DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 25

1. REPRESENTING GRAPHS: EXAMPLES IN RUST

2. SAMPLE GRAPH ALGORITHMS

3. MODULES

DISCUSSION SECTION TODAY

- Reading input from file
  - You'll be asked to do this on your homework
- Additional examples of using collections

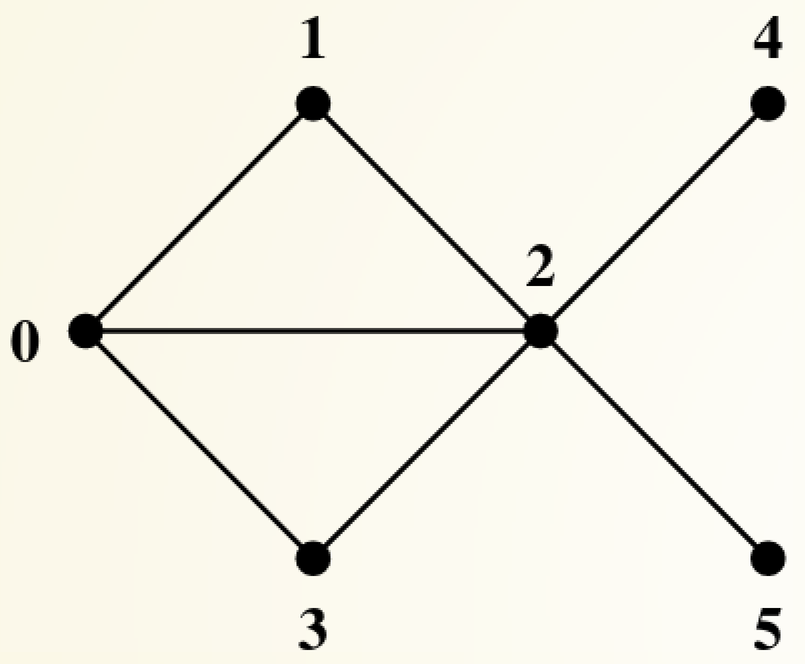
1. REPRESENTING GRAPHS: EXAMPLES IN RUST

2. SAMPLE GRAPH ALGORITHMS

3. MODULES

SAMPLE GRAPH

Sample graph from the previous lecture:

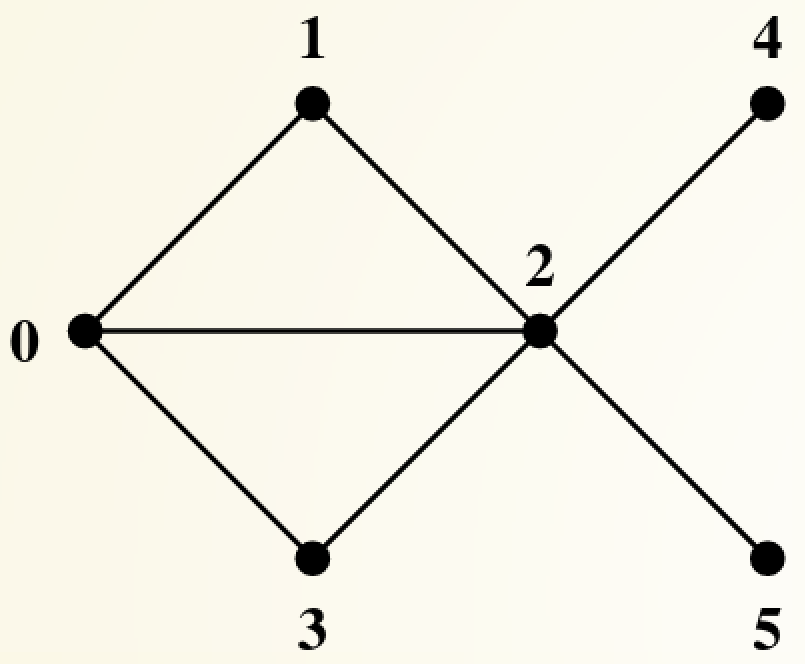


This lecture's graphs:

- undirected
- no self-loops
  - self-loop: edge connecting a vertex to itself
- no parallel edges (connecting the same pair of vertices)

