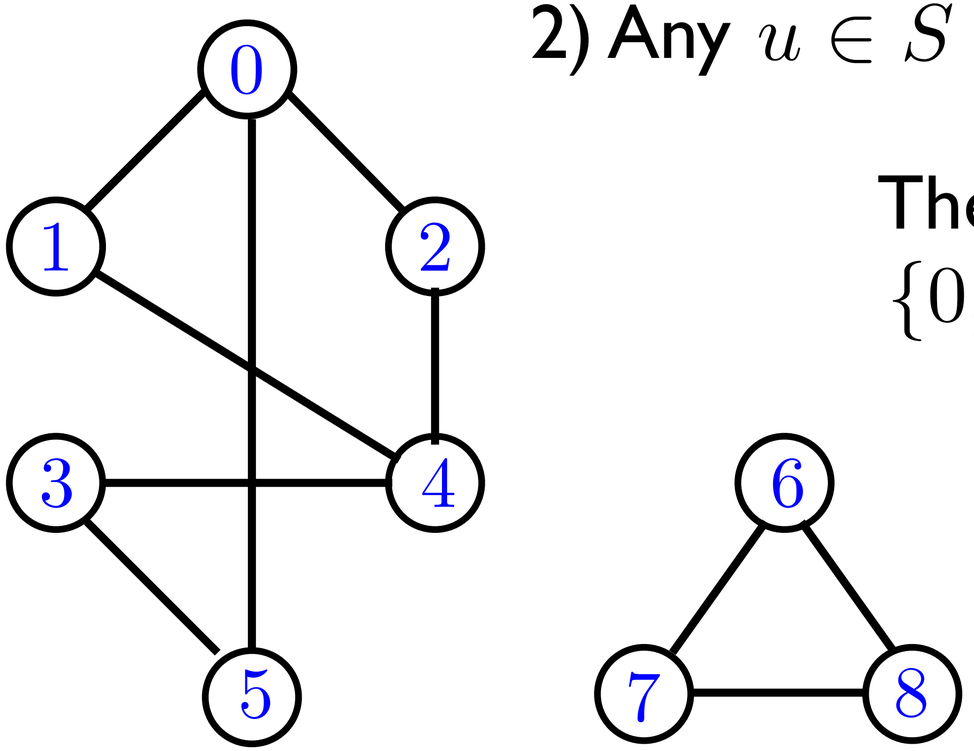


Connected Component

A connected component of a graph is a subset S of vertices that

1) is connected, i.e. there is a path between every $u, v \in S$.

2) Any $u \in S$ is not connected to any $v \notin S$.



The connected components in this graph are $\{0, 1, 2, 3, 4, 5\}$ and $\{6, 7, 8\}$.

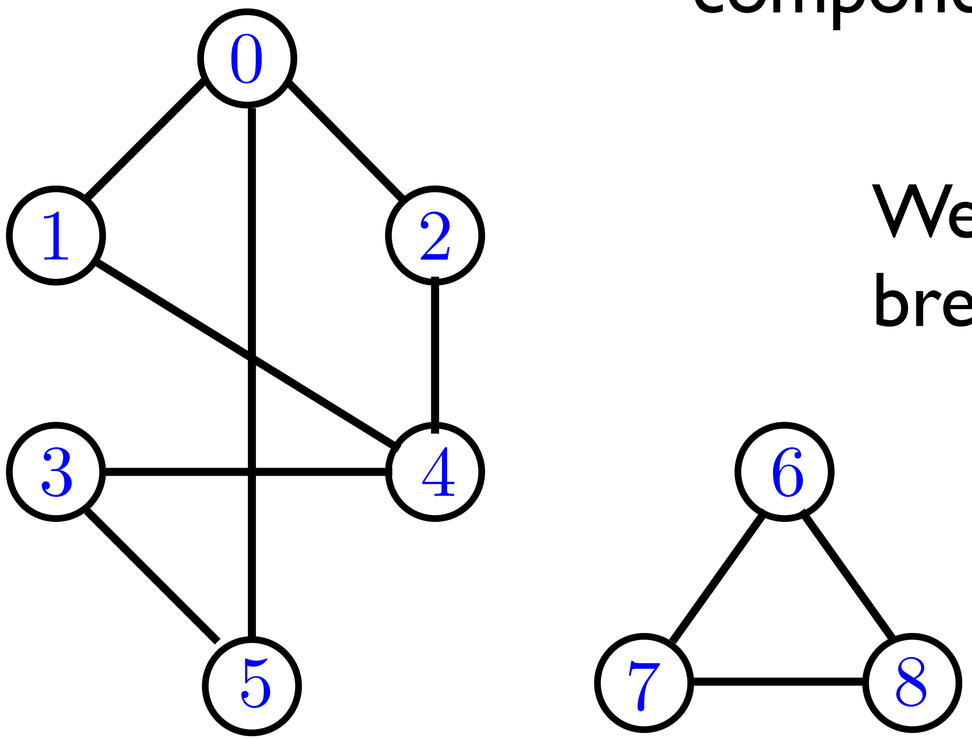
The connected components partition the vertices of the graph.

Connected Component

When processing a graph, usually we work on each connected component separately.

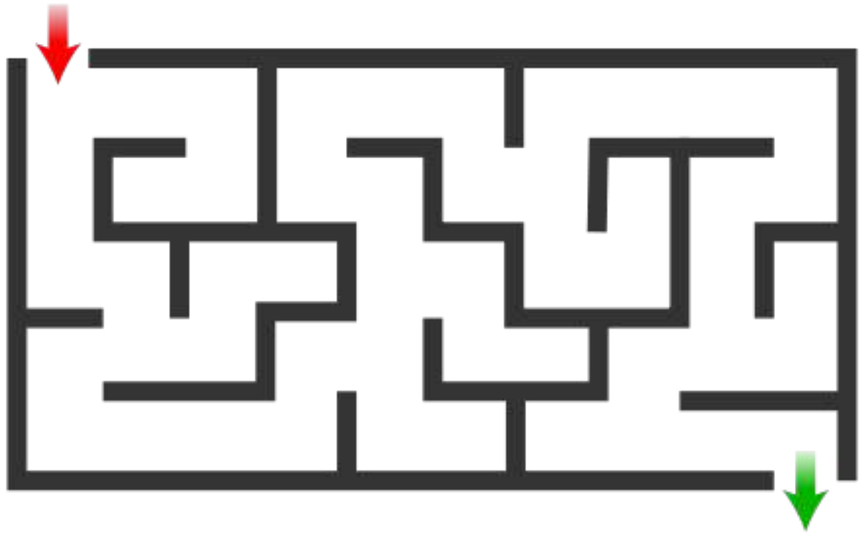
A natural first step is to find all the connected components.

We can do this (and more) with depth- or breadth-first search.

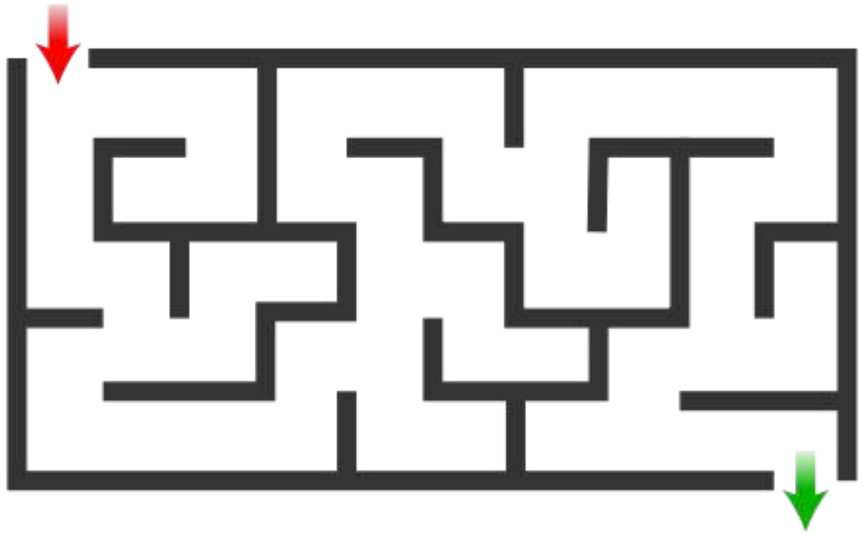


Depth First Search

Depth-first search



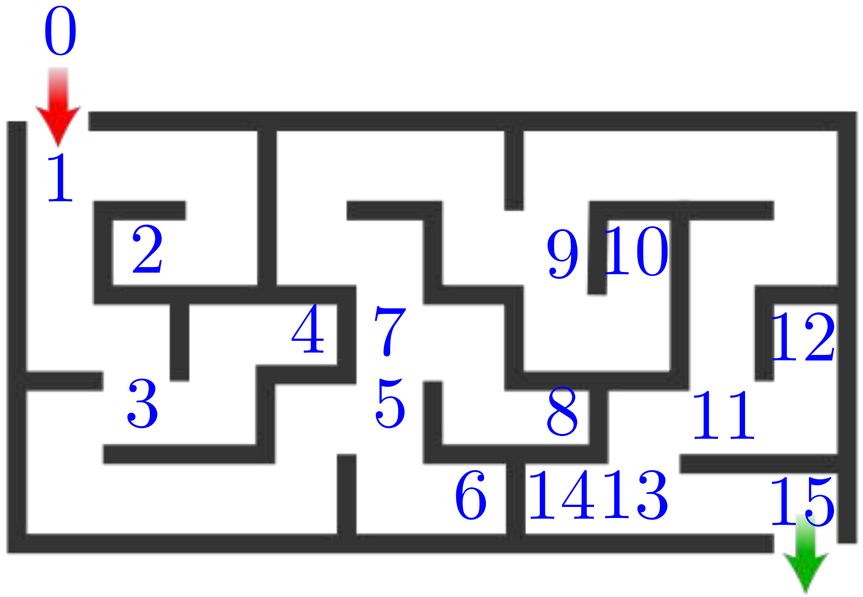
Depth-first search



We can think of a maze as a graph.

The vertices are the branch points in the maze, and the edges are corridors of the maze.

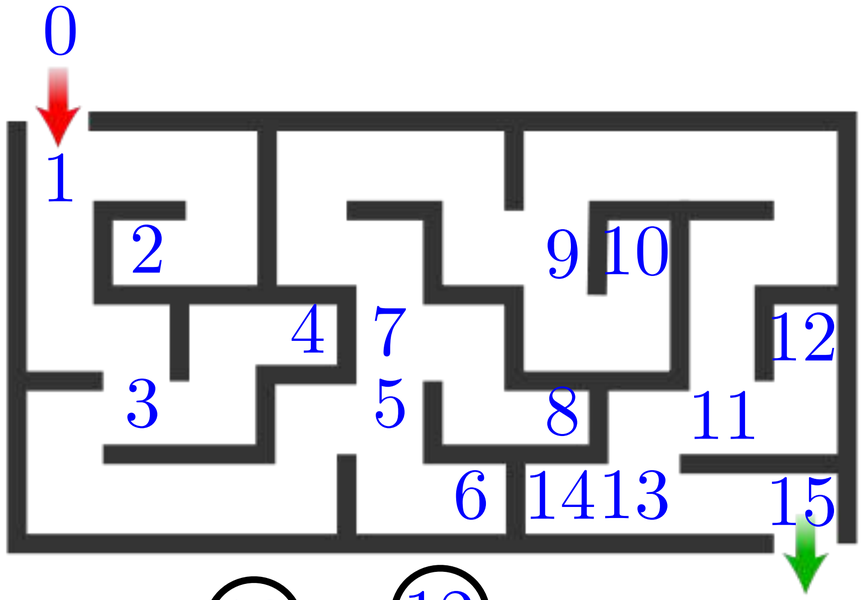
Depth-first search

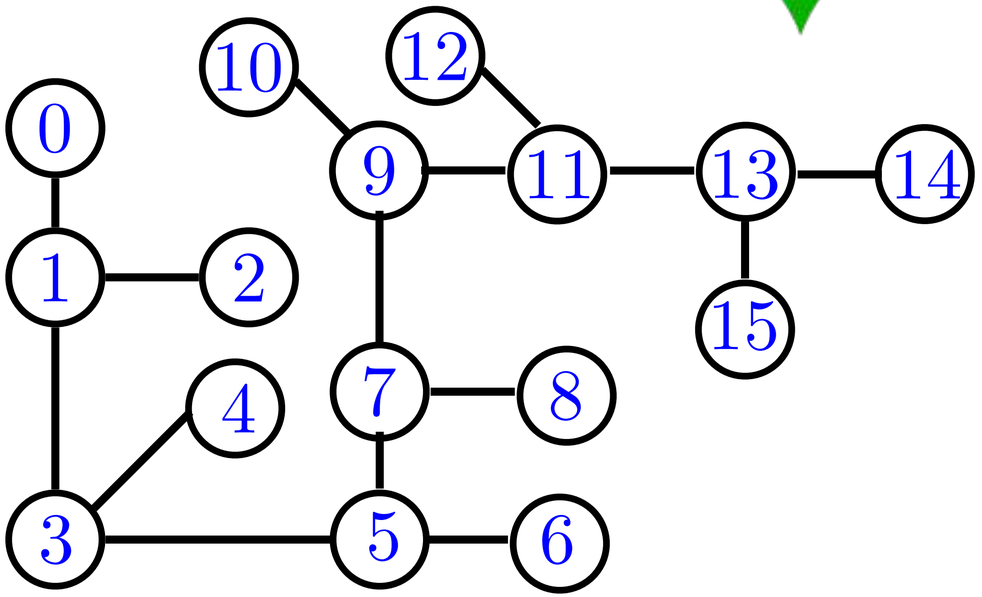


We can think of a maze as a graph.

The vertices are the branch points in the maze, and the edges are corridors of the maze.

Depth-first search

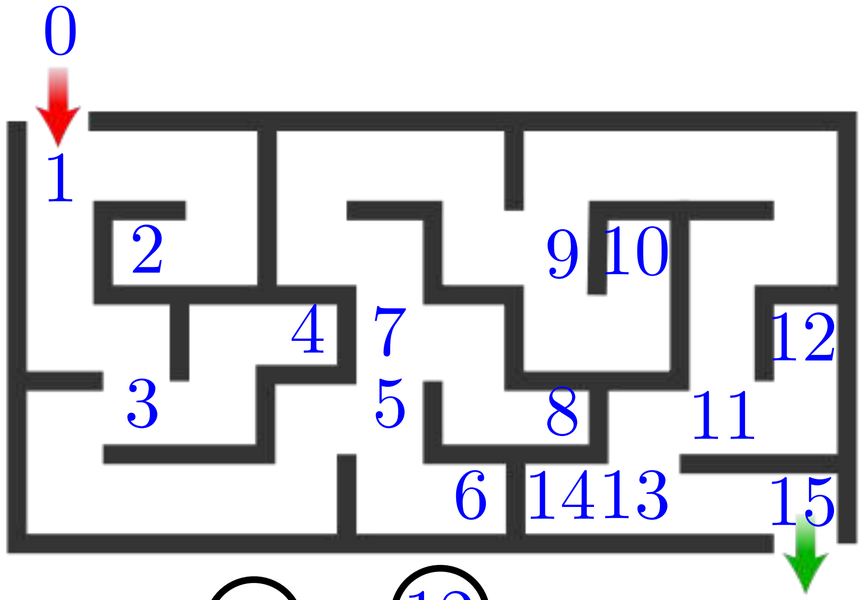


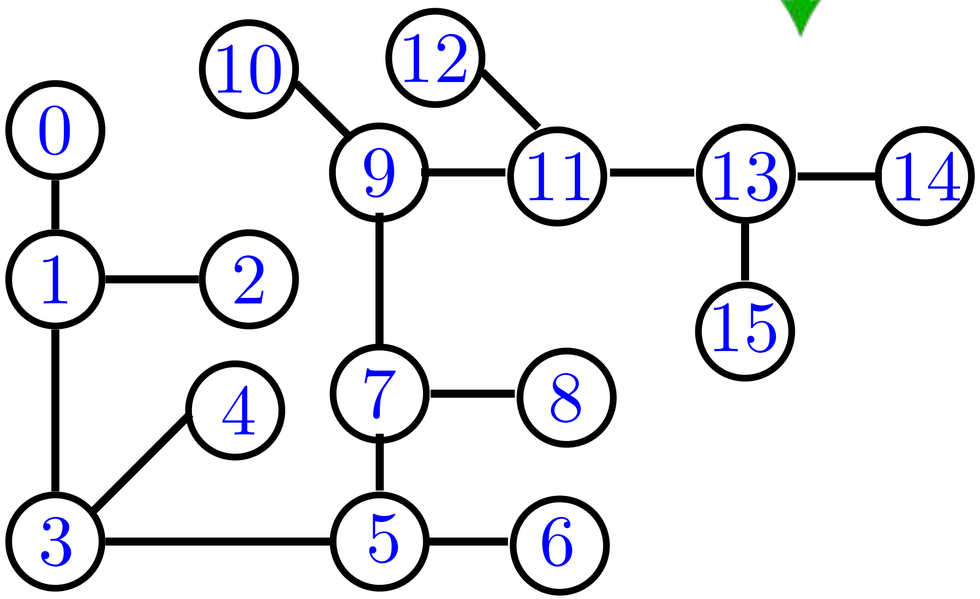


We can think of a maze as a graph.

The vertices are the branch points in the maze, and the edges are corridors of the maze.

Depth-first search





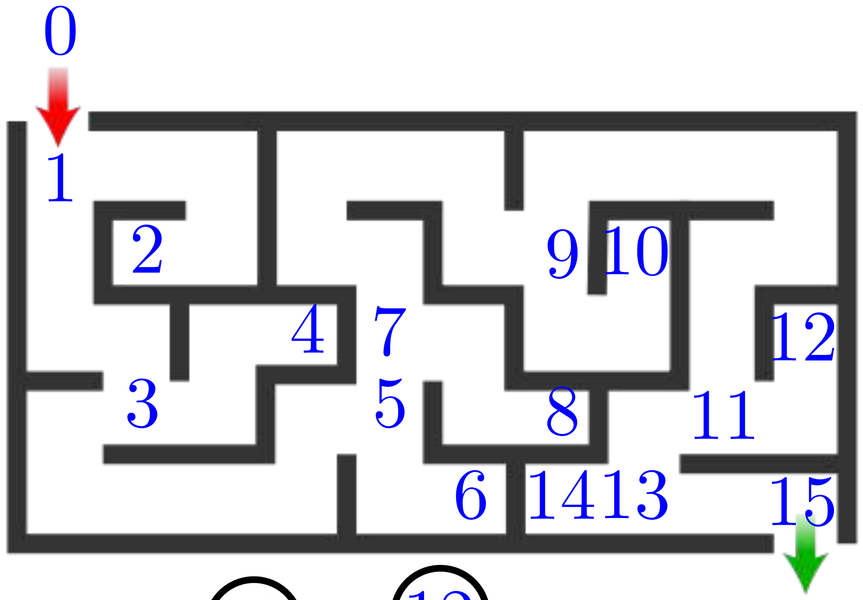
We can think of a maze as a graph.

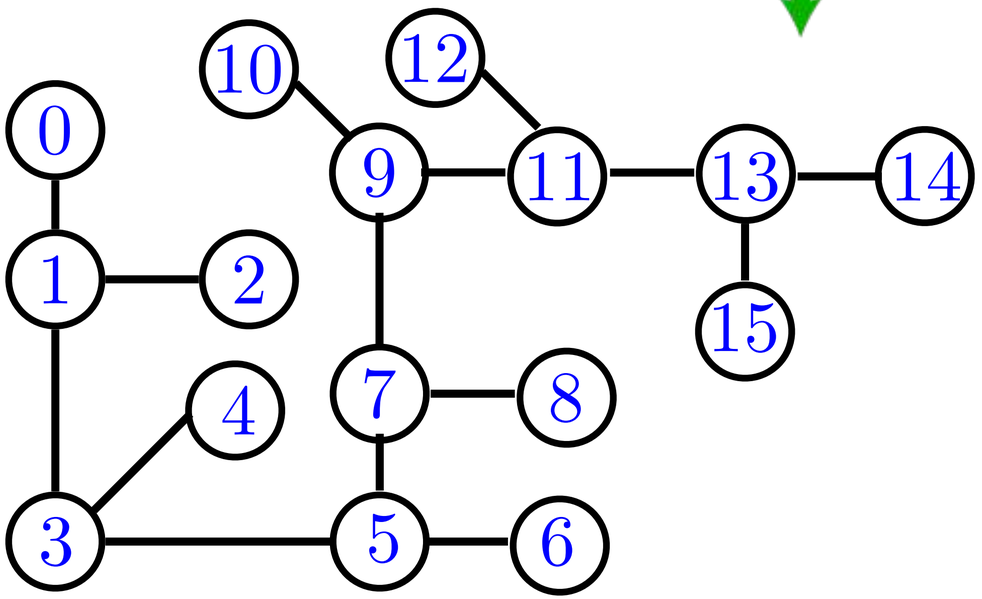
The vertices are the branch points in the maze, and the edges are corridors of the maze.

Determining if there is a way out of the maze is a connectivity problem.

Of course we also want to know the path out of the maze!

Depth-first search





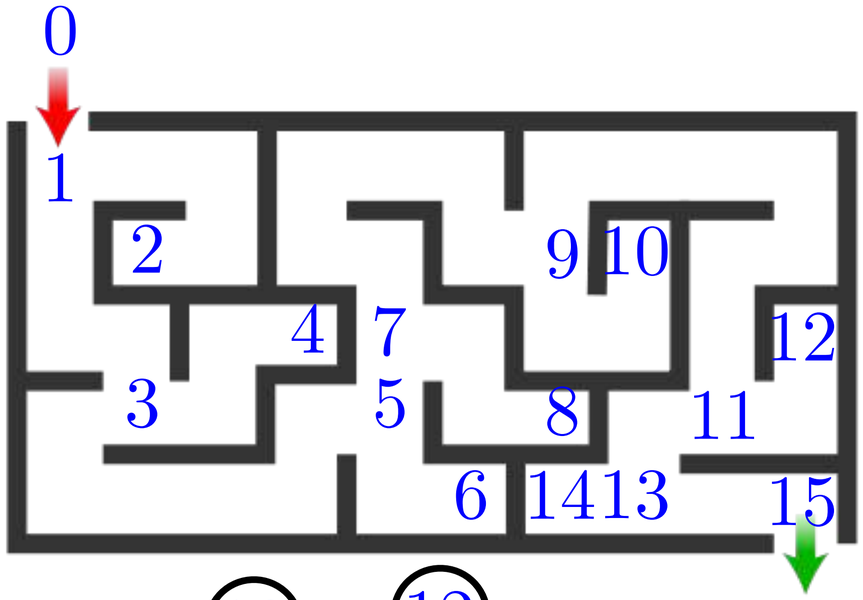
Depth-first search is similar to trying to find your way out of a maze with a ball of string and a piece of chalk.

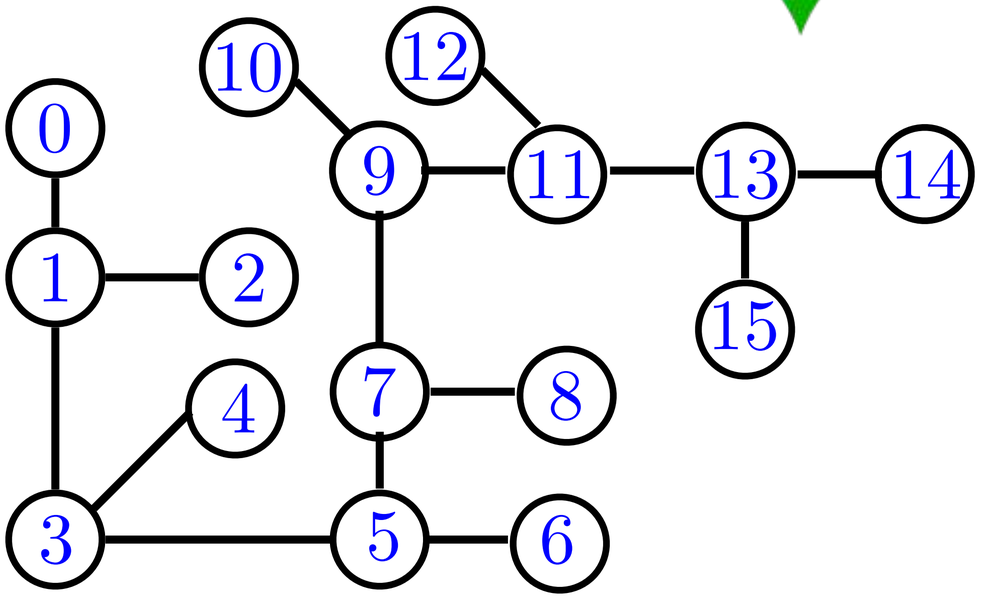
You tether your string to the start of the maze.

You mark a corridor with the chalk when you take it.

When you come to a branch point you follow a corridor not already marked.

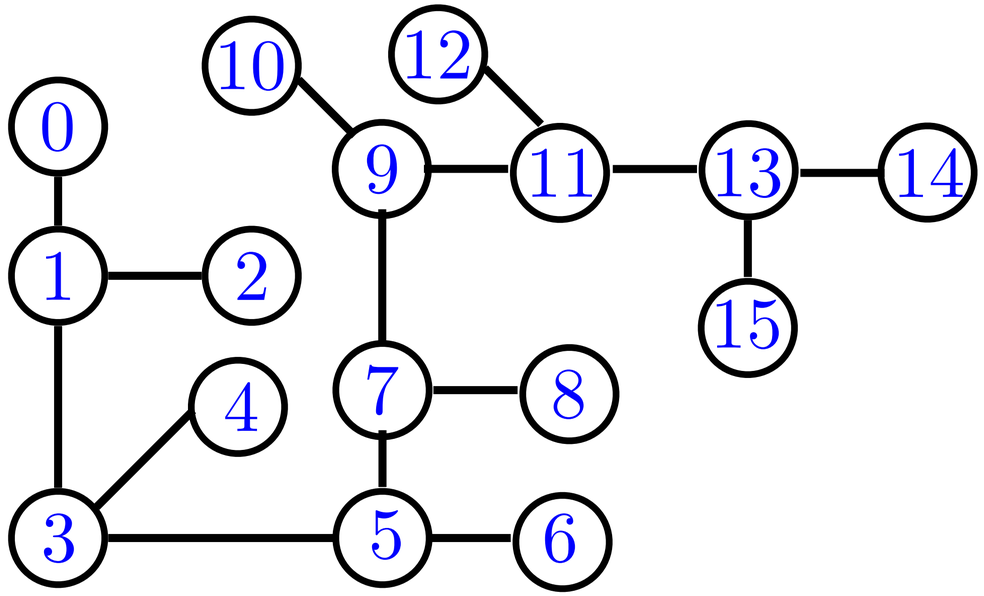
Depth-first search





When all corridors at a branch point have been visited, use the string to go back to the previous branch point.

Depth-first search



We are using the adj. list representation.

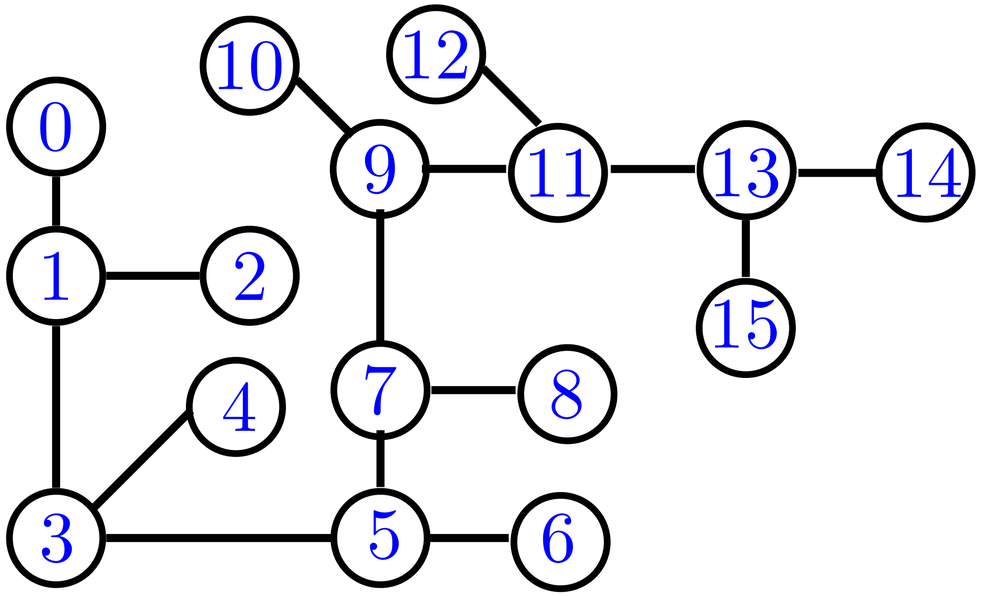
In the for loop we iterate through the neighbors of the vertex we are visiting.

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

https://godbolt.org/z/TzrjaYs3G

Depth-first search



We are using the adj. list representation.

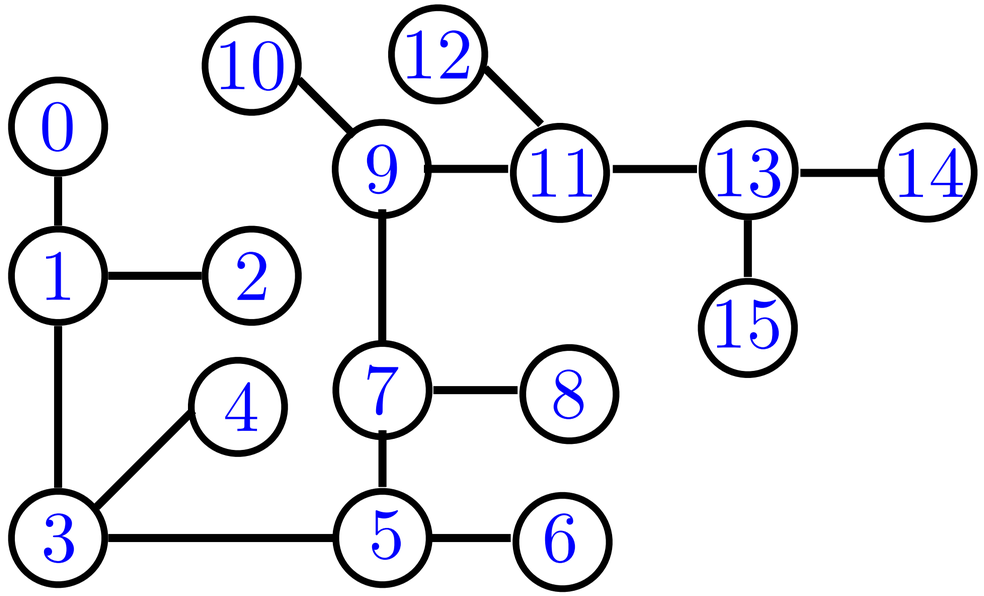
In the for loop we iterate through the neighbors of the vertex we are visiting.

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

https://godbolt.org/z/TzrjaYs3G

Depth-first search



We are using the adj. list representation.

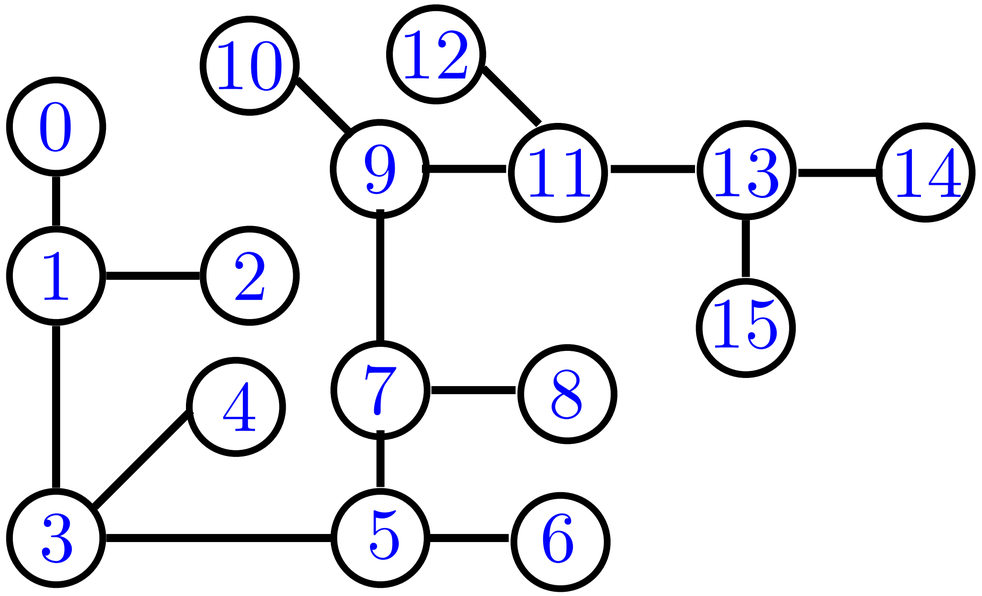
In the for loop we iterate through the neighbors of the vertex we are visiting.

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

https://godbolt.org/z/TzrjaYs3G

Depth-first search



We are using the adj. list representation.

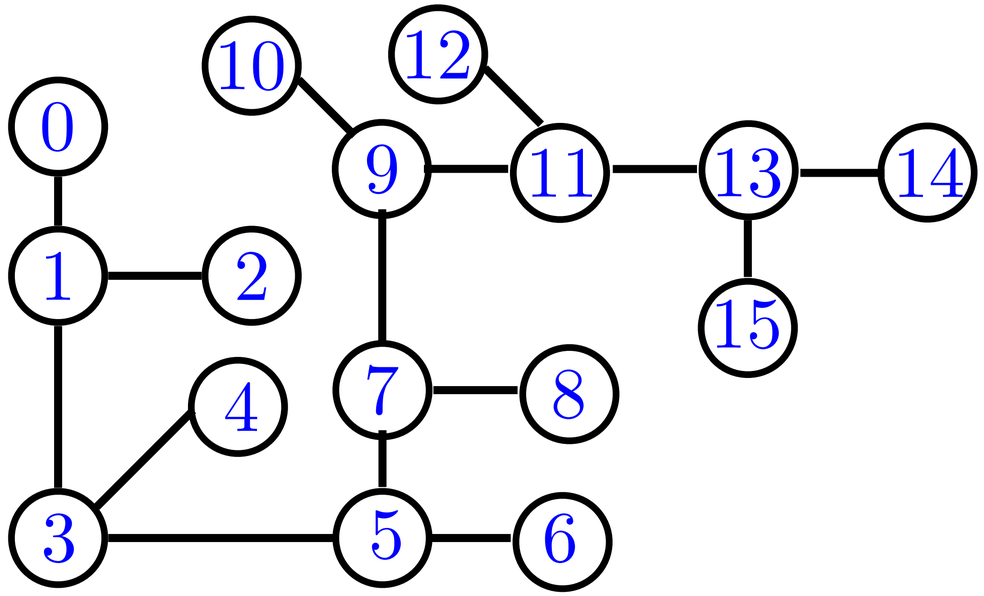
In the for loop we iterate through the neighbors of the vertex we are visiting.

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

https://godbolt.org/z/TzrjaYs3G

Depth-first search



We are using the adj. list representation.

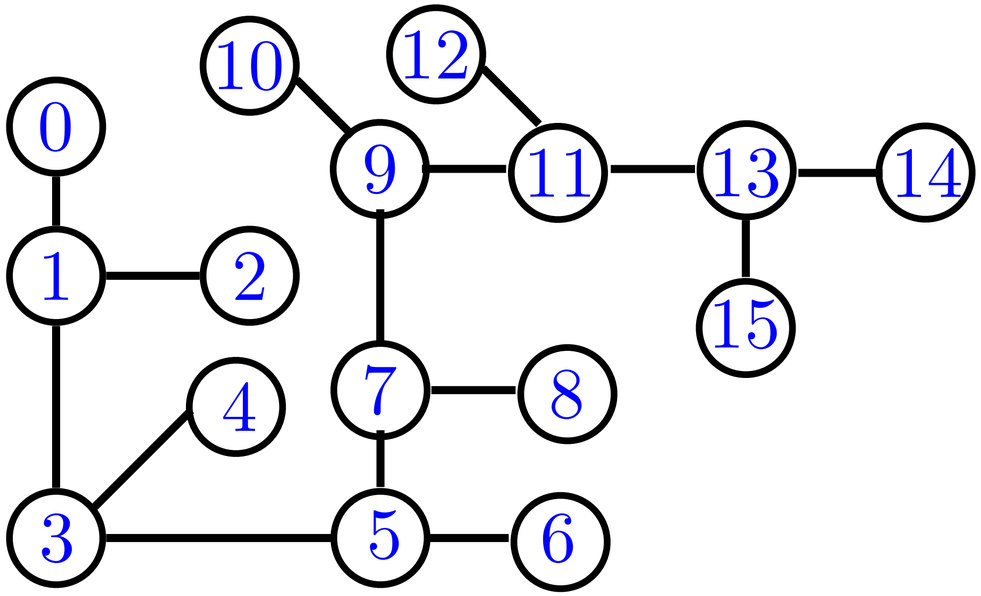
In the for loop we iterate through the neighbors of the vertex we are visiting.

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

https://godbolt.org/z/TzrjaYs3G

Depth-first search



We are using the adj. list representation.

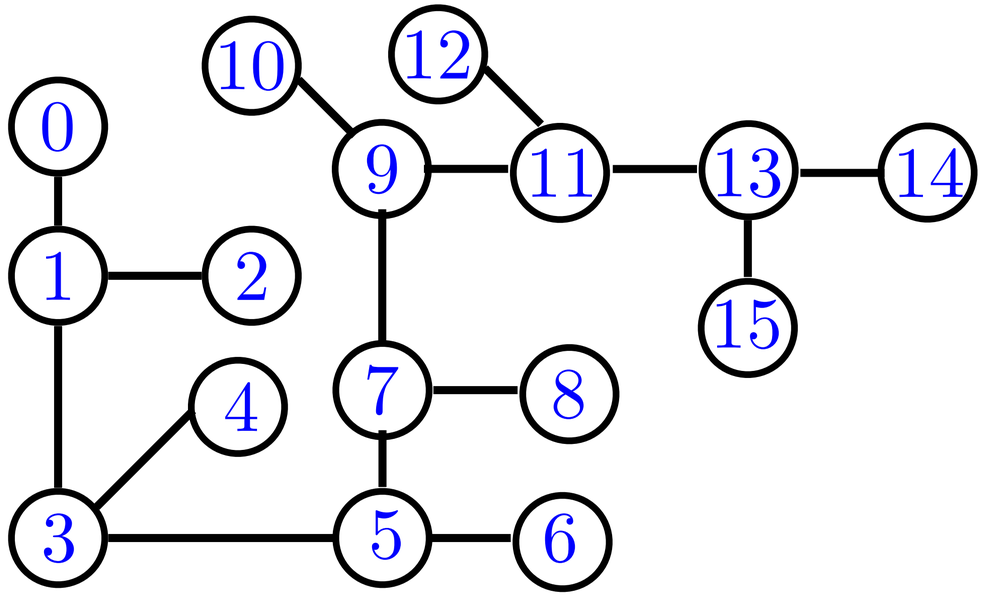
In the for loop we iterate through the neighbors of the vertex we are visiting.

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

https://godbolt.org/z/TzrjaYs3G

Depth-first search

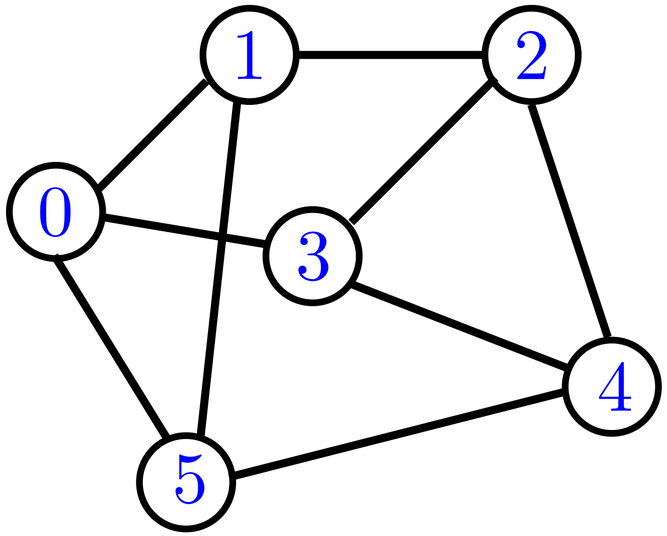


The search route of DFS depends on the order of vertices in the adj. lists.