SAMPLE GRAPH

Sample graph from the previous lecture:



This lecture's graphs:

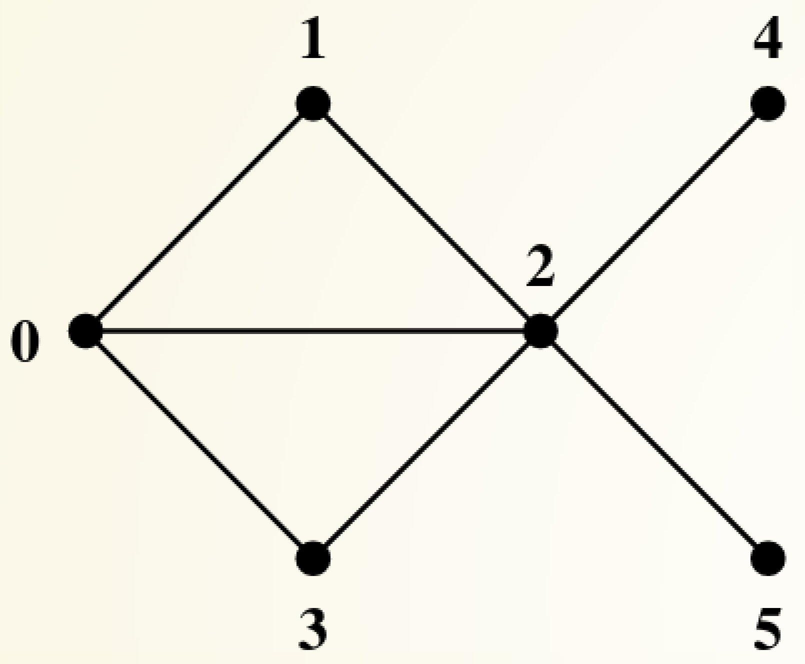
- undirected
- no self-loops
  - self-loop: edge connecting a vertex to itself
- no parallel edges (connecting the same pair of vertices)

Simplifying assumption:

- n vertices labeled $0 \ldots n-1$

SAMPLE GRAPH

Sample graph from the previous lecture:



This lecture's graphs:

- undirected
- no self-loops
  - self-loop: edge connecting a vertex to itself
- no parallel edges (connecting the same pair of vertices)

Simplifying assumption:

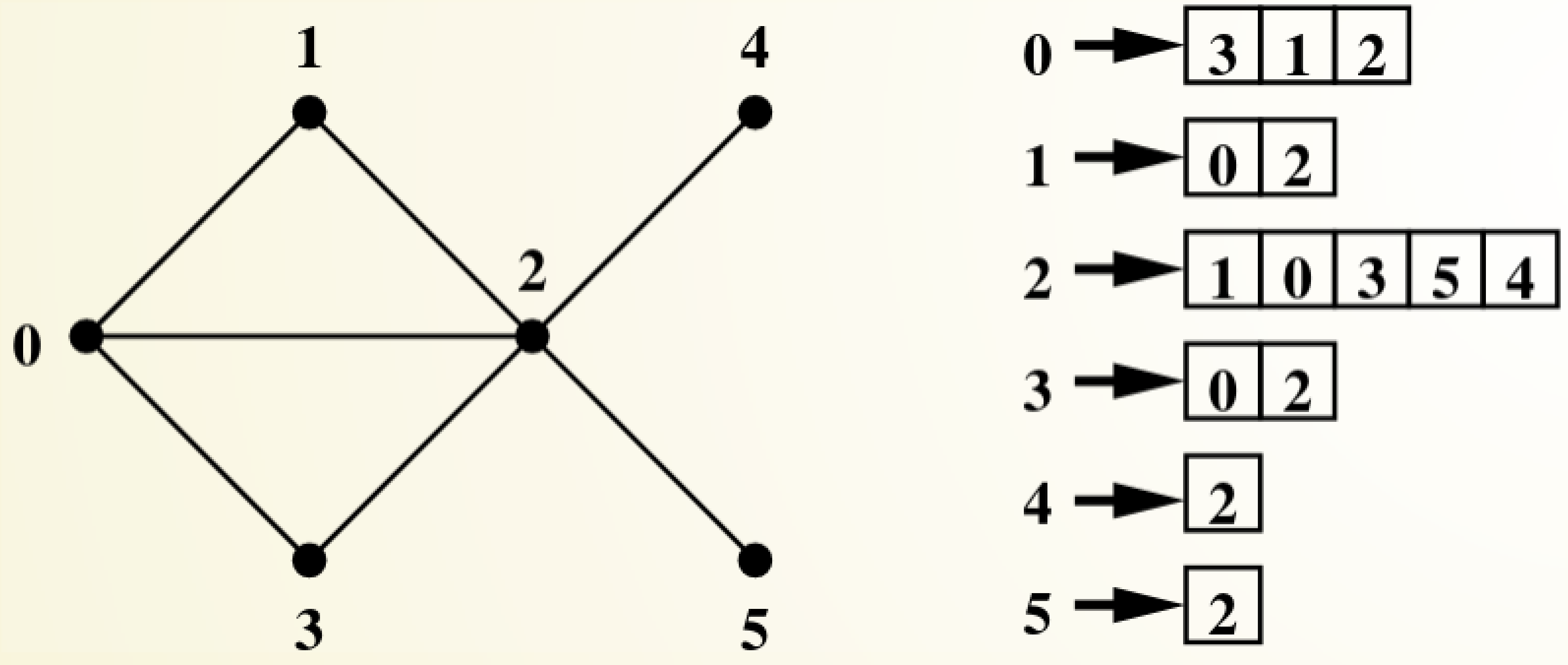
- n vertices labeled $0 \dots n-1$

In [2]:
```
// number of vertices
let n : usize = 6;

// list of edges
let edges : Vec<(usize,usize)> = vec![(0,1), (1,2), (2,3), (3,0), (2,0), (2,4), (2,5)];
```