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

Depth-first search



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: F
1: F
2: F
3: F
4: F
5: F

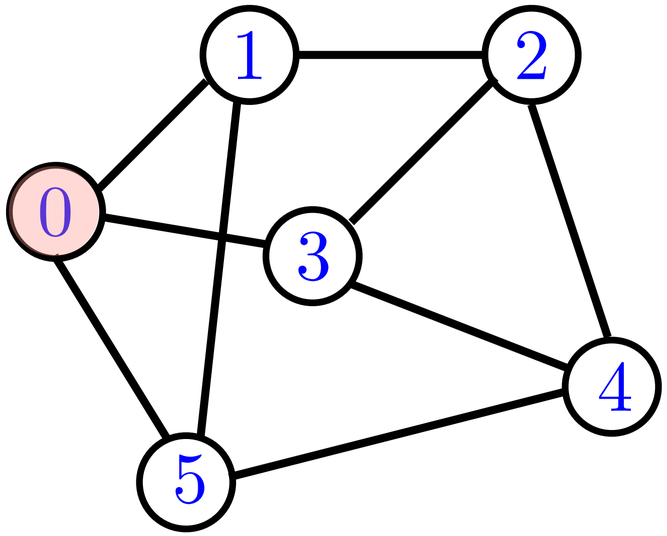
```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

Let's look at a trace of `dfs(0)`.

https://godbolt.org/z/eEjTM6xYd

dfs(0)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

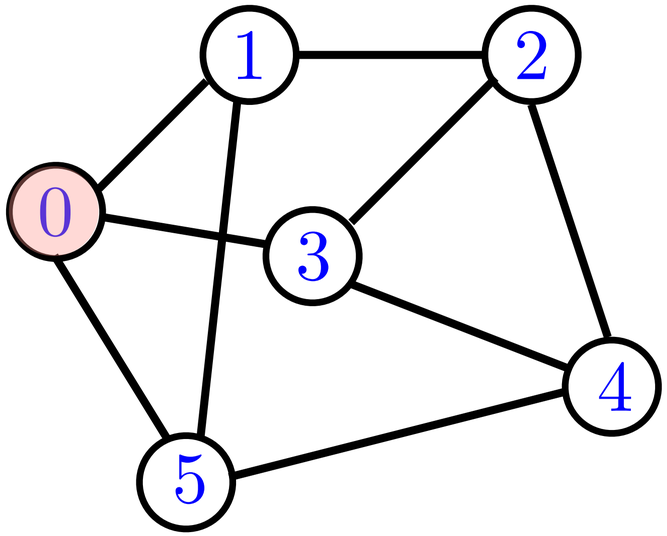
0: T
1: F
2: F
3: F
4: F
5: F

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We first mark vertex 0.

dfs(0)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

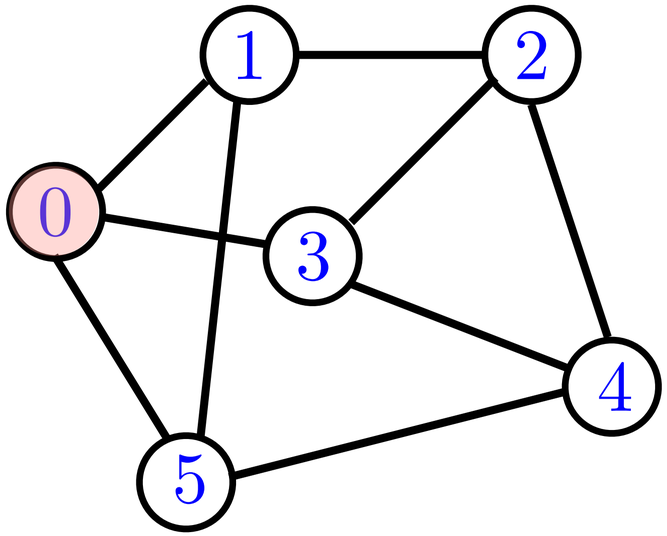
0: T
1: F
2: F
3: F
4: F
5: F

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We first mark vertex 0.

dfs(0)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

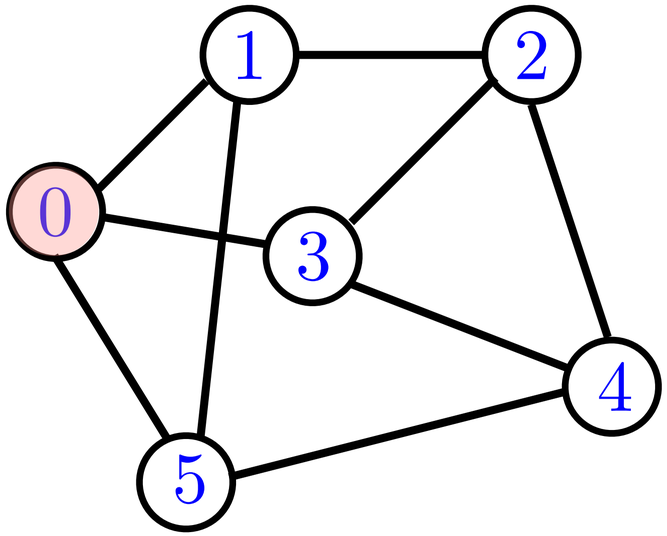
0: T
1: F
2: F
3: F
4: F
5: F

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We first mark vertex 0.

dfs(0)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

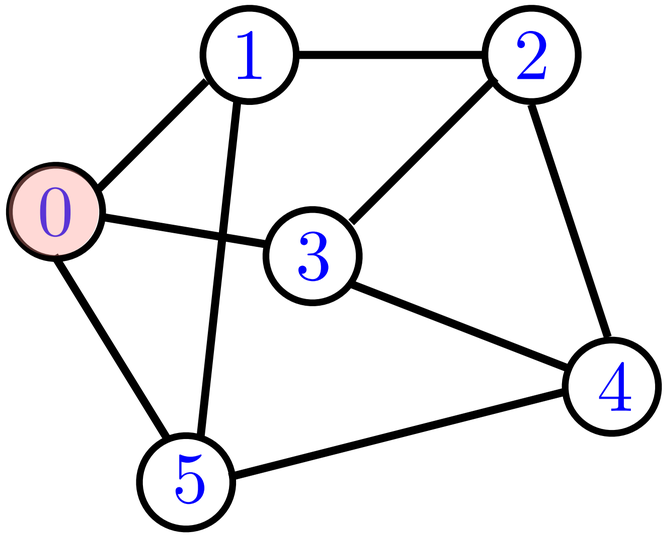
0: T
1: F
2: F
3: F
4: F
5: F

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We first mark vertex 0.

dfs(0)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

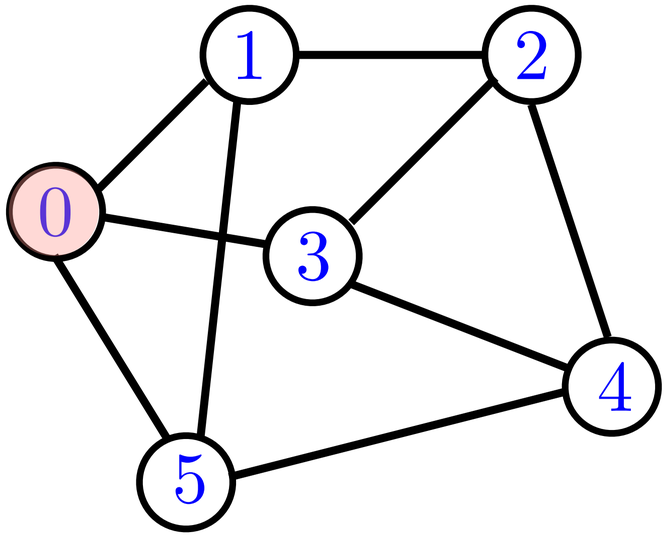
0: T
1: F
2: F
3: F
4: F
5: F

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

Vertex 1 is not marked, so now we move to call `dfs(1)`.

dfs(0)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

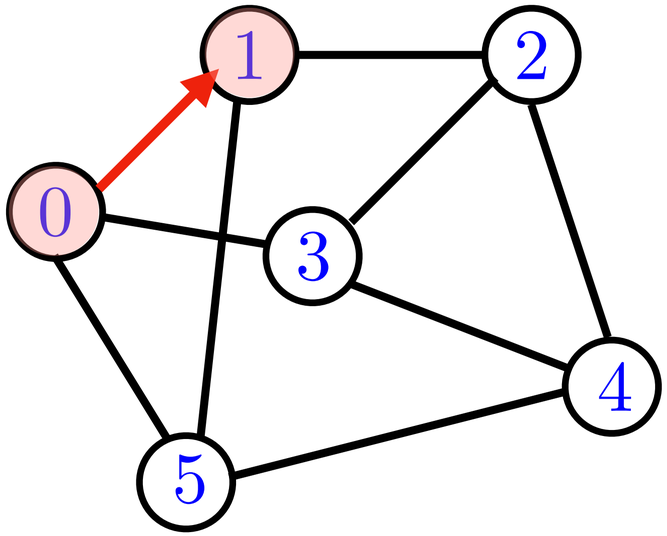
0: T
1: F
2: F
3: F
4: F
5: F

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

Vertex 1 is not marked, so now we move to call `dfs(1)`.

dfs(0)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

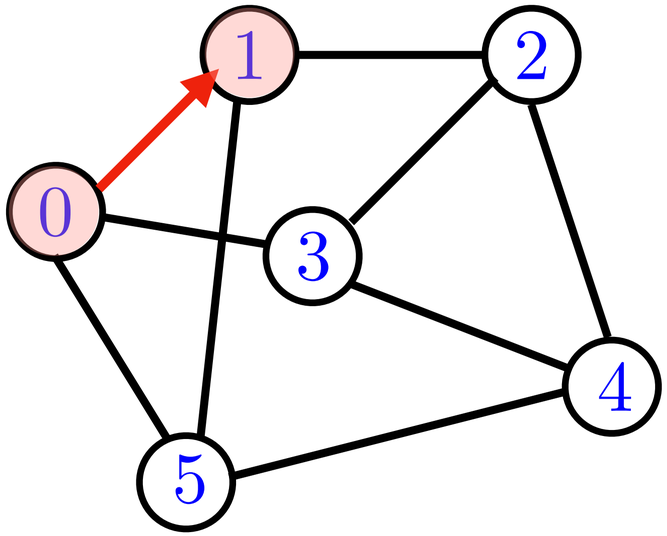
0: T
1: F
2: F
3: F
4: F
5: F

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

Vertex 1 is not marked, so now we move to call `dfs(1)`.

dfs(1)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: F
5: F

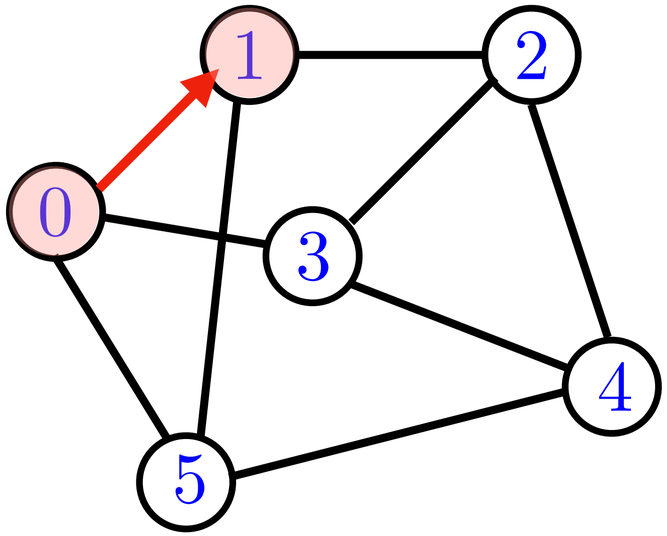
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 1.

In the for loop we first consider vertex 5.

dfs(1)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: F
5: F

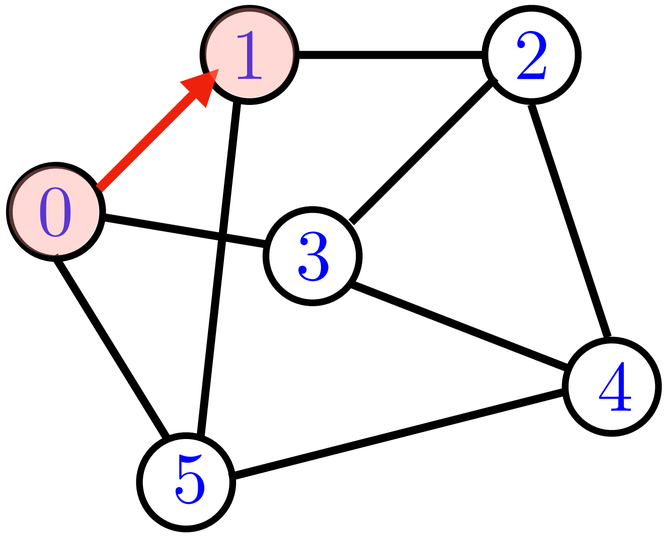
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 1.

In the for loop we first consider vertex 5.

dfs(1)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: F
5: F

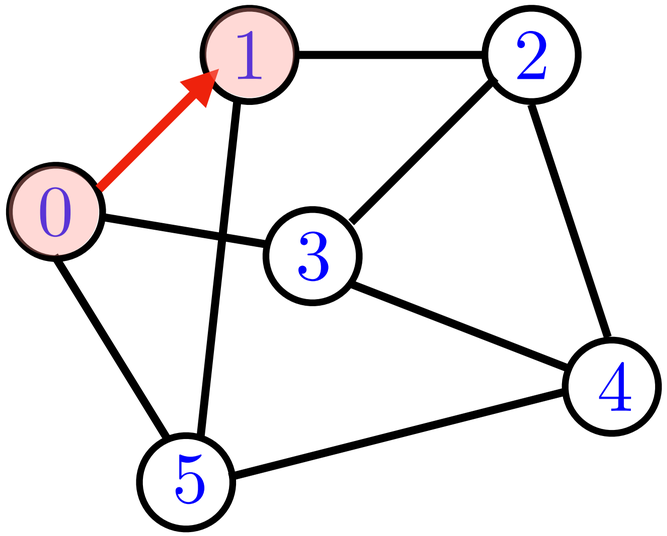
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 1.

In the for loop we first consider vertex 5.

dfs(1)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

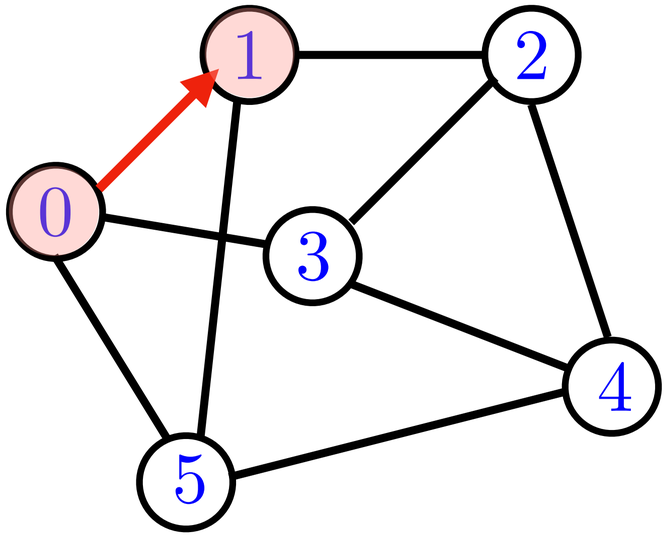
0: T
1: T
2: F
3: F
4: F
5: F

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Vertex 5 is not marked, so next we call `dfs(5)`.

dfs(1)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

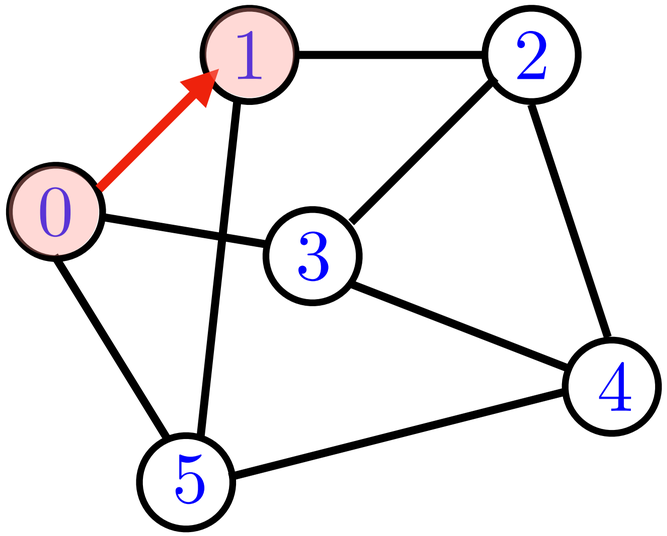
0: T
1: T
2: F
3: F
4: F
5: F

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Vertex 5 is not marked, so next we call `dfs(5)`.

dfs(1)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

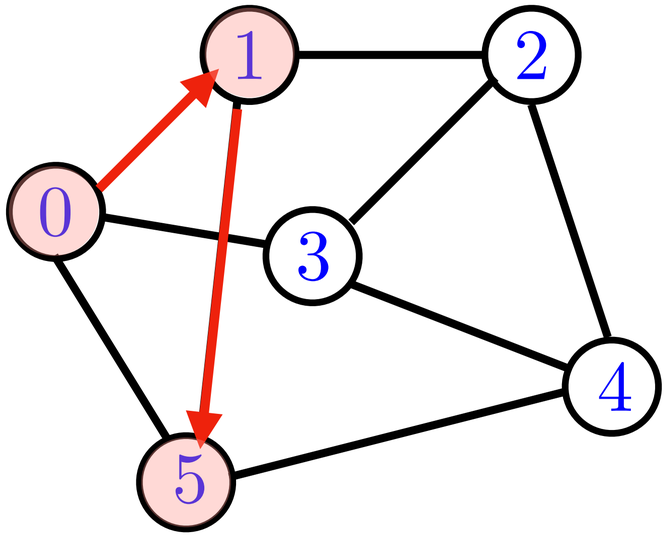
0: T
1: T
2: F
3: F
4: F
5: F

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Vertex 5 is not marked, so next we call `dfs(5)`.

dfs(1)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

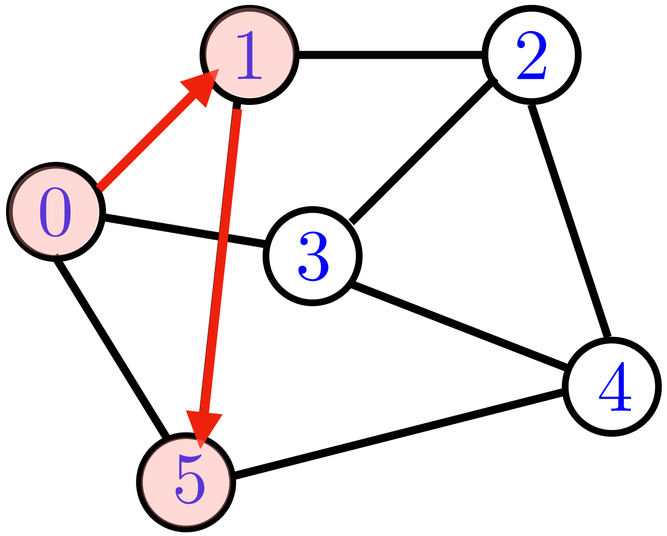
0: T
1: T
2: F
3: F
4: F
5: F

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Vertex 5 is not marked, so next we call `dfs(5)`.

dfs(5)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: F
5: T

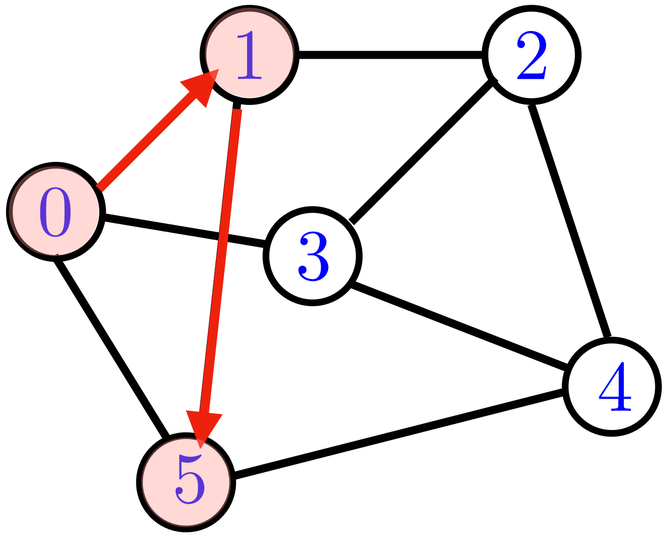
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 5.

In the for loop we first consider vertex 4, which is not marked, so we call `dfs(4)` .

dfs(5)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: F
5: T

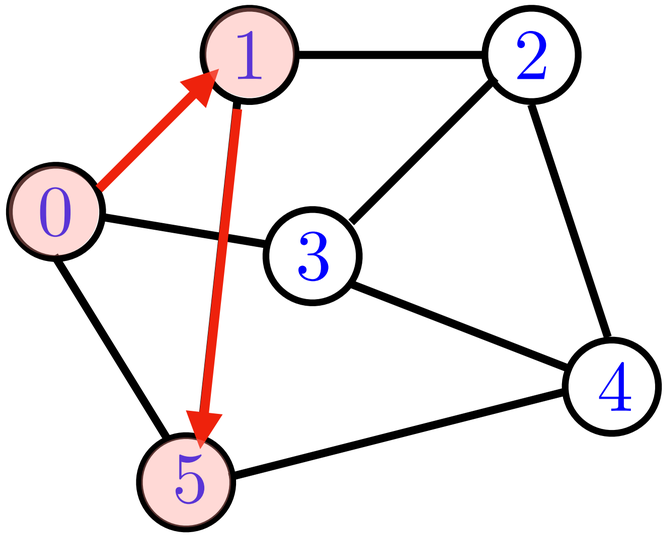
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 5.

In the for loop we first consider vertex 4, which is not marked, so we call `dfs(4)` .

dfs(5)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: F
5: T

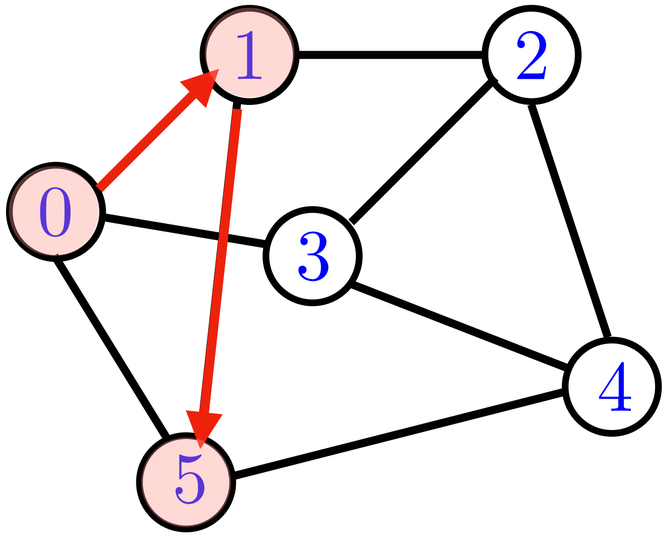
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 5.

In the for loop we first consider vertex 4, which is not marked, so we call `dfs(4)` .

dfs(5)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: F
5: T

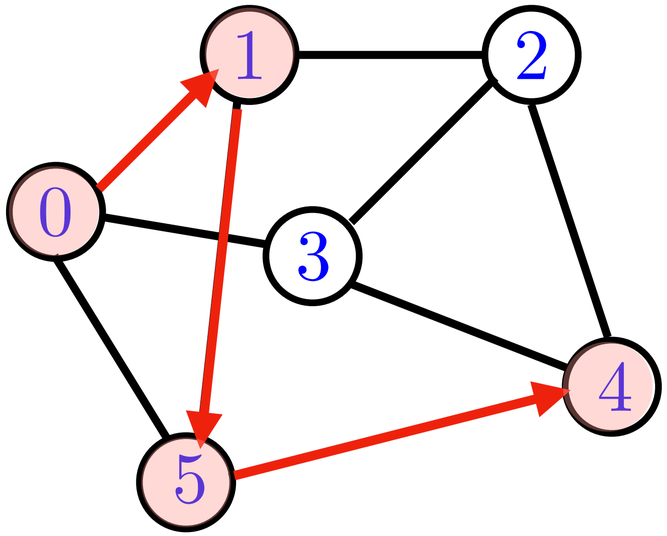
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 5.

In the for loop we first consider vertex 4, which is not marked, so we call `dfs(4)` .

dfs(5)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: F
5: T

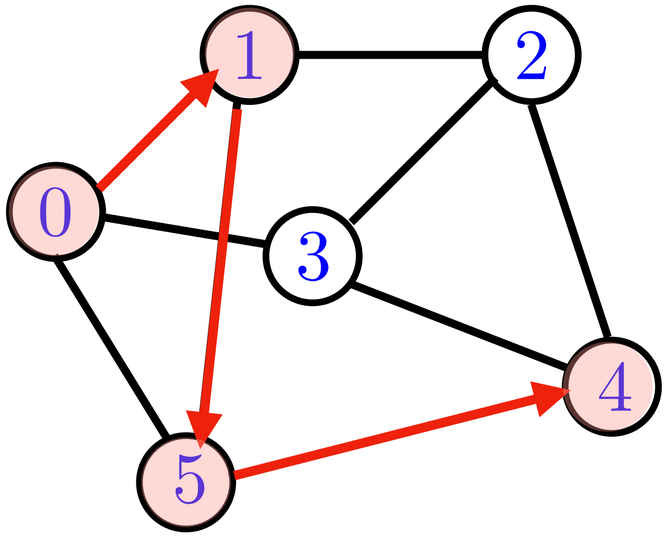
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 5.

In the for loop we first consider vertex 4, which is not marked, so we call `dfs(4)` .

dfs(4)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: T
5: T

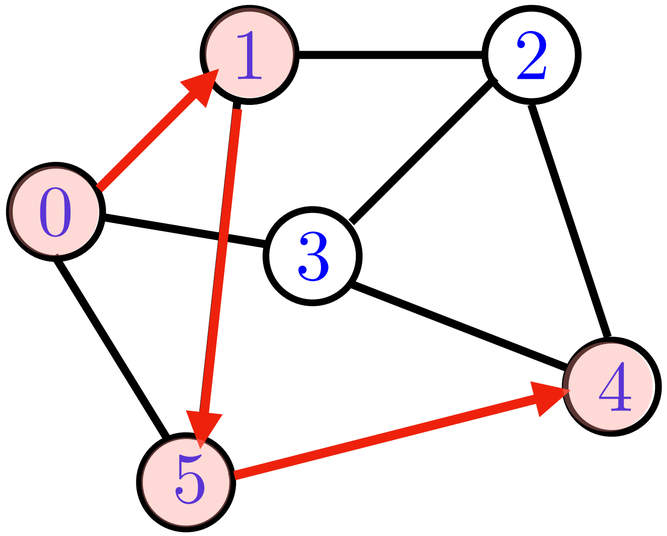
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 4.

In the for loop we first consider vertex 5.

dfs(4)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: T
5: T

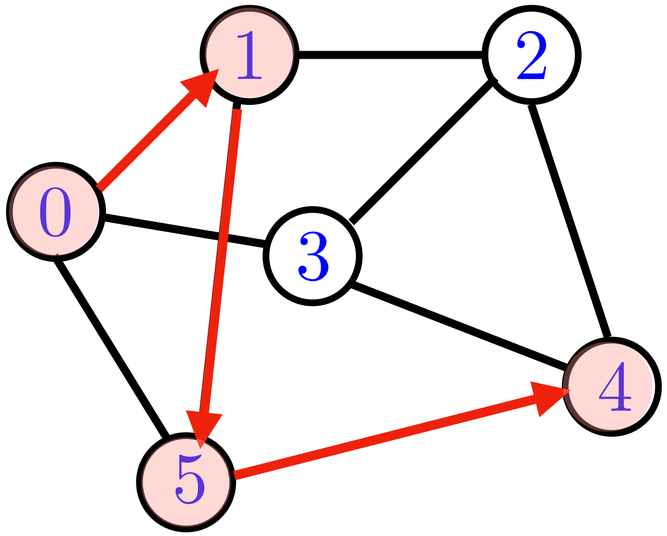
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 4.

In the for loop we first consider vertex 5.

dfs(4)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: T
5: T

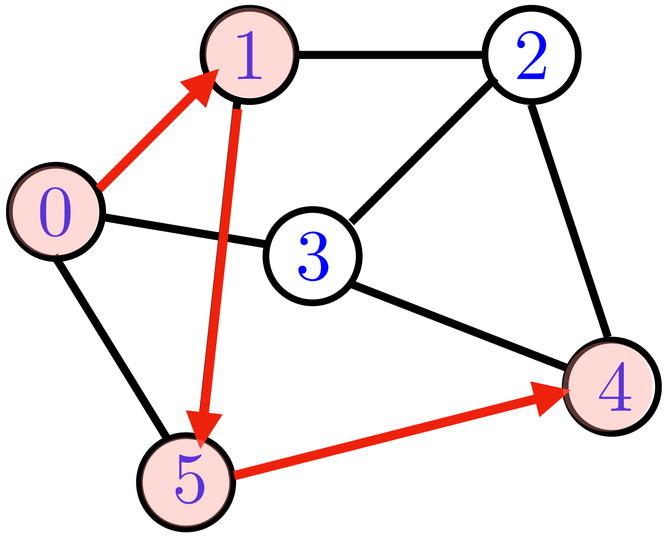
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 4.

In the for loop we first consider vertex 5.

dfs(4)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: T
5: T

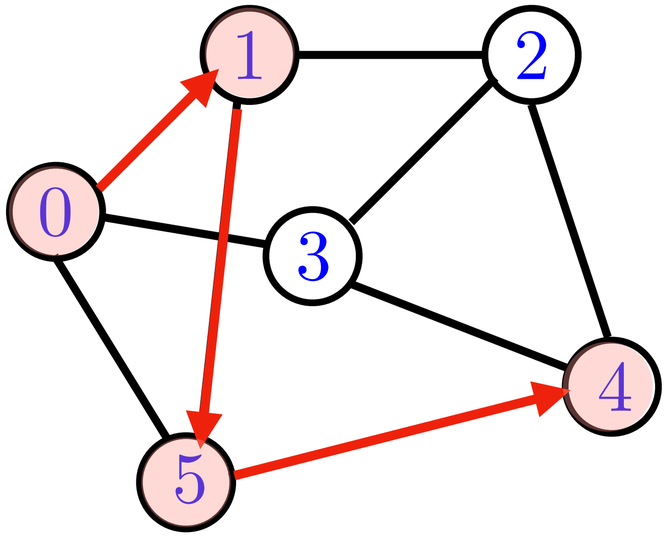
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 4.

In the for loop we first consider vertex 5.

dfs(4)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: T
5: T

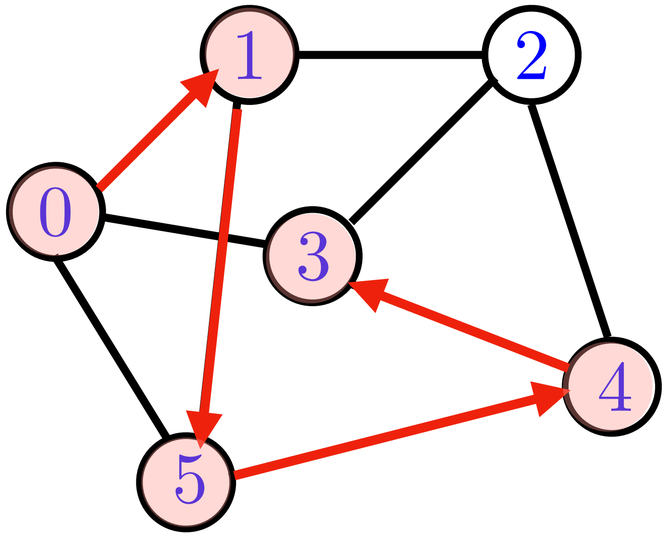
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 4.

In the for loop we first consider vertex 5.

dfs(4)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: F
4: T
5: T

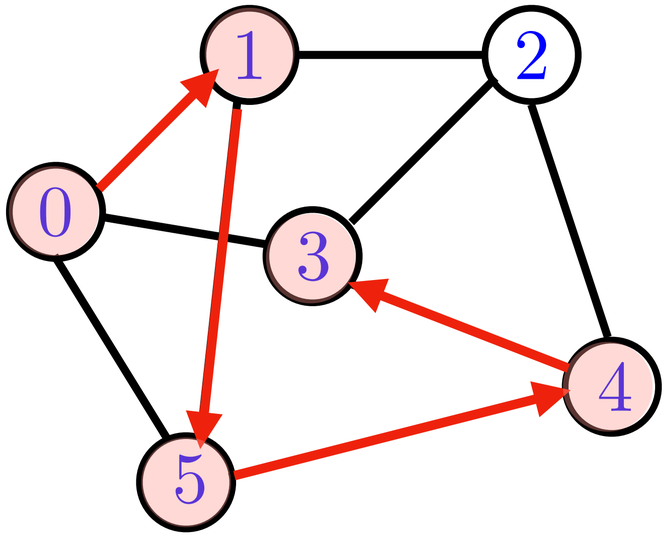
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

Now we mark vertex 4.

In the for loop we first consider vertex 5.

dfs(3)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: T
4: T
5: T

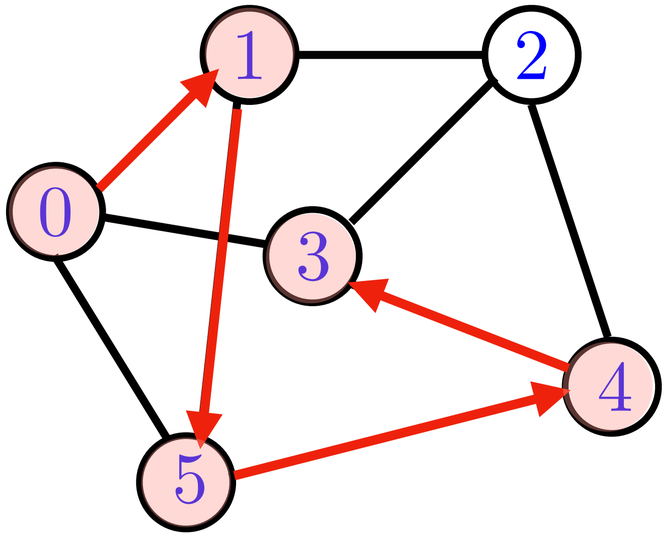
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We mark vertex 3.

Vertex 4 is already marked, so we move to vertex 2.

dfs(3)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: T
4: T
5: T

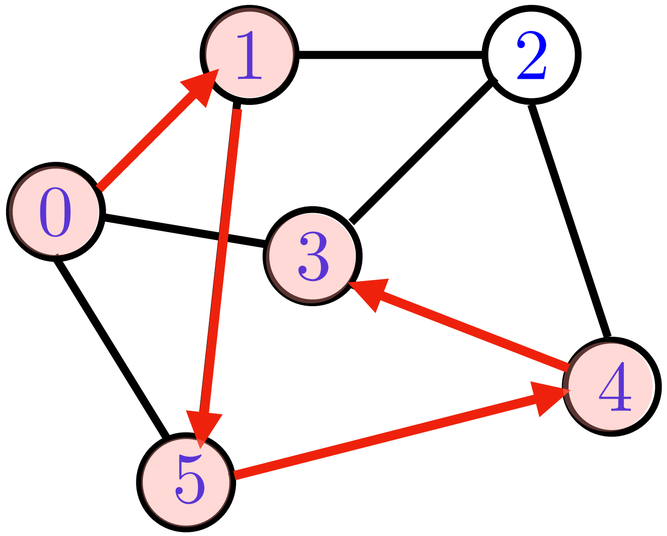
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We mark vertex 3.

Vertex 4 is already marked, so we move to vertex 2.

dfs(3)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: T
4: T
5: T

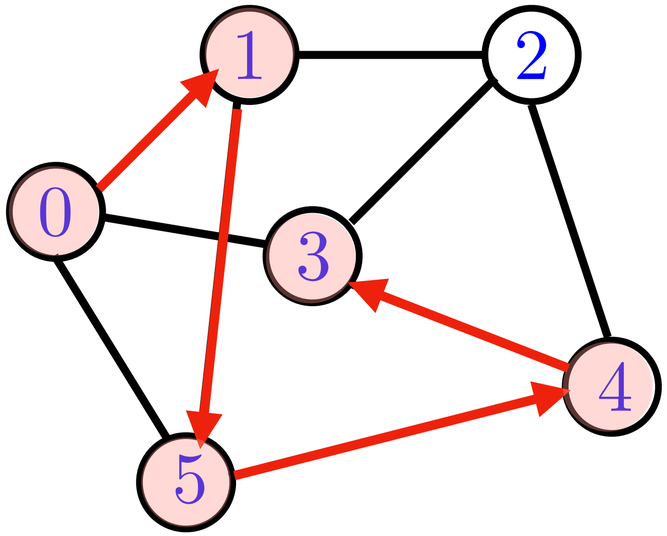
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We mark vertex 3.

Vertex 4 is already marked, so we move to vertex 2.

dfs(3)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: T
4: T
5: T

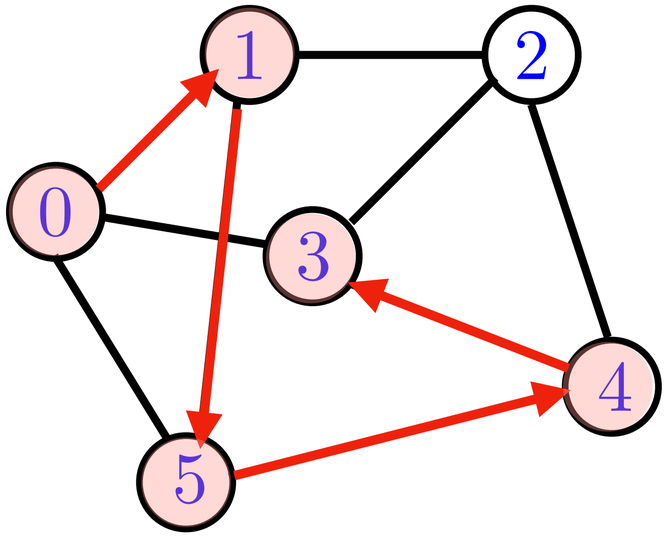
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We mark vertex 3.

Vertex 4 is already marked, so we move to vertex 2.

dfs(3)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: T
4: T
5: T

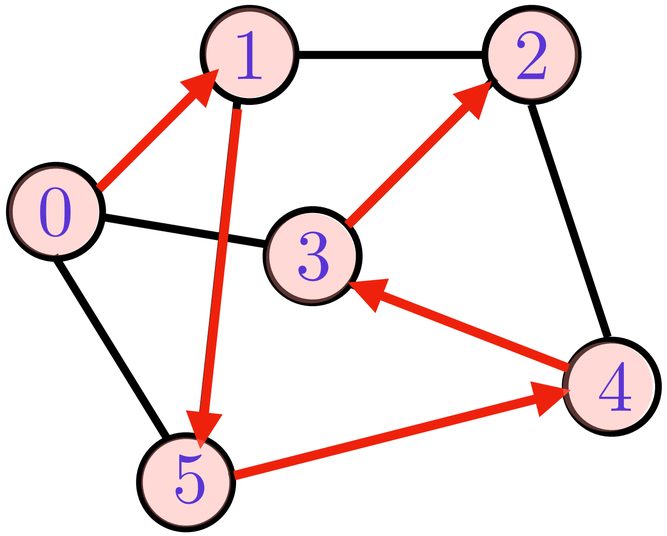
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We mark vertex 3.

Vertex 4 is already marked, so we move to vertex 2.

dfs(3)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0: T
1: T
2: F
3: T
4: T
5: T

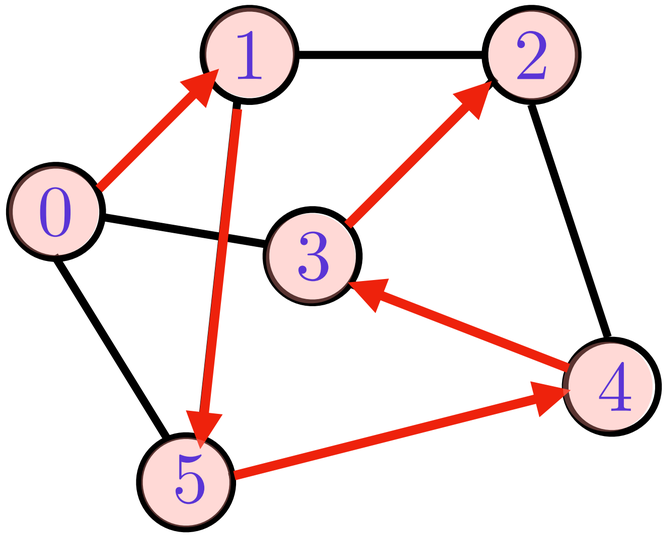
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We mark vertex 3.

Vertex 4 is already marked, so we move to vertex 2.

dfs(2)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0:T
1:T
2:T
3:T
4:T
5:T

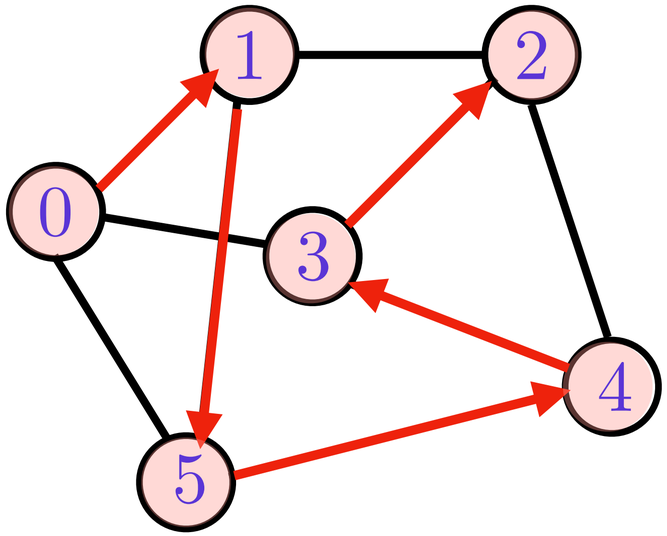
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We mark vertex 2.

Now all vertices have been marked.

dfs(2)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0:T
1:T
2:T
3:T
4:T
5:T

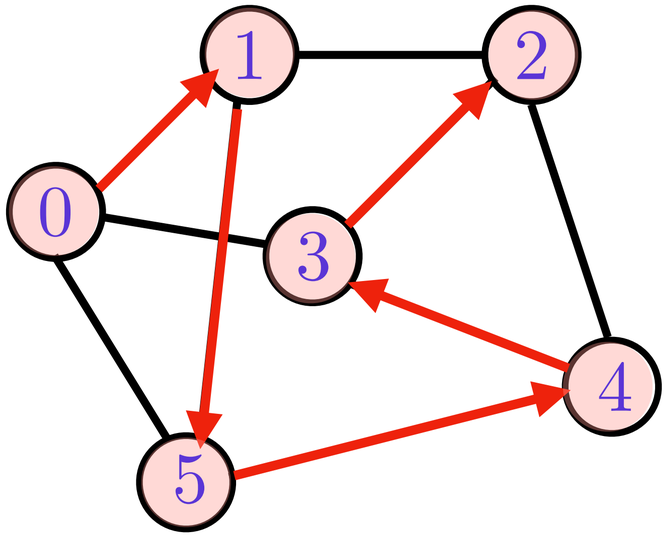
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We mark vertex 2.

Now all vertices have been marked.

dfs(2)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0:T
1:T
2:T
3:T
4:T
5:T

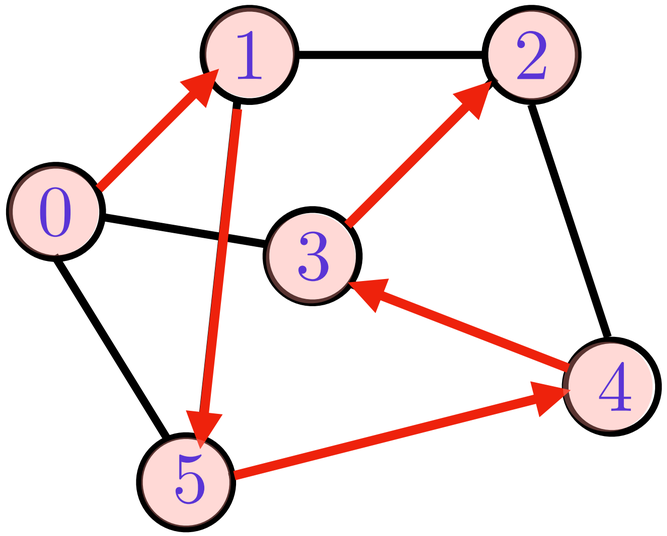
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We mark vertex 2.

Now all vertices have been marked.

dfs(2)



Adjacency List	Marked
0: 1 5 3	0:T
1: 5 2 0	1:T
2: 4 3 1	2:T
3: 4 2 0	3:T
4: 5 3 2	4:T
5: 4 1 0	5:T

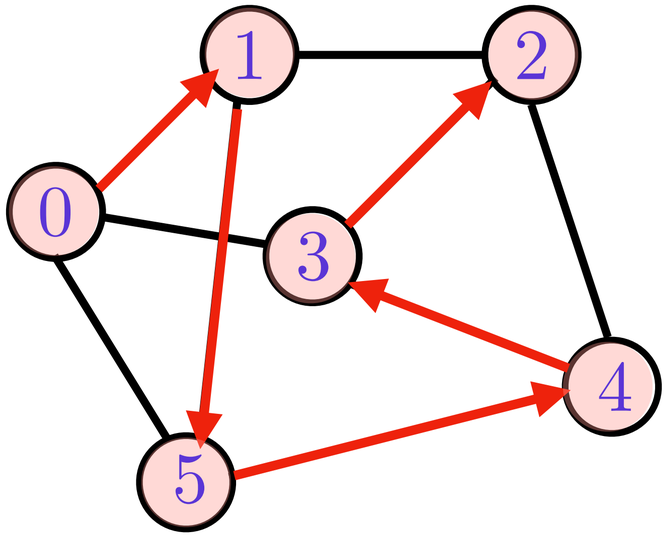
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We cycle through all neighbors of vertex 2, but do nothing as they are all marked.

The call to `dfs(2)` terminates.

dfs(2)



Adjacency List	Marked
0: 1 5 3	0:T
1: 5 2 0	1:T
2: 4 3 1	2:T
3: 4 2 0	3:T
4: 5 3 2	4:T
5: 4 1 0	5:T

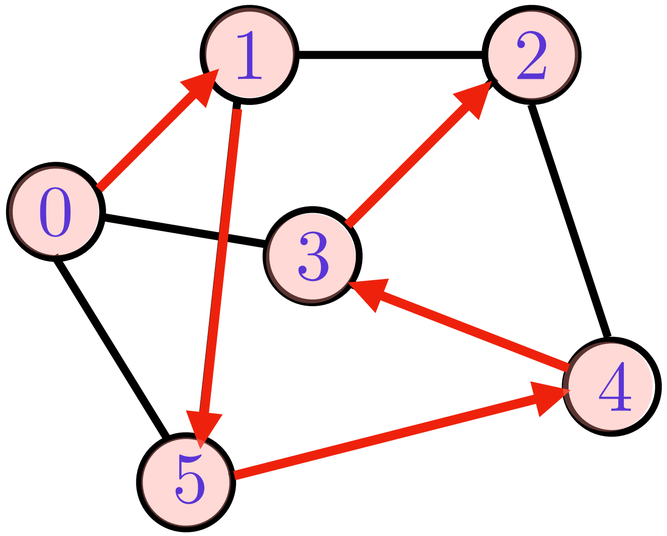
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We cycle through all neighbors of vertex 2, but do nothing as they are all marked.

The call to `dfs(2)` terminates.

dfs(2)



Adjacency List	Marked
0: 1 5 3	0:T
1: 5 2 0	1:T
2: 4 3 1	2:T
3: 4 2 0	3:T
4: 5 3 2	4:T
5: 4 1 0	5:T

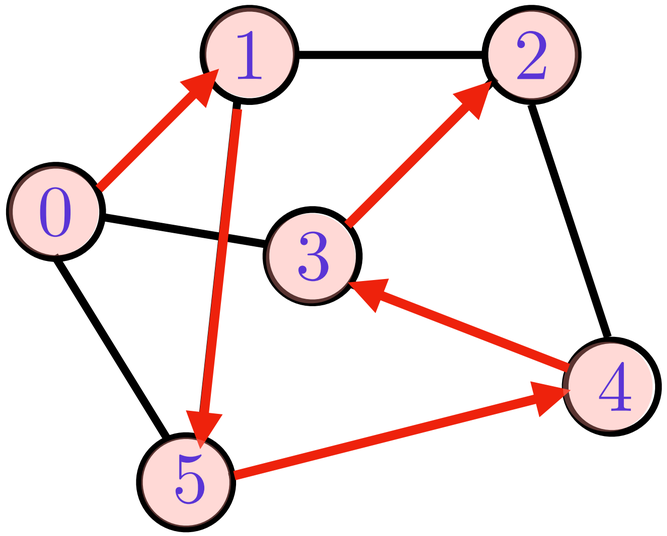
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We cycle through all neighbors of vertex 2, but do nothing as they are all marked.

The call to `dfs(2)` terminates.

dfs(2)



Adjacency List	Marked
0: 1 5 3	0:T
1: 5 2 0	1:T
2: 4 3 1	2:T
3: 4 2 0	3:T
4: 5 3 2	4:T
5: 4 1 0	5:T

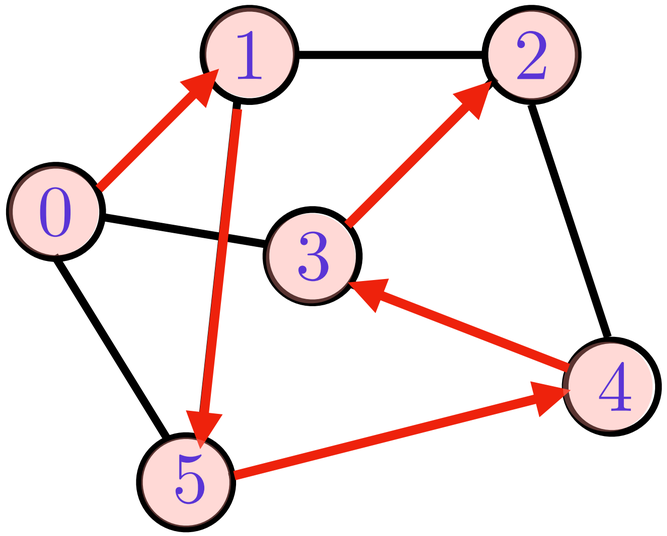
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We cycle through all neighbors of vertex 2, but do nothing as they are all marked.

The call to `dfs(2)` terminates.

dfs(3)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0:T
1:T
2:T
3:T
4:T
5:T

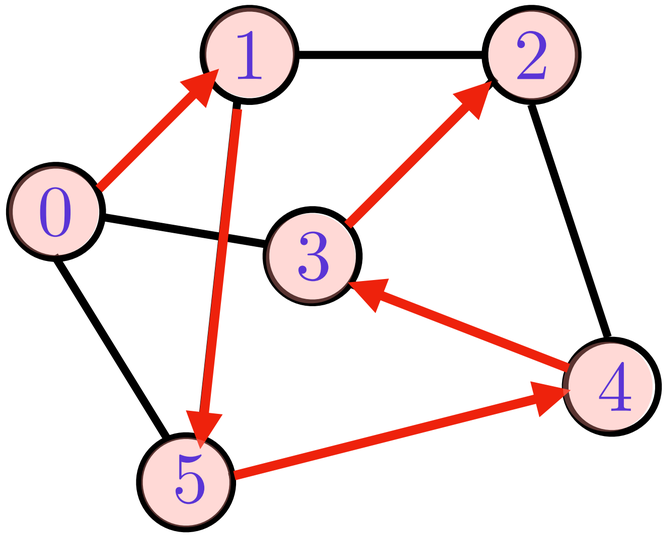
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We return to the call of `dfs(3)` and finish cycling through the neighbors.

The call to `dfs(3)` terminates.

dfs(3)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0:T
1:T
2:T
3:T
4:T
5:T

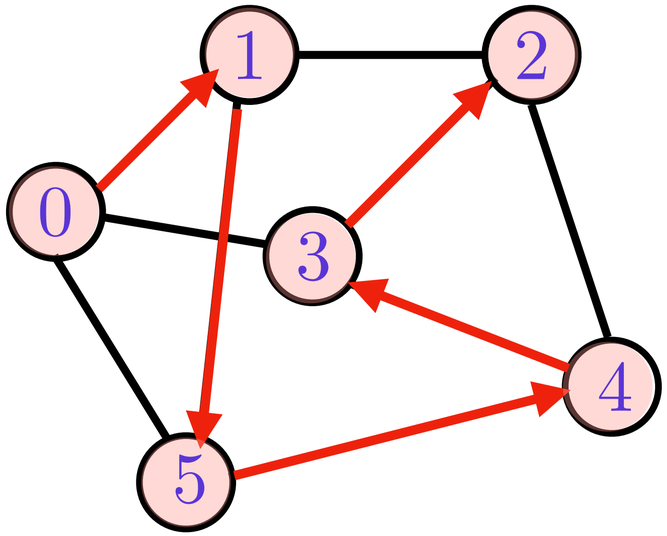
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We return to the call of `dfs(3)` and finish cycling through the neighbors.

The call to `dfs(3)` terminates.

dfs(3)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0:T
1:T
2:T
3:T
4:T
5:T

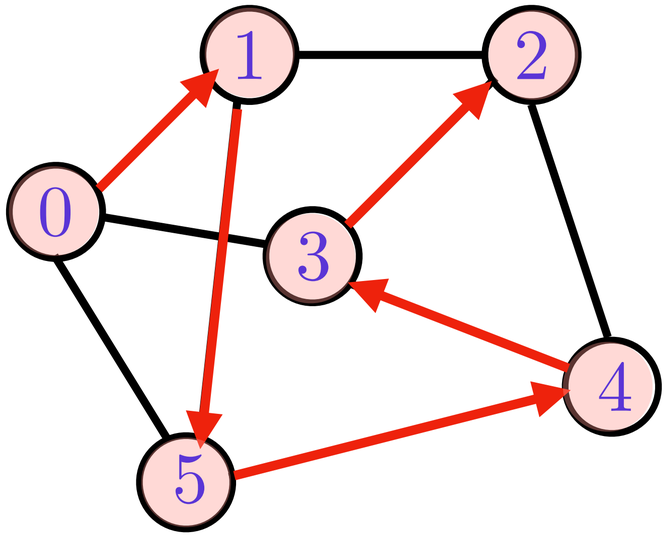
```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We return to the call of `dfs(3)` and finish cycling through the neighbors.