ADJACENCY LIST REPRESENTATION

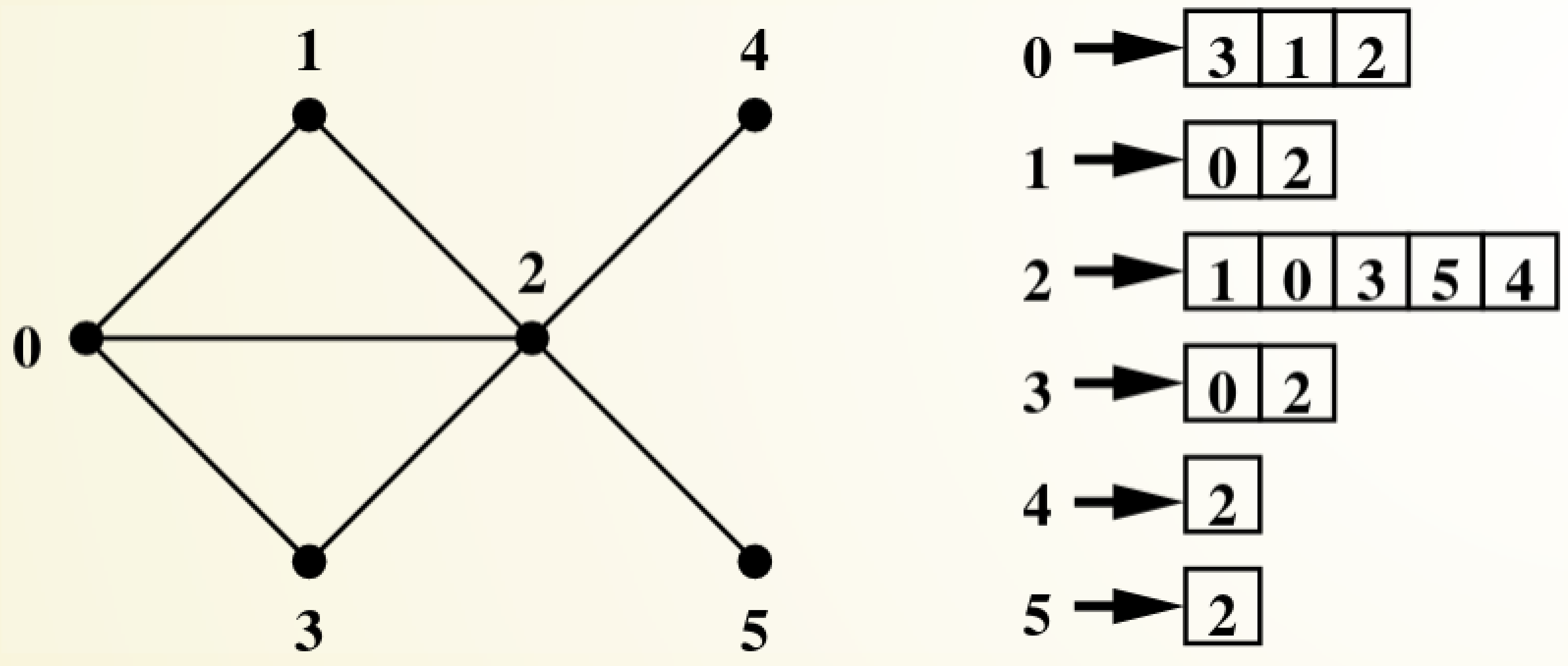
List of neighbors for each vertex



ADJACENCY LIST REPRESENTATION

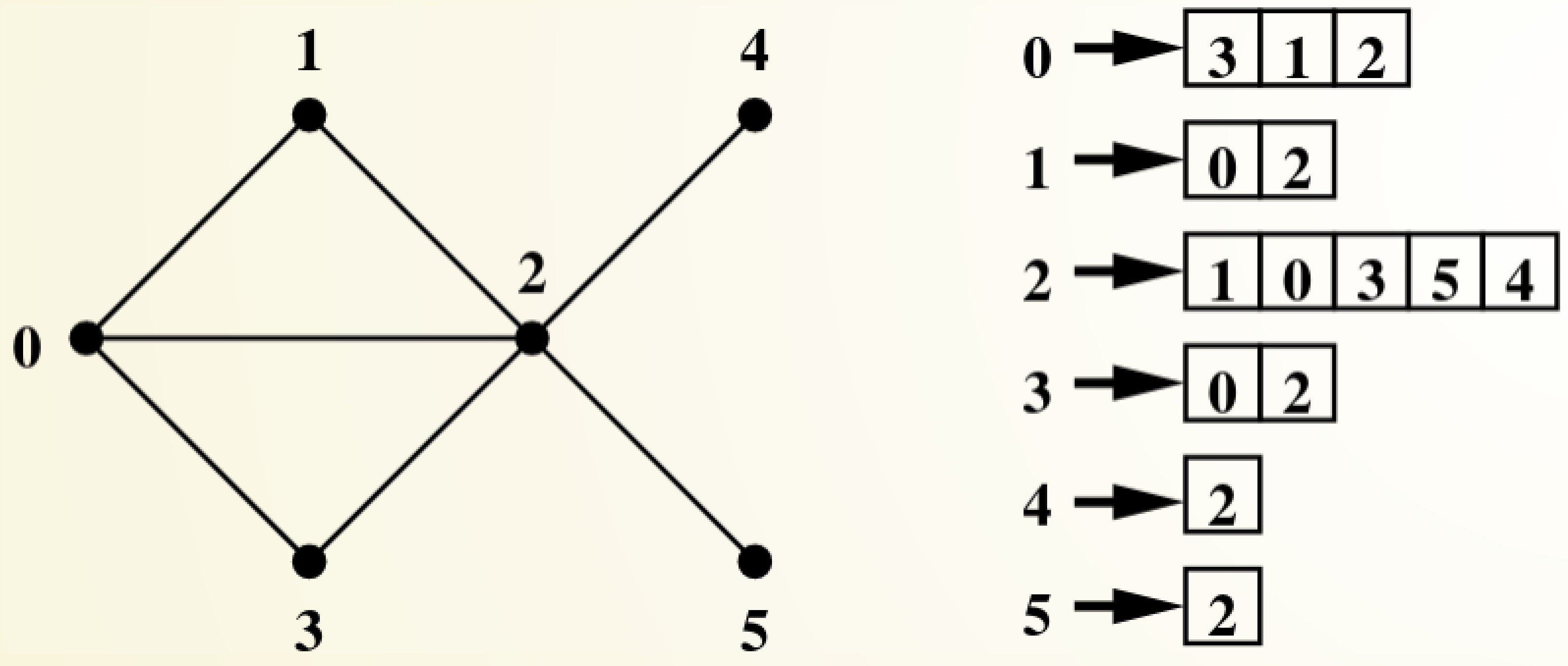
List of neighbors for each vertex

```
In [3]: let mut graph_list : Vec<Vec<usize>> = vec![vec![];n];
```