The call to `dfs(3)` terminates.

dfs(4)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

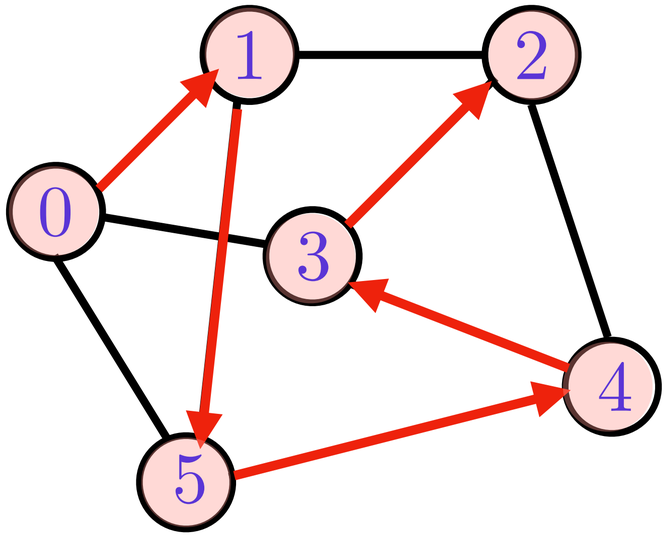
0:T
1:T
2:T
3:T
4:T
5:T

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We finish cycling through all the neighbors of vertex 4.

dfs(5)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

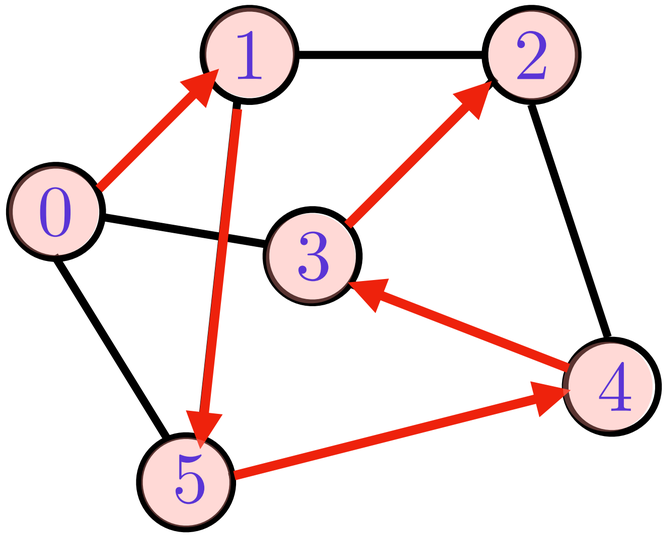
0:T
1:T
2:T
3:T
4:T
5:T

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We finish cycling through all the neighbors of vertex 5.

dfs(1)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

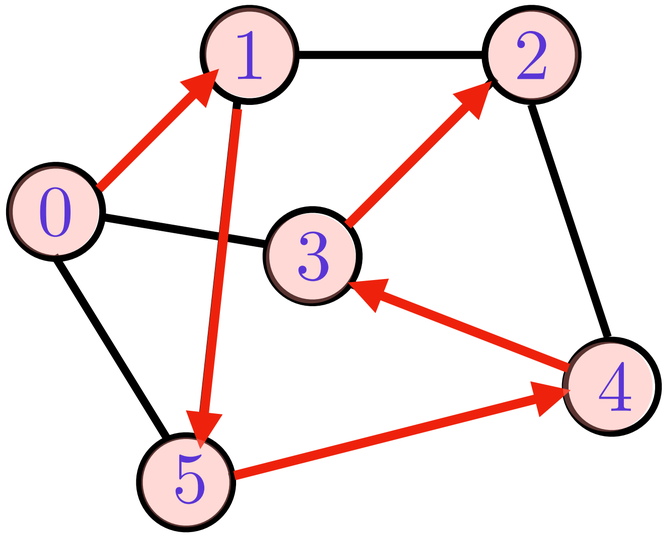
0:T
1:T
2:T
3:T
4:T
5:T

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We finish cycling through all the neighbors of vertex 1.

dfs(0)



Adjacency List

0: 1 5 3
1: 5 2 0
2: 4 3 1
3: 4 2 0
4: 5 3 2
5: 4 1 0

Marked

0:T
1:T
2:T
3:T
4:T
5:T

```
bool marked[N] {};

void dfs(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
               dfs(u);
          }
     }
}
```

We finish cycling through all the neighbors of vertex 0.

The algorithm terminates!

Depth-First Search

Running depth-first search on vertex v will mark exactly the vertices in the connected component of v.

The for loop just iterates over neighbors of the vertex we are visiting.

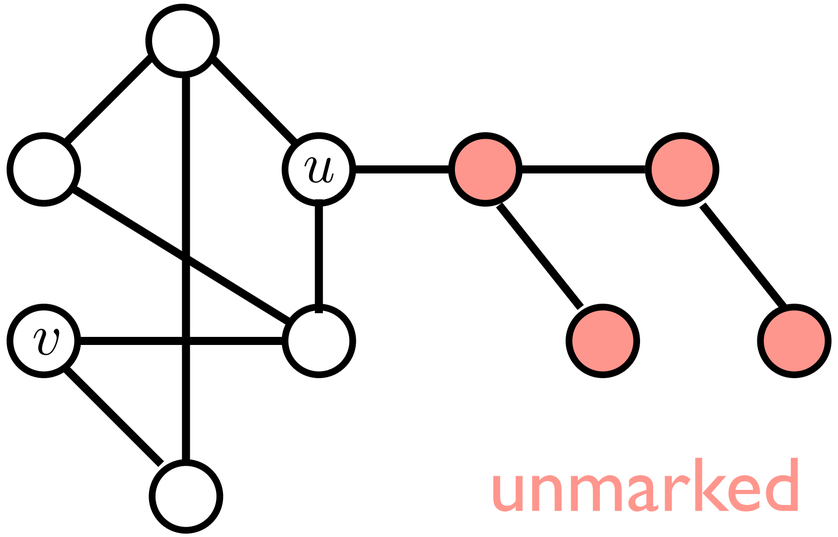
We never call $\texttt{dfs}(u)$ on a vertex u not in the connected component of v.

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

Running depth-first search on vertex v will mark exactly the vertices in the connected component of v.

If a vertex in the connected component is unmarked, there would be a marked vertex with an unmarked neighbor---but that is not possible.



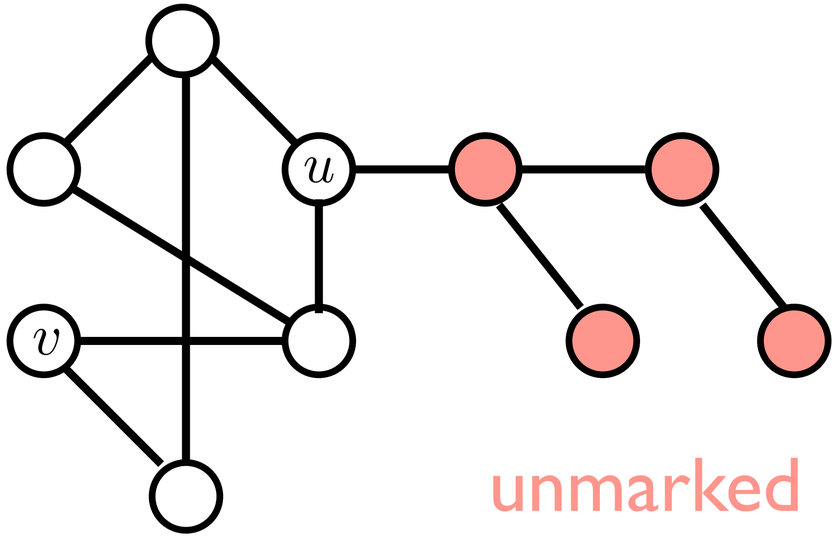
unmarked

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

Running depth-first search on vertex v will mark exactly the vertices in the connected component of v.

If a vertex in the connected component is unmarked, there would be a marked vertex with an unmarked neighbor---but that is not possible.




```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

Time Complexity

Running depth-first search on vertex v takes time proportional to the number of edges in the connected component of v in the adjacency list model.

We only call $\texttt{dfs}(u)$ once on vertex u, because after that it is marked.

The time spent in $\texttt{dfs}(u)$ is proportional to its degree in the adjacency list model.

```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

Further Additions

We can use depth-first search to find all connected components by wrapping it in a loop that performs `dfs` on any vertex still not marked.

The running time becomes $O(|V| + |E|)$.

We can also use depth-first search to find paths between vertices, not just whether or not they are connected.

For this we use another array that records from where a vertex was visited (we will talk more about this in the next video).

Iterative DFS

We can straightforwardly transform the recursive version of DFS into an iterative one by using a stack data structure.

The stack simulates the call function stack in the recursive version.

Iterative DFS

```
void iterative_dfs(unsigned v)
{
    visit_stack.push(v);
    while(!visit_stack.empty())
    {
        unsigned x = visit_stack.top();
        visit_stack.pop();
        if(marked[x])
        {
            continue;
        }
        marked[x] = true;
        print_marked();
        for(auto u : arr[x])
        {
            if(!marked[u])
            {
                visit_stack.push(u);
            }
        }
    }
}
```

To get exactly the same visit order as the recursive one we should push vertices in `arr[x]` in reverse order.

We allow a vertex to be pushed on the stack more than once in order to simulate the visit order of recursive DFS.

https://godbolt.org/z/ovq99ebGc

Breadth-First Search

Shortest Paths

Let us continue working on an undirected and unweighted graph.

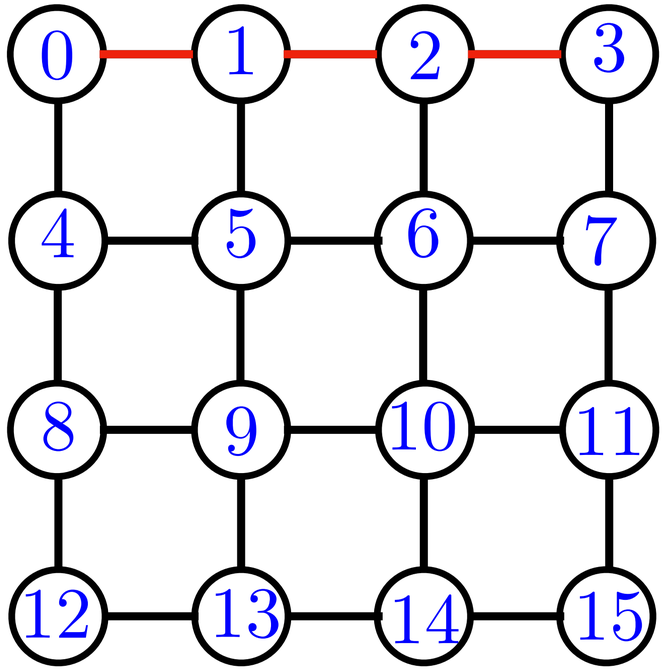
Single-source shortest paths problem: Given a vertex v, find shortest paths from v to all other vertices in its connected component.

Shortest Paths

Let us continue working on an undirected and unweighted graph.

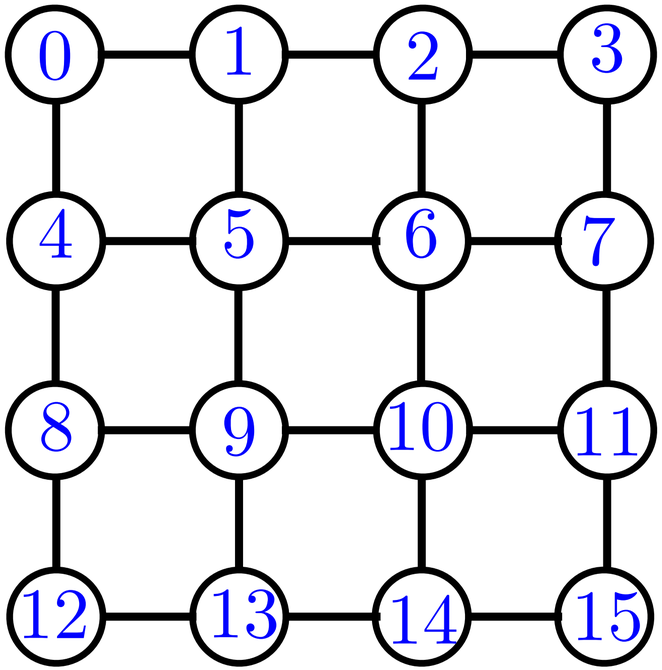
Single-source shortest paths problem: Given a vertex v, find shortest paths from v to all other vertices in its connected component.

Example graph:



The shortest path from vertex 0 to 3 is of length 3.

Single-source shortest paths problem: Given a vertex v, find shortest paths from v to all other vertices in its connected component.



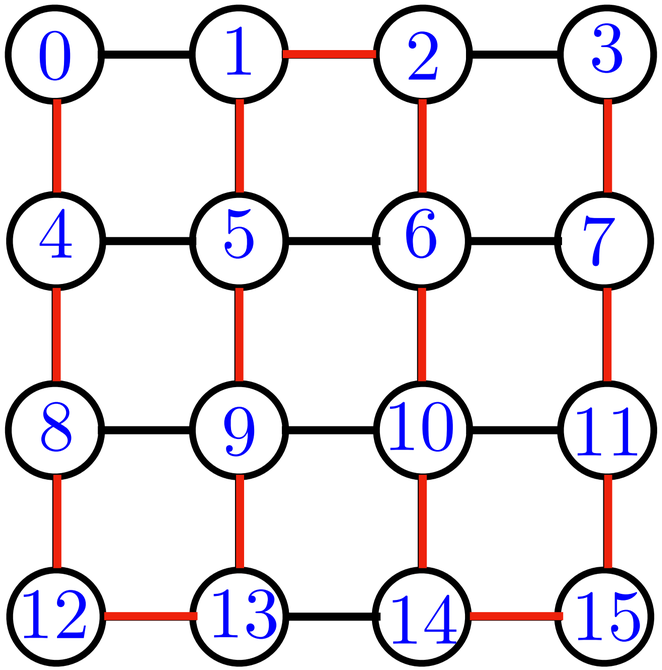
Say that vertical neighbors always precede horizontal neighbors in the adjacency lists.