ADJACENCY LIST REPRESENTATION

List of neighbors for each vertex

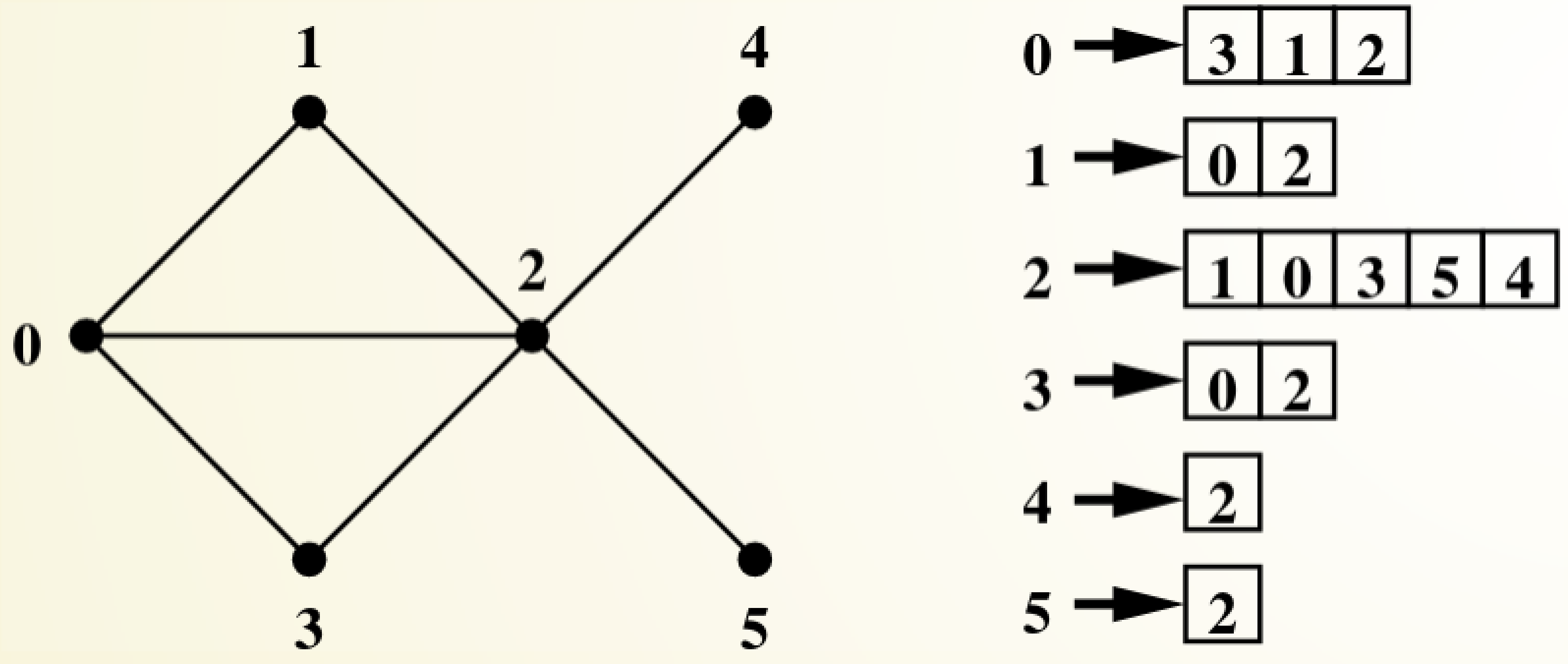


In [3]:
```
let mut graph_list : Vec<Vec<usize>> = vec![vec![];n];
```

In [4]:
```
for (v,w) in edges.iter() {
    graph_list[*v].push(*w);
    graph_list[*w].push(*v);
};
```

ADJACENCY LIST REPRESENTATION

List of neighbors for each vertex



In [3]:
```
let mut graph_list : Vec<Vec<usize>> = vec![vec![];n];
```

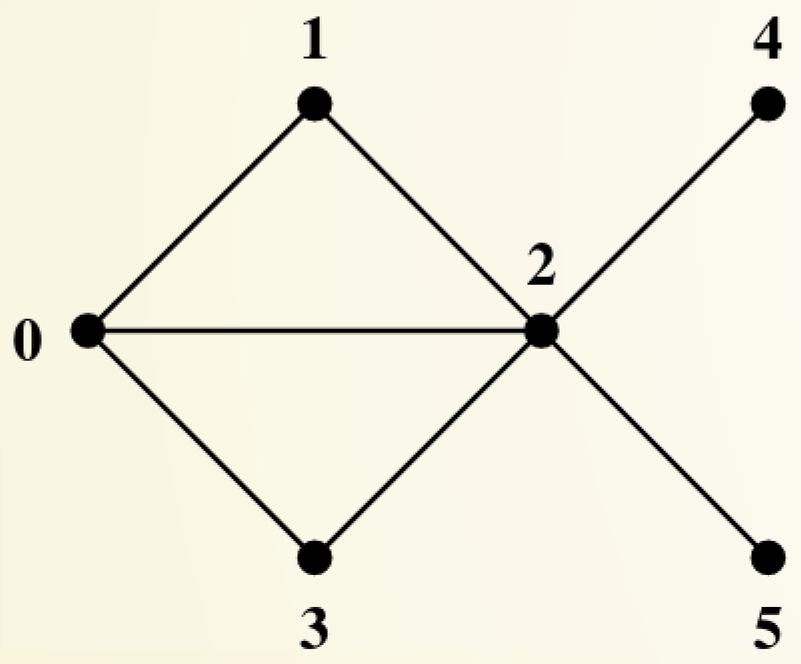
In [4]:
```
for (v,w) in edges.iter() {
    graph_list[*v].push(*w);
    graph_list[*w].push(*v);
};
```

In [5]:
```
for i in 0..graph_list.len() {
    println!("{}: {:?}", i, graph_list[i]);
};
```