What is the path taken by depth-first search from vertex 0 to vertex 3?

Definitely not the shortest path!

https://godbolt.org/z/54KT9v3Ec

Single-source shortest paths problem: Given a vertex v, find shortest paths from v to all other vertices in its connected component.



Say that vertical neighbors always precede horizontal neighbors in the adjacency lists.

What is the path taken by depth-first search from vertex 0 to vertex 3?

Definitely not the shortest path!

https://godbolt.org/z/54KT9v3Ec

Breadth-First Search

Another classic graph algorithm, breadth-first search, can solve the single source shortest path problem in an unweighted graph.

To do breadth-first search we can take the iterative version of DFS and replace the stack with a queue.

```
void bfs(unsigned v)
{
    visit_queue.push(v);
    marked[v] = true;
    while(!visit_queue.empty())
    {
        unsigned x = visit_queue.front();
        visit_queue.pop();
        for(auto u : arr[x])
        {
            if(!marked[u])
            {
                visit_queue.push(u);
                marked[u] = true;
                // we came to u from x
                edge_to[u] = x;
            }
        }
    }
}
```

`visit_queue`

queue that holds vertices where we still need to explore their neighbors.

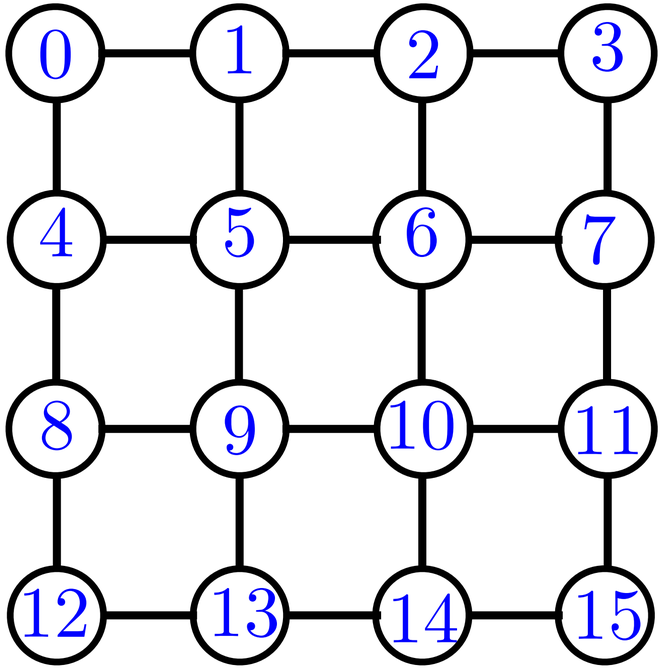
`marked`

array of bools, indicates if vertex has been added to queue.

`edge_to`

array of vertices. Tells vertex we came from to visit a vertex.

https://godbolt.org/z/Kac9W4d4G



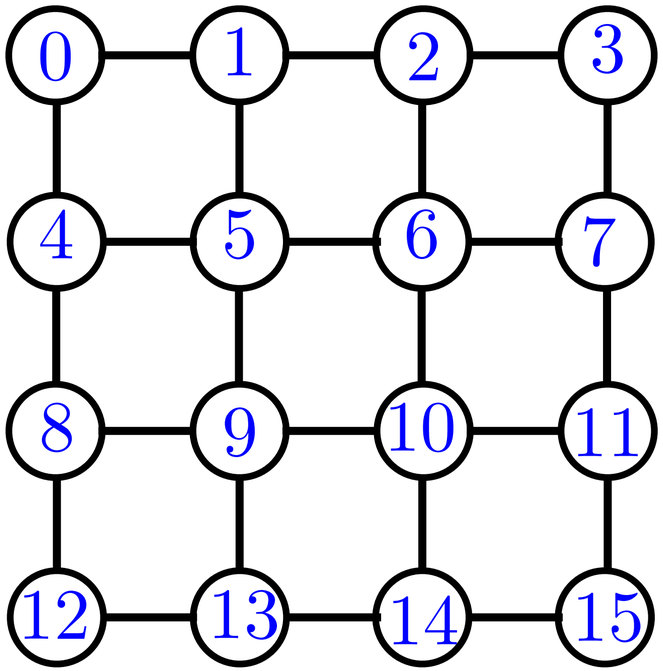
visit_queue

0

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Start at vertex 0.

0 is marked and popped from the queue.

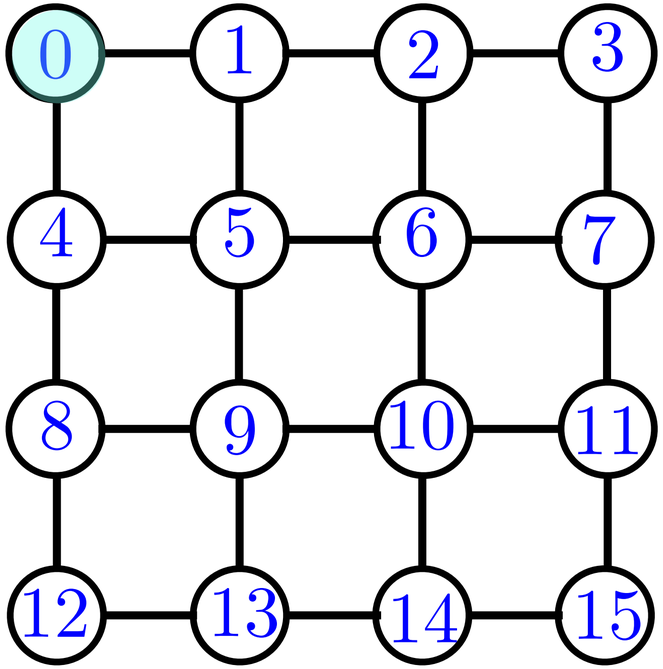


visit_queue

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Start at vertex 0.

0 is marked and popped from the queue.

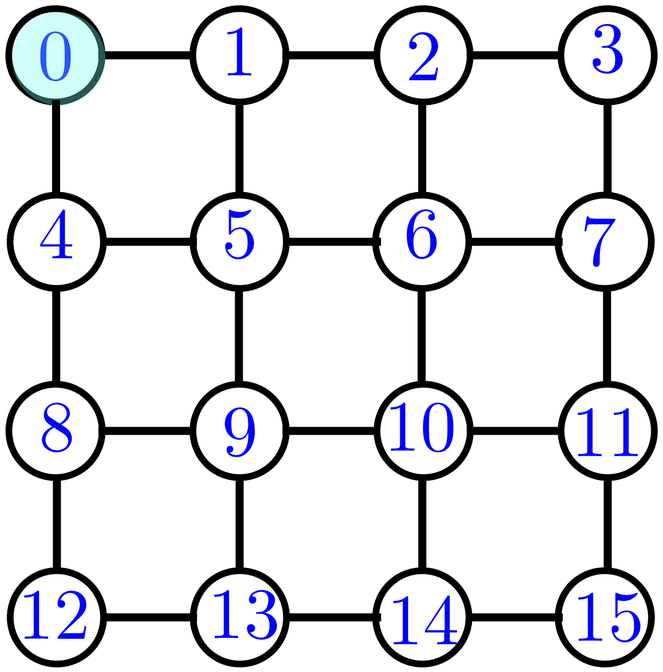


visit_queue

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Start at vertex 0.

0 is marked and popped from the queue.



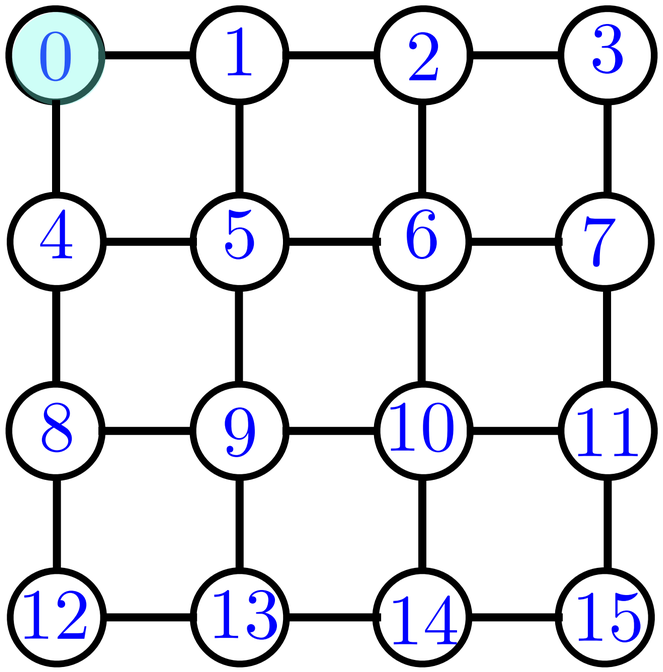
visit_queue

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Add the neighbors of 0 to the queue, mark them, and indicate we visited them from 0.

$$\texttt{edge\_to}[4] = 0$$

$$\texttt{edge\_to}[1] = 0$$



visit_queue

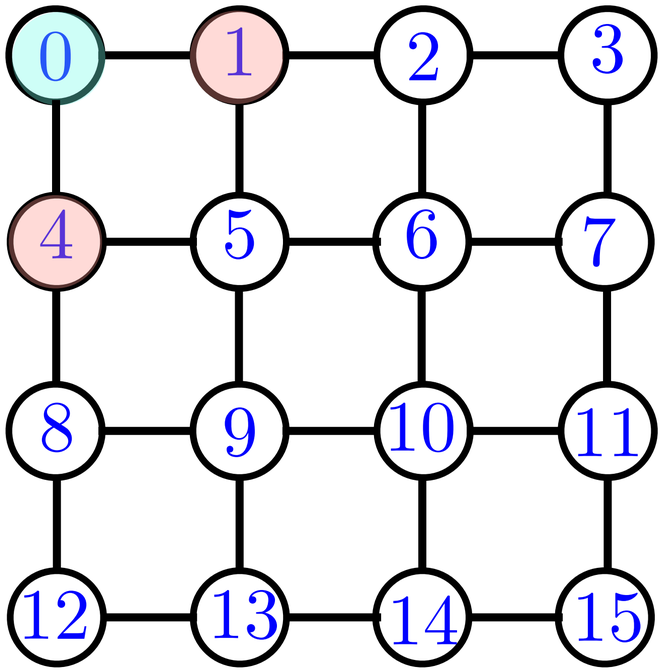
4
1

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Add the neighbors of 0 to the queue, mark them, and indicate we visited them from 0.

$$\texttt{edge\_to}[4] = 0$$

$$\texttt{edge\_to}[1] = 0$$



visit_queue

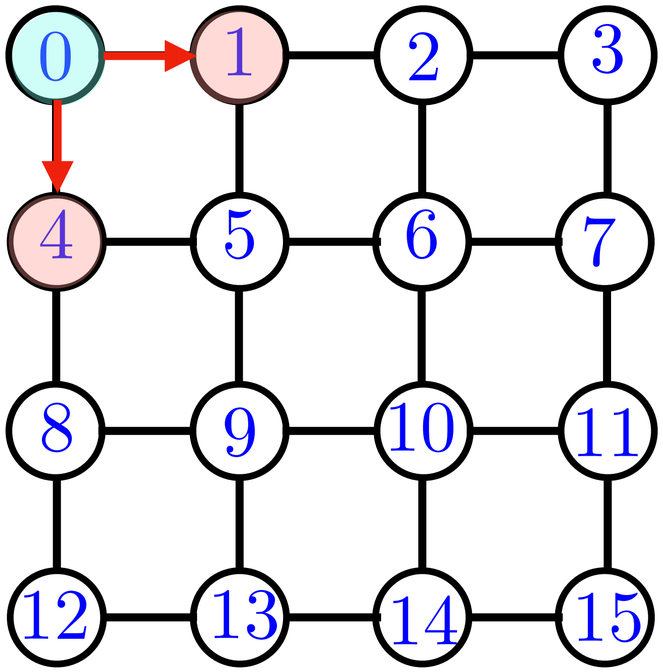
4
1

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Add the neighbors of 0 to the queue, mark them, and indicate we visited them from 0.

$$\texttt{edge\_to}[4] = 0$$

$$\texttt{edge\_to}[1] = 0$$



visit_queue

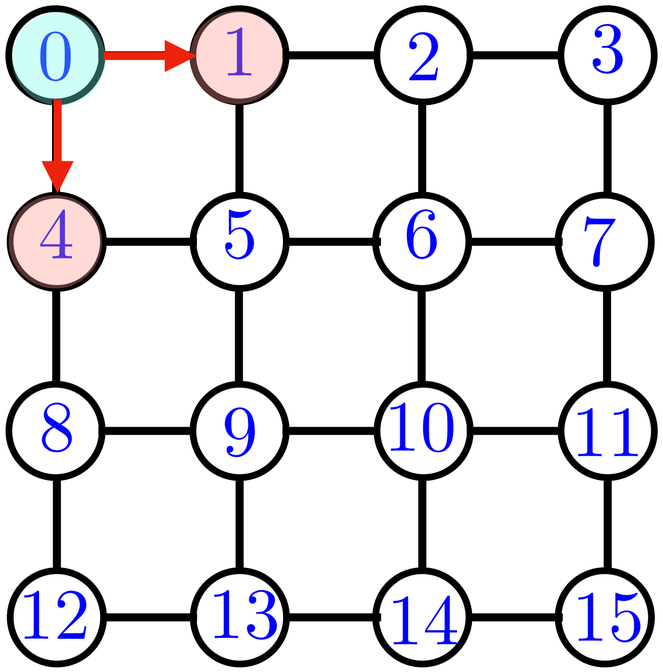
4
1

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Add the neighbors of 0 to the queue, mark them, and indicate we visited them from 0.

$$\texttt{edge\_to}[4] = 0$$

$$\texttt{edge\_to}[1] = 0$$



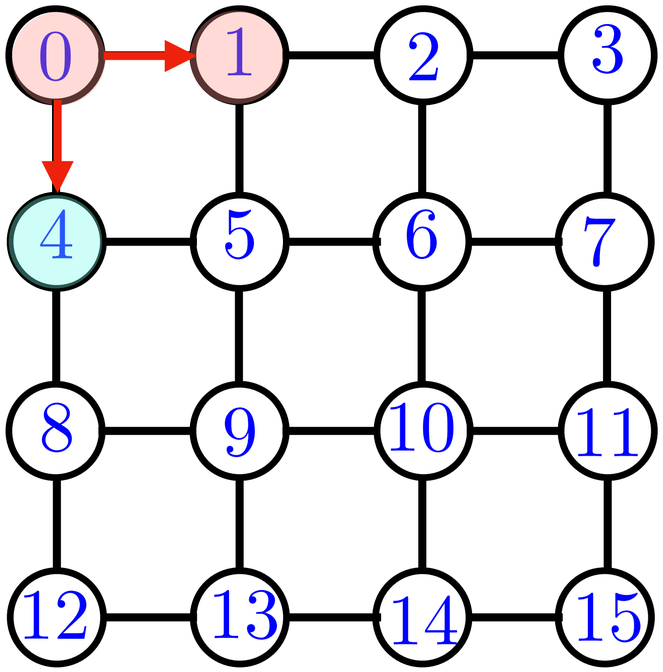
visit_queue

4

1

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

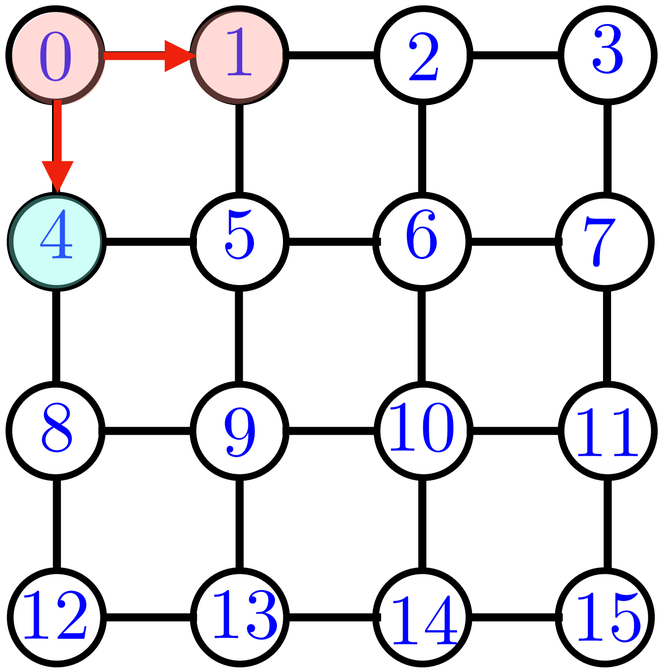
Note that the vertices in the queue are exactly those vertices at distance 1 from vertex 0.



visit_queue

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

We process the oldest item in the queue, which is vertex 4.

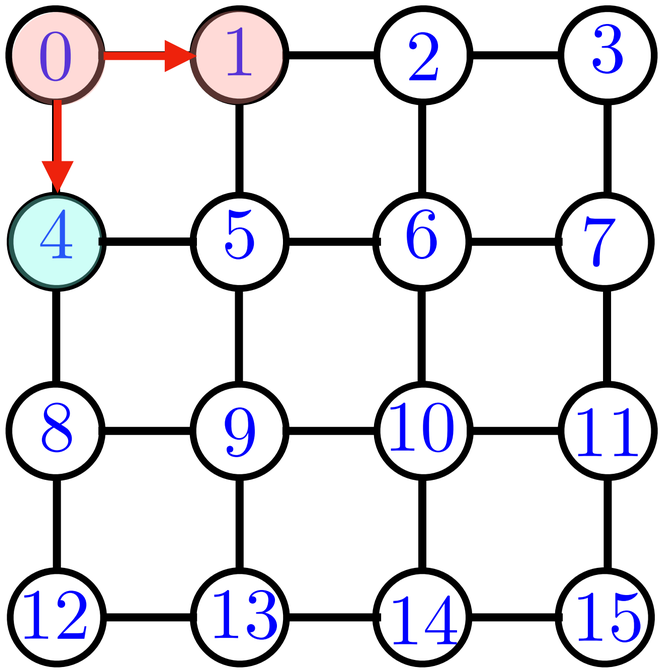


visit_queue

4
1

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

We process the oldest item in the queue, which is vertex 4.



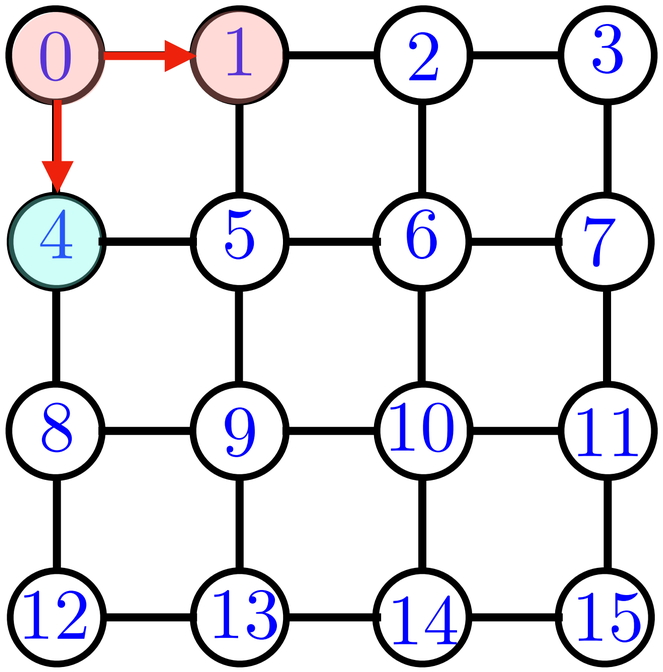
visit_queue

1

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

4 is popped out of the queue, and we add its unmarked neighbors to the queue and mark them.

All vertices in the queue are at distance 1 or 2 from vertex 0, and those at distance 1 precede those at distance 2.



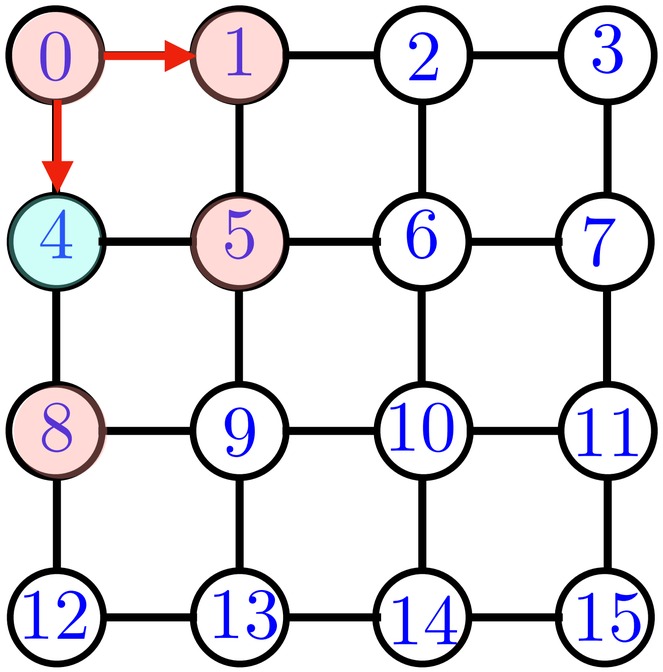
visit_queue

1
8
5

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

4 is popped out of the queue, and we add its unmarked neighbors to the queue and mark them.

All vertices in the queue are at distance 1 or 2 from vertex 0, and those at distance 1 precede those at distance 2.



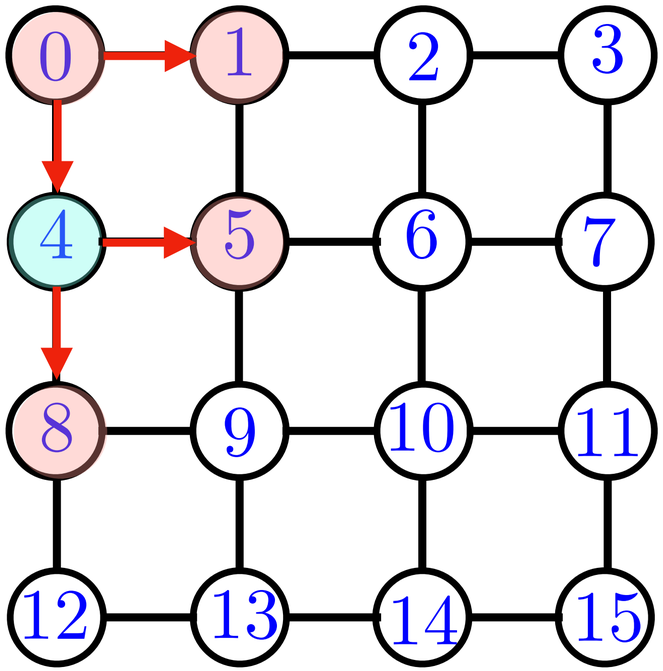
visit_queue

1
8
5

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

4 is popped out of the queue, and we add its unmarked neighbors to the queue and mark them.

All vertices in the queue are at distance 1 or 2 from vertex 0, and those at distance 1 precede those at distance 2.



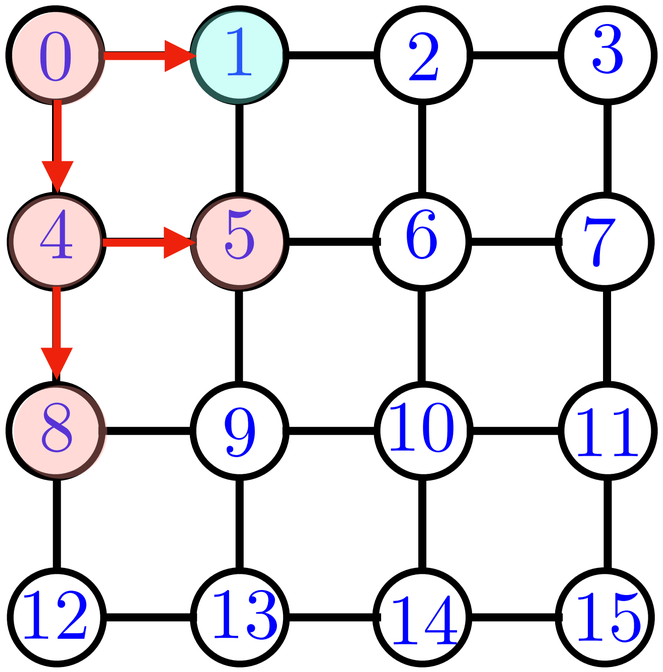
visit_queue

1
8
5

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

4 is popped out of the queue, and we add its unmarked neighbors to the queue and mark them.

All vertices in the queue are at distance 1 or 2 from vertex 0, and those at distance 1 precede those at distance 2.

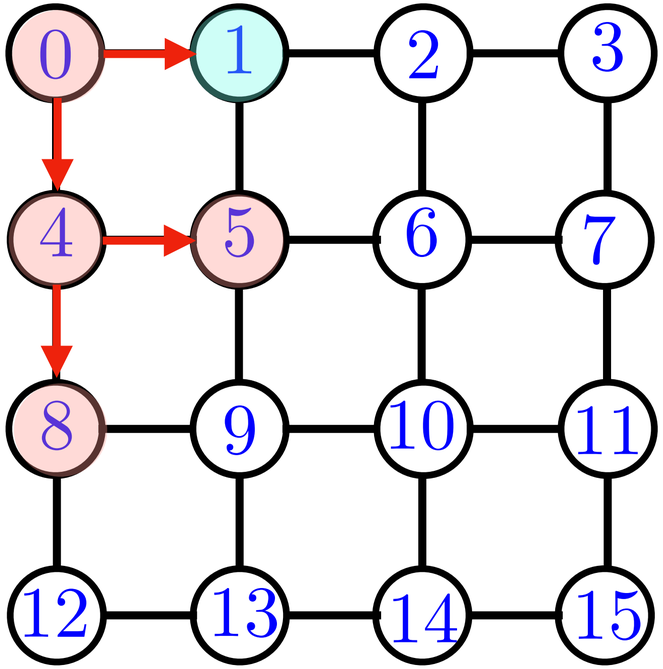


visit_queue

1
8
5

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Next we pop vertex 1 out of the queue.



visit_queue

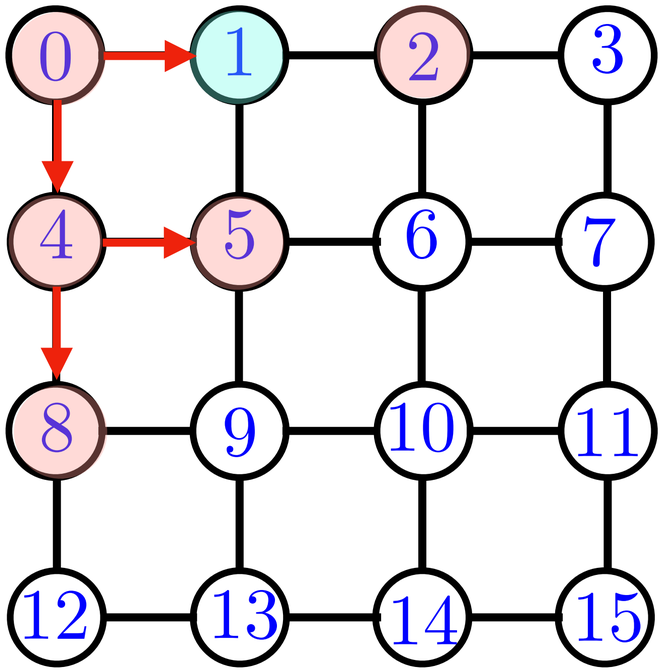
8
5
2

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

We add the unmarked neighbors of vertex 1 to the queue, and mark them.

We have finished exploring all vertices at distance 1 from vertex 0.

All vertices in the queue are at distance 2 from vertex 0.



visit_queue

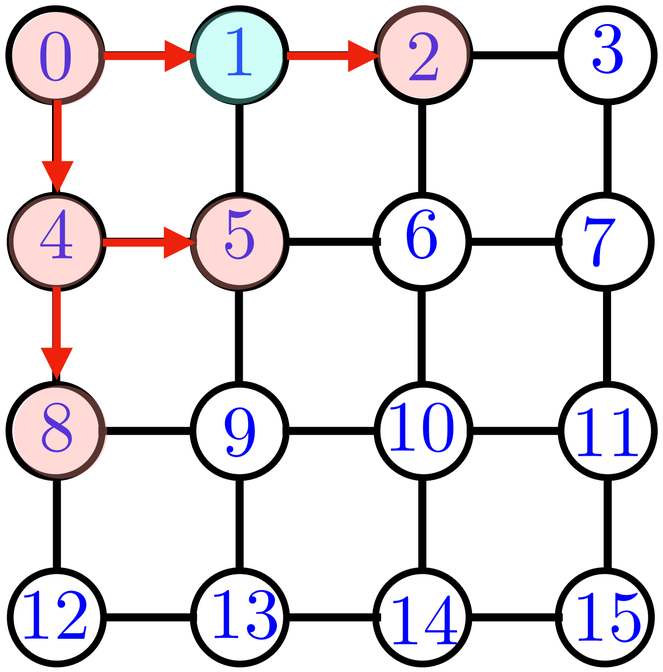
8
5
2

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

We add the unmarked neighbors of vertex 1 to the queue, and mark them.

We have finished exploring all vertices at distance 1 from vertex 0.

All vertices in the queue are at distance 2 from vertex 0.



visit_queue

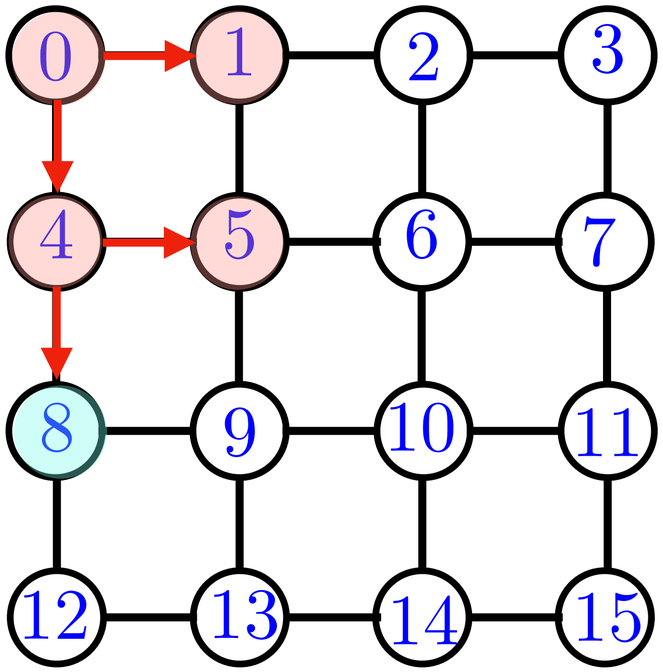
8
5
2

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

We add the unmarked neighbors of vertex 1 to the queue, and mark them.

We have finished exploring all vertices at distance 1 from vertex 0.

All vertices in the queue are at distance 2 from vertex 0.

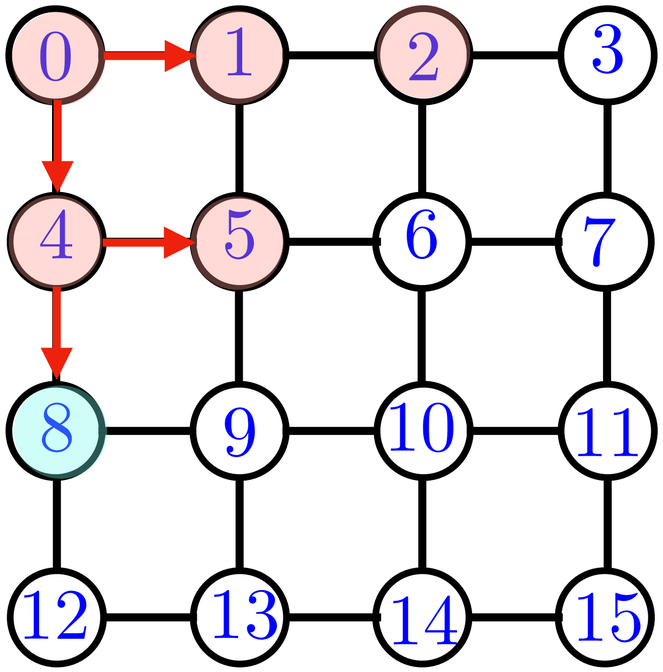


visit_queue

8
5
2

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 8 next.

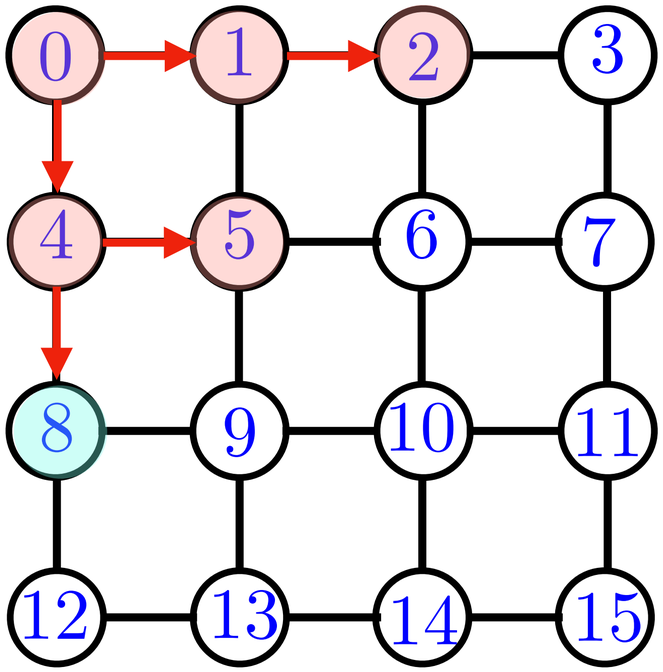


visit_queue

8
5
2

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 8 next.

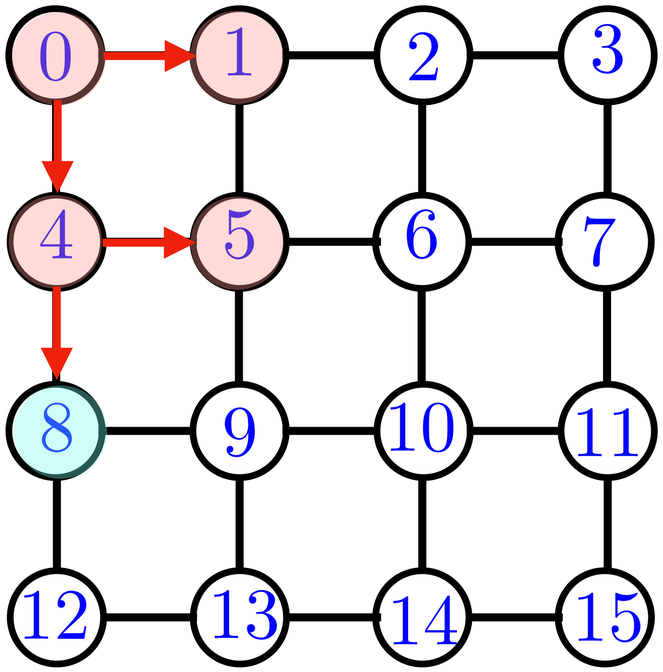


visit_queue

8
5
2

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 8 next.

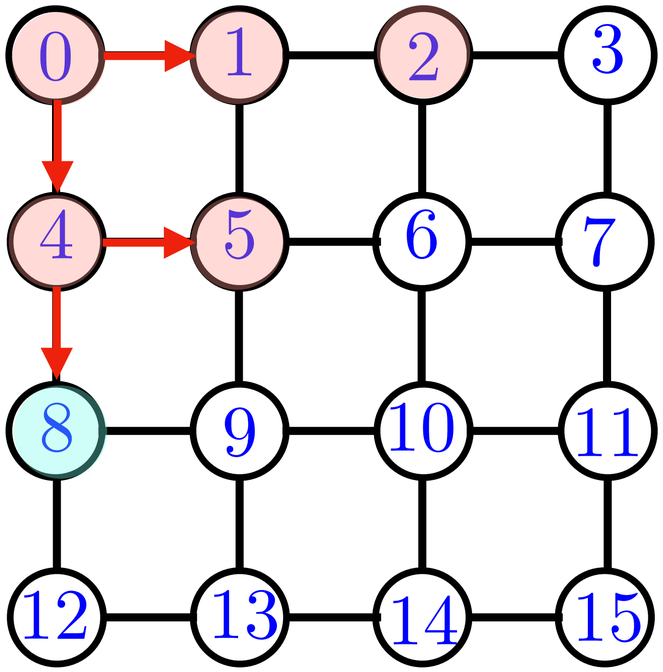


visit_queue

5
2

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 8 next.

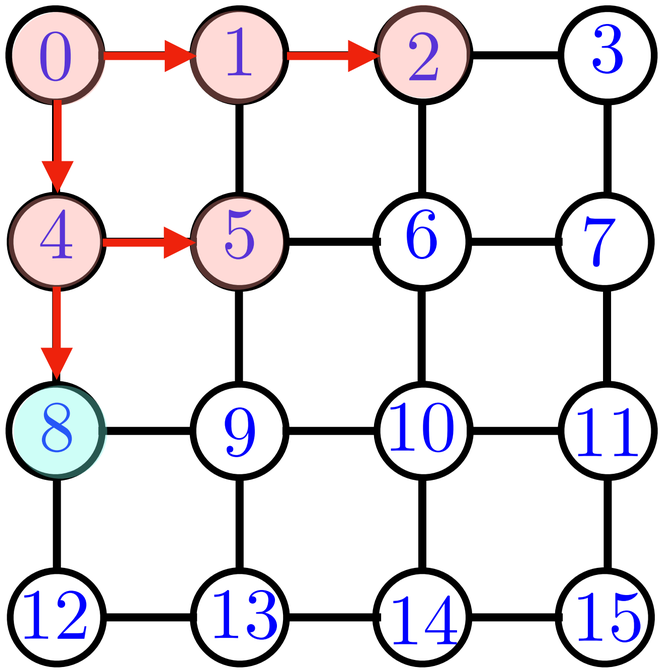


visit_queue

5
2

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 8 next.

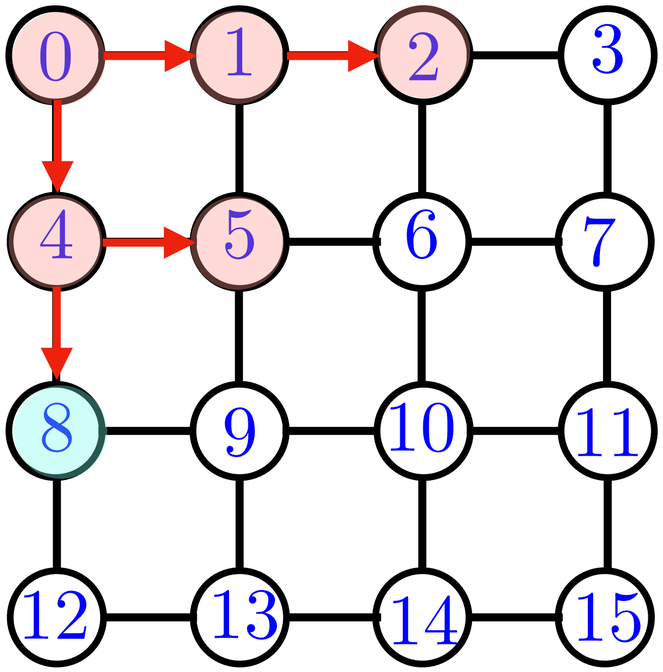


visit_queue

5
2

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 8 next.

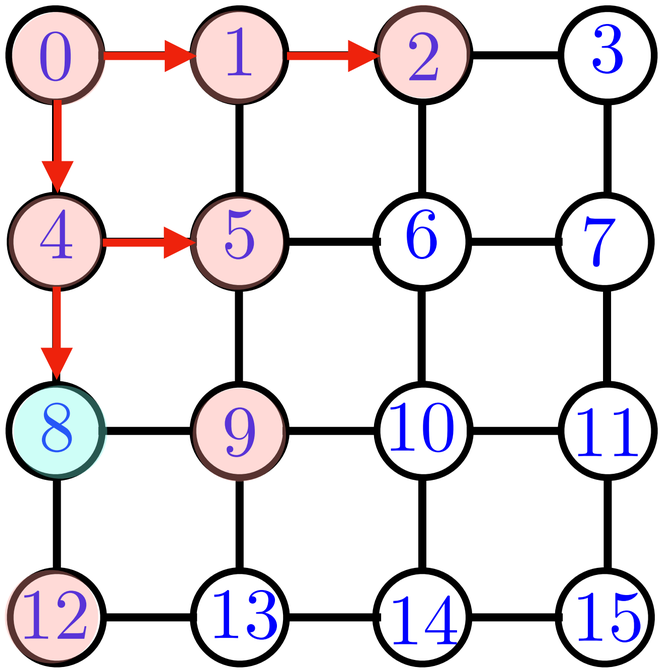


visit_queue

5
2
12
9

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 8 next.

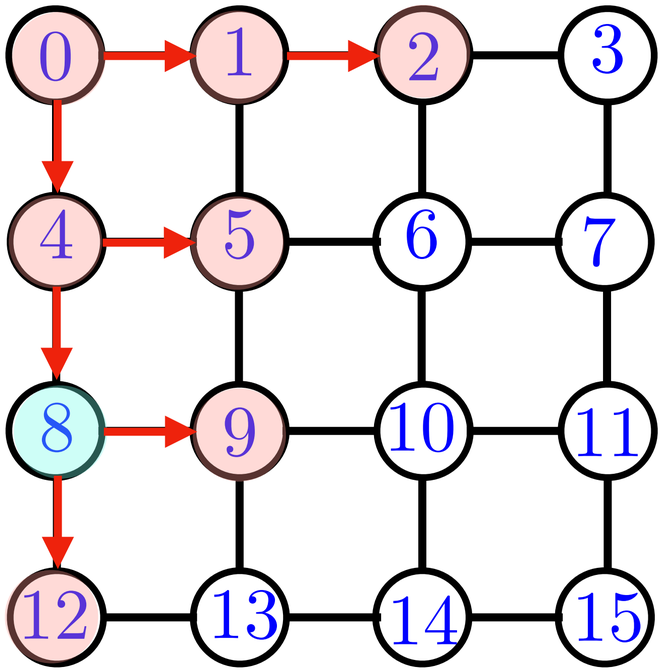


visit_queue

5
2
12
9

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 8 next.

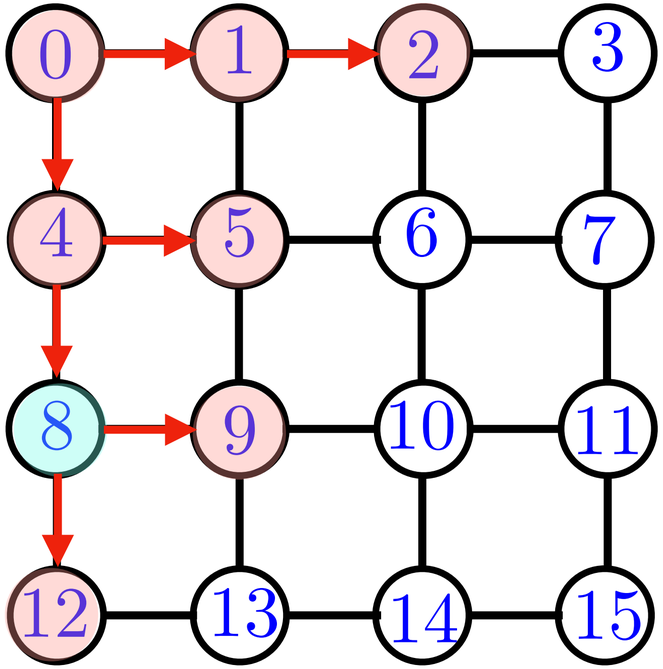


visit_queue

5
2
12
9

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 8 next.