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

ADJACENCY MATRIX REPRESENTATION

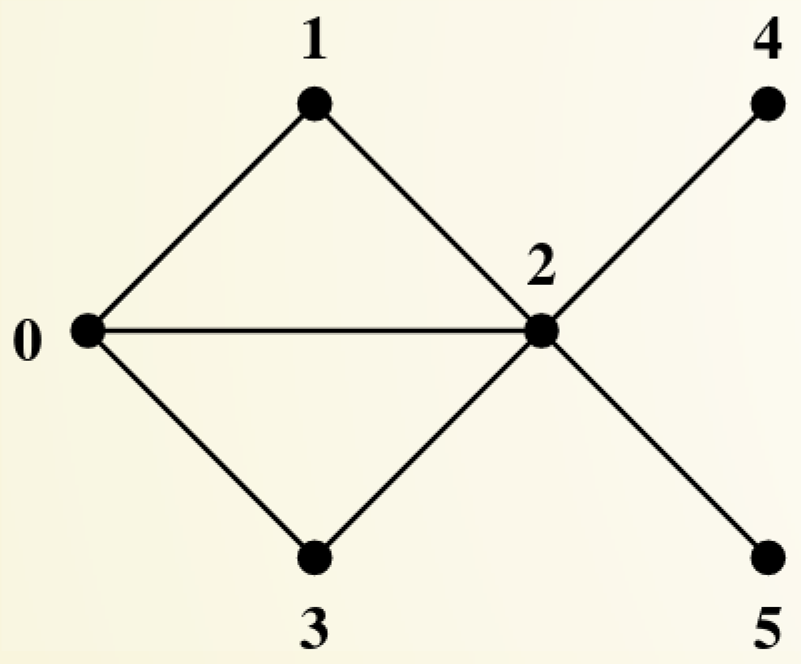
Matrix of Boolean values



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

ADJACENCY MATRIX REPRESENTATION

Matrix of Boolean values

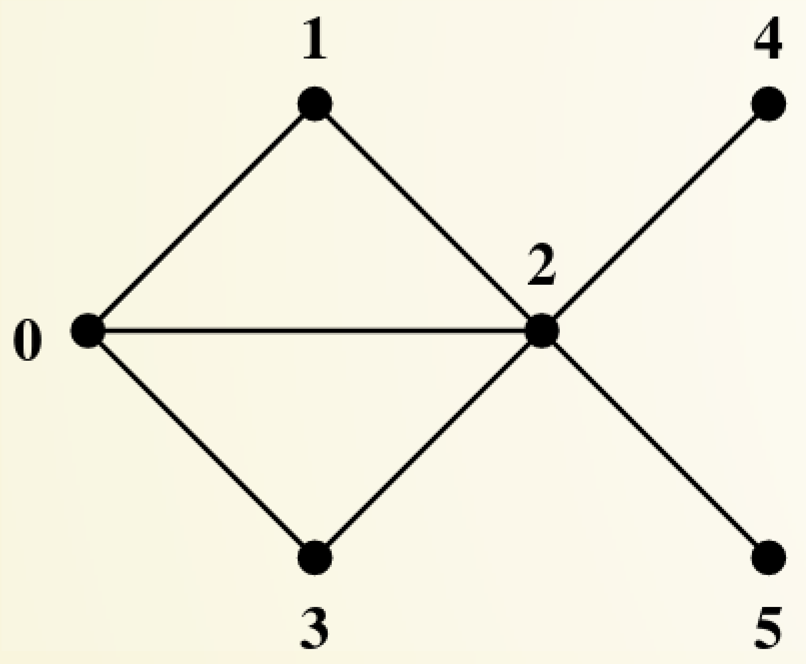


	0	1	2	3	4	5
0		1	1	1	0	0
1	1		1	0	0	0
2	1	1		1	1	1
3	1	0	1		0	0
4	0	0	1	0		0
5	0	0	1	0	0	

In [6]:
```
let mut graph_matrix = vec![vec![false;n];n];
```

ADJACENCY MATRIX REPRESENTATION

Matrix of Boolean values



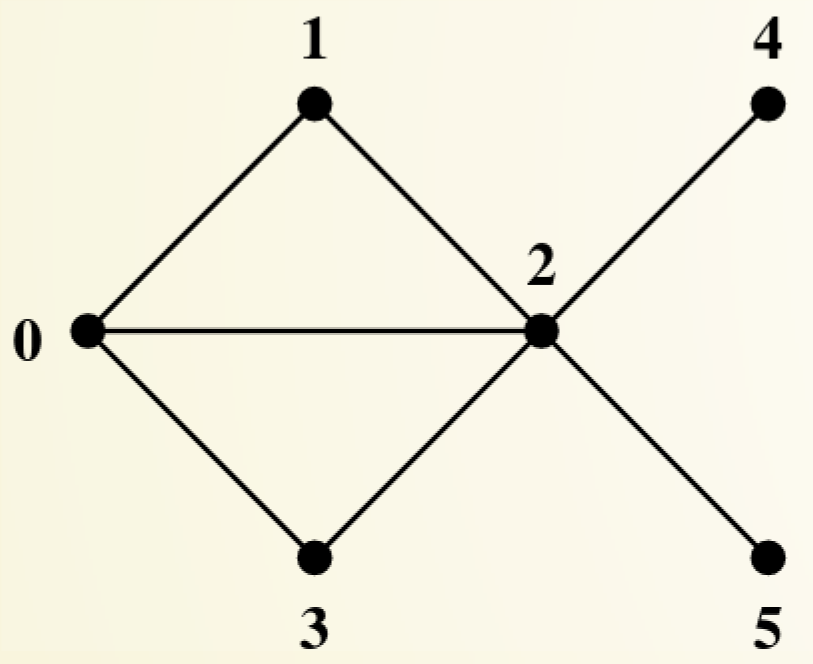
	0	1	2	3	4	5
0		1	1	1	0	0
1	1		1	0	0	0
2	1	1		1	1	1
3	1	0	1		0	0
4	0	0	1	0		0
5	0	0	1	0	0	