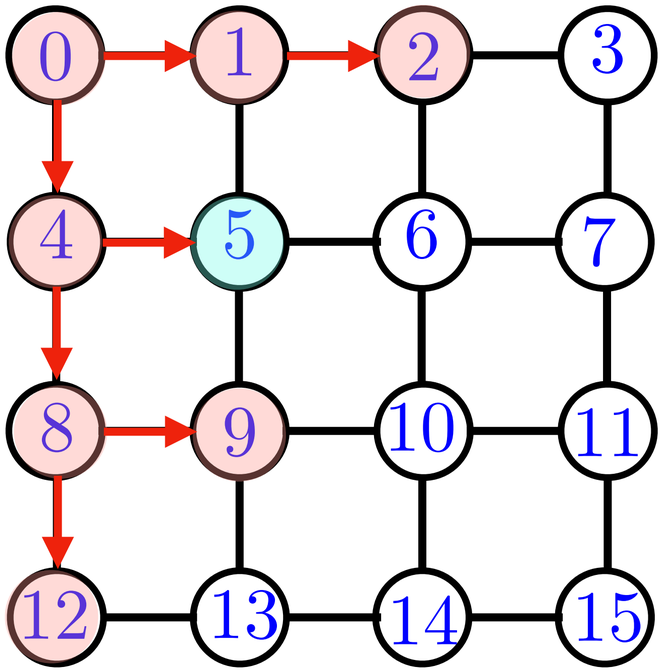
visit_queue

5
2
12
9

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 8 next.

All vertices in the queue are at distance 2 or 3 from vertex 0, with those at distance 2 preceeding those at distance 3.

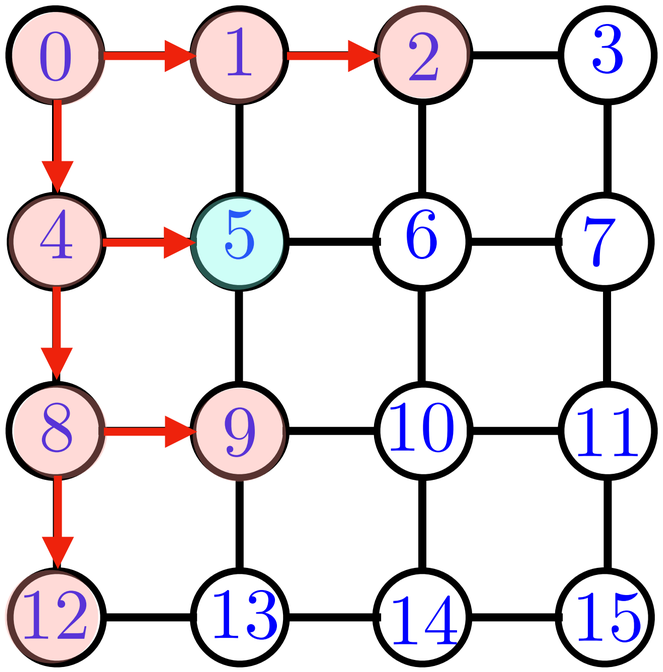


visit_queue

2
12
9

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 5 next.



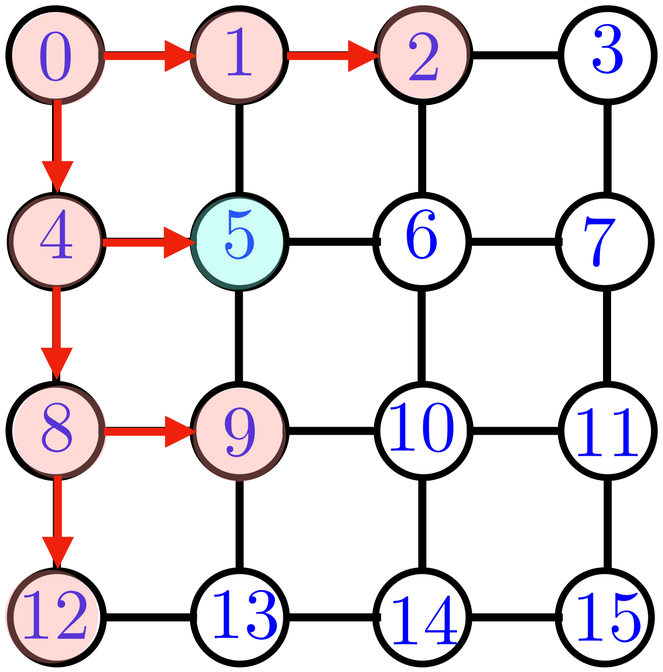
visit_queue

2
12
9

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 5 next.

All vertices in the queue are at distance 2 or 3 from vertex 0, with those at distance 2 preceeding those at distance 3.

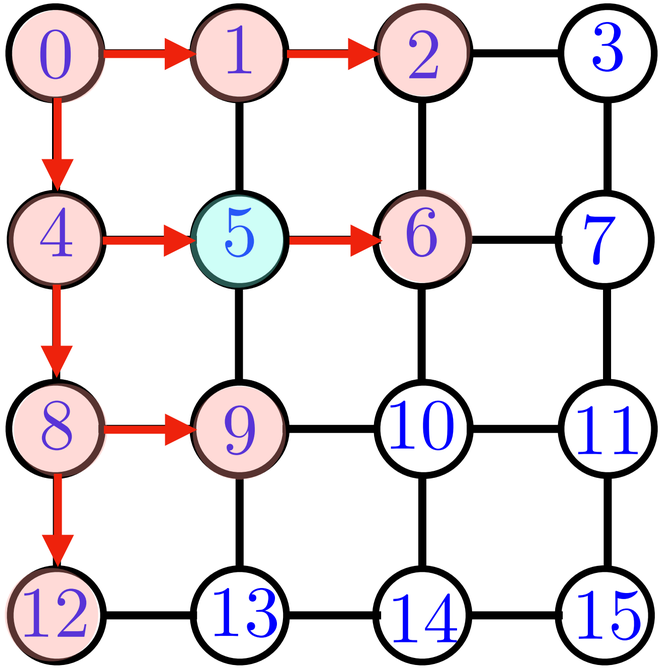


visit_queue
2
12
9
6

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 5 next.

All vertices in the queue are at distance 2 or 3 from vertex 0, with those at distance 2 preceeding those at distance 3.

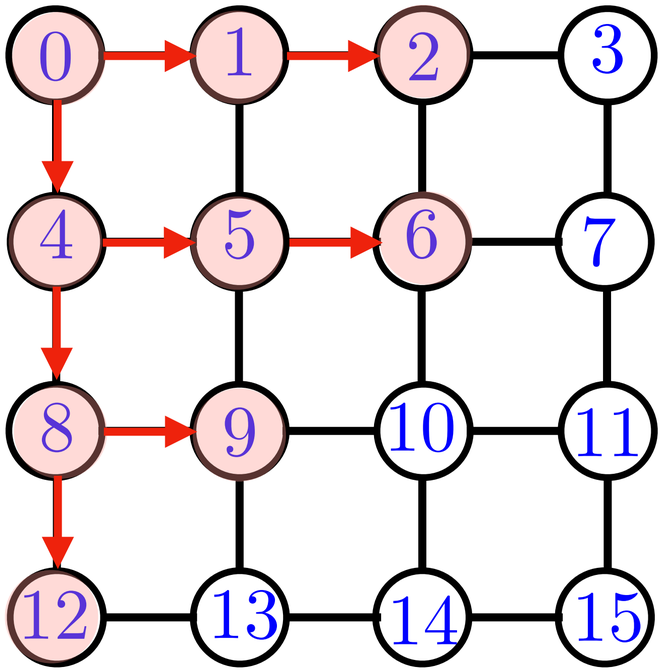


visit_queue
2
12
9
6

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 5 next.

All vertices in the queue are at distance 2 or 3 from vertex 0, with those at distance 2 preceeding those at distance 3.

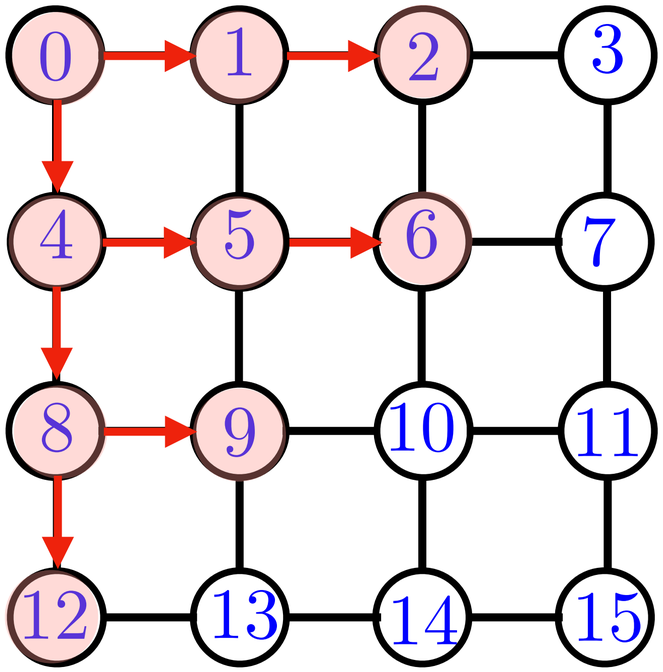


visit_queue

12
9
6

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 2 next.



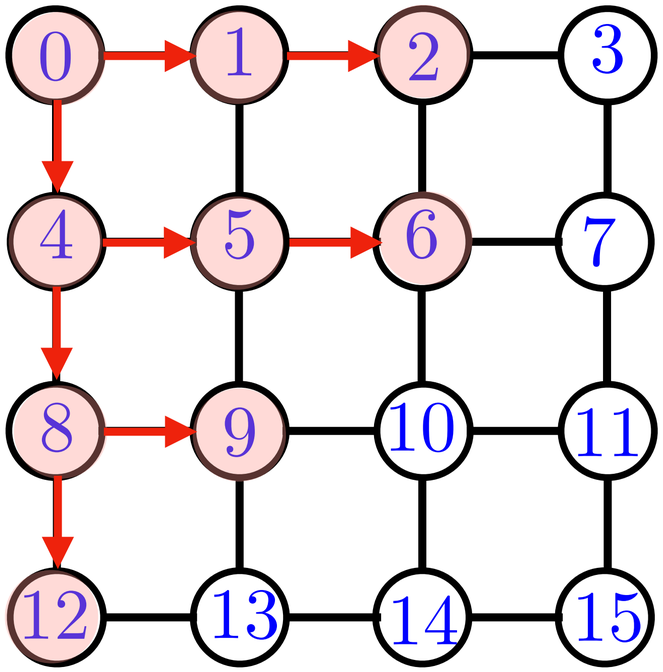
visit_queue

12
9
6

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 2 next.

Now we have found a path of length 3 from vertex 0 to vertex 3.



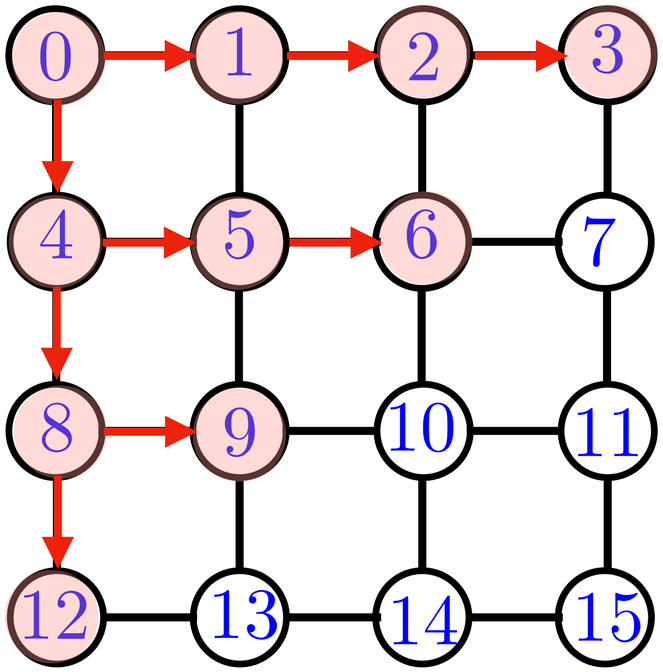
visit_queue

12
9
6
3

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 2 next.

Now we have found a path of length 3 from vertex 0 to vertex 3.



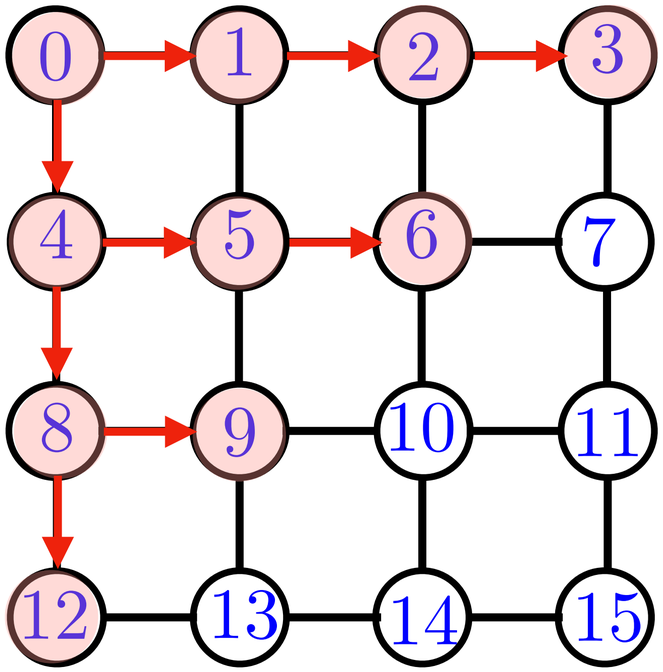
visit_queue

12
9
6
3

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

Explore vertex 2 next.

Now we have found a path of length 3 from vertex 0 to vertex 3.



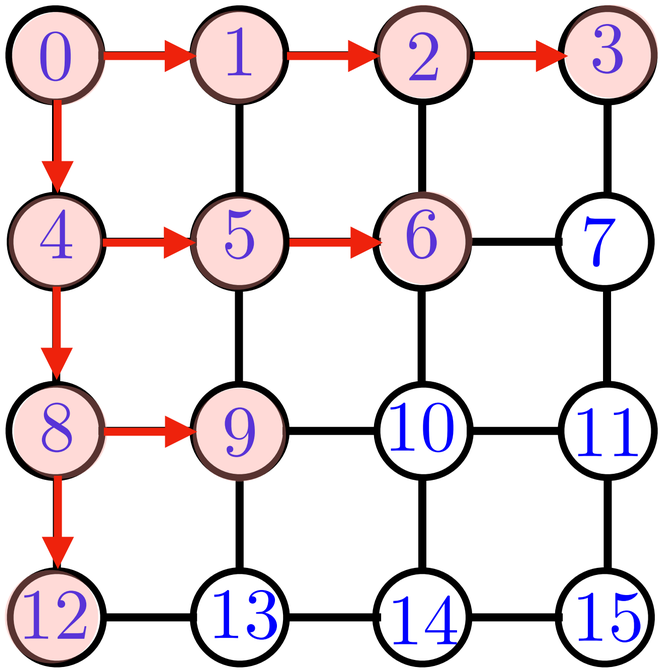
visit_queue

12
9
6
3

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

We can trace the path "backwards" from vertex 3:

3



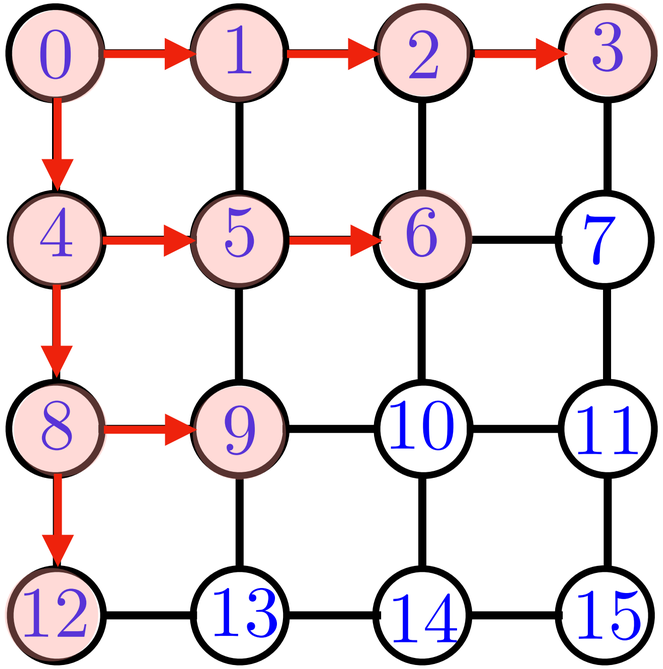
visit_queue

12
9
6
3

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

We can trace the path "backwards" from vertex 3:

3 — edge_to$[3] == 2$



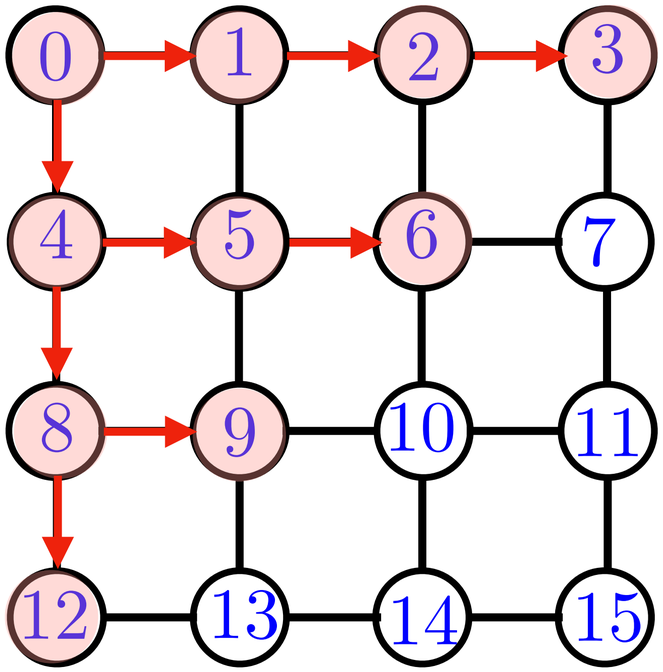
visit_queue

12
9
6
3

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

We can trace the path "backwards" from vertex 3:

$$3 - \texttt{edge\_to}[3] == 2 - \texttt{edge\_to}[2] == 1$$



visit_queue

12
9
6
3

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

We can trace the path "backwards" from vertex 3:

$$3 - \texttt{edge\_to}[3] == 2 - \texttt{edge\_to}[2] == 1 - \texttt{edge\_to}[1] == 0$$

BFS Properties

Say we run BFS starting on vertex v.

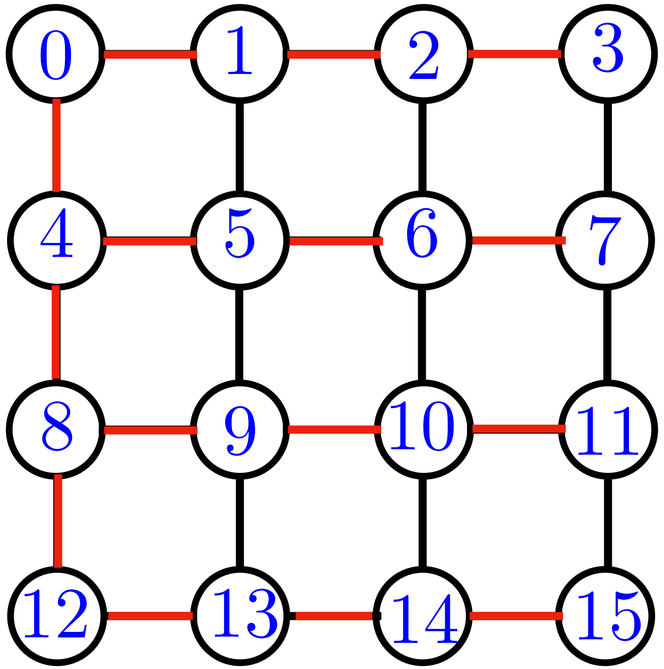
Fact: For any vertex u connected to v, tracing back from $\texttt{edge\_to}[u]$ will give a shortest path from v to u.

As we saw in the example, vertices enter into the queue in order of their distance from v.

A vertex at distance k from v will always enter the queue before any vertex at distance $k+1$.

We will discover u from its neighbor that is closest to v.

BFS Tree



The edges in red represent the `edge_to` array at the end of the BFS algorithm on our example graph.

We have drawn a edge between u and `edge_to`$[u]$ for every vertex $u \neq 0$ (the start vertex).

These edges form a tree!

This is a shortest path tree. The edges in the tree give us shortest paths from vertex 0 to every other vertex in the graph.

BFS Running Time

Each vertex is added to the queue at most once.

When we pop and process a vertex we spend time proportional to its degree.

The total running time is $O(|V| + |E|)$ in the adjacency list model.

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

BFS Running Time

Each vertex is added to the queue at most once.

When we pop and process a vertex we spend time proportional to its degree.

The total running time is $O(|V| + |E|)$ in the adjacency list model.

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

BFS Running Time

Each vertex is added to the queue at most once.

When we pop and process a vertex we spend time proportional to its degree.

The total running time is $O(|V| + |E|)$ in the adjacency list model.

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```

BFS Running Time

Each vertex is added to the queue at most once.

When we pop and process a vertex we spend time proportional to its degree.

The total running time is $O(|V| + |E|)$ in the adjacency list model.

```
while(!visit_queue.empty())
{
    unsigned x = visit_queue.front();
    visit_queue.pop();
    for(auto u : arr[x])
    {
        if(!marked[u])
        {
            visit_queue.push(u);
            marked[u] = true;
            // we came to u from x
            edge_to[u] = x;
        }
    }
}
```