In [6]:
```
let mut graph_matrix = vec![vec![false;n];n];
```

In [7]:
```
for (v,w) in edges.iter() {
    graph_matrix[*v][*w] = true;
    graph_matrix[*w][*v] = true;
};
```

ADJACENCY MATRIX REPRESENTATION

Matrix of Boolean values



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

In [6]:
```
let mut graph_matrix = vec![vec![false;n];n];
```

In [7]:
```
for (v,w) in edges.iter() {
    graph_matrix[*v][*w] = true;
    graph_matrix[*w][*v] = true;
};
```

In [8]:
```
for row in &graph_matrix {
    for entry in row.iter() {
        print!("{}",if *entry {"1"} else {"0"});
    }
    println!("");
};
```

```
011100
101000
110111
101000
001000
001000
```

WHAT IF LABELS ARE NOT IN $\{0, 1, \dots n-1\}$?

`T` = type of labels

WHAT IF LABELS ARE NOT IN $\{0, 1, \ldots n - 1\}$?

`T` = type of labels

Solution 1: Map everything to this range

- Create hash maps from input labels to $\{0, 1, \ldots n - 1\}$
- Create a reverse hash map to recover labels when needed

WHAT IF LABELS ARE NOT IN $\{0, 1, \ldots n-1\}$?

`T` $=$ type of labels

Solution 1: Map everything to this range

- Create hash maps from input labels to $\{0, 1, \ldots n-1\}$
- Create a reverse hash map to recover labels when needed

Solution 2: Replace with hash maps and hash sets

- Adjacency lists: use `HashMap<T,Vec<T>>`
- Adjacency matrix: use `HashSet<(T,T)>`
- Bonus gain: `HashSet<(T,T)>` better than adjacency matrix for sparse graphs

WHAT IF THE GRAPH IS DIRECTED?

WHAT IF THE GRAPH IS DIRECTED?

Adjacency lists:

- separate lists incoming/outgoing edges
- depends on what information needed for your algorithm

WHAT IF THE GRAPH IS DIRECTED?

Adjacency lists:

- separate lists incoming/outgoing edges
- depends on what information needed for your algorithm

Adjacency matrix:

- example: edge $u \rightarrow v$ and no edge in the opposite direction:
  - `matrix[u][v] = true`
  - `matrix[v][u] = false`

1. REPRESENTING GRAPHS: EXAMPLES IN RUST

2. SAMPLE GRAPH ALGORITHMS

3. MODULES

COUNT TRIANGLES

Problem to solve: Consider all triples of vertices. What is the number of those in which all vertices are connected?

COUNT TRIANGLES

Problem to solve: Consider all triples of vertices. What is the number of those in which all vertices are connected?

Solution 1: Enumerate explicitly over all triples and check which are triangles, using the adjacency matrix

In [9]:
```
let mut count: u32 = 0;
for u in 0..n {
    for v in u+1..n {
        for w in v+1..n {
            if (graph_matrix[u][v] && graph_matrix[v][w] && graph_matrix[u][w]) {
                count += 1;
            }
        }
    }
}
count
```

Out[9]: 2

COUNT TRIANGLES

Problem to solve: Consider all triples of vertices. What is the number of those in which all vertices are connected?

Solution 2: Follow links from each vertex to see if you come back in three steps

In [10]:
```
let mut count: u32 = 0;
for u in 0..n {
    for v in &graph_list[u] {
        for w in &graph_list[*v] {
            for u2 in &graph_list[*w] {
                if u == *u2 {
                    count += 1;
                }
            }
        }
    }
}
count
```

Out[10]:
```
12
```

COUNT TRIANGLES

Problem to solve: Consider all triples of vertices. What is the number of those in which all vertices are connected?

Solution 2: Follow links from each vertex to see if you come back in three steps

In [10]:
```
let mut count: u32 = 0;
for u in 0..n {
    for v in &graph_list[u] {
        for w in &graph_list[*v] {
            for u2 in &graph_list[*w] {
                if u == *u2 {
                    count += 1;
                }
            }
        }
    }
}
count
```

Out[10]: 12

In [11]:
```
// need to divide by 6
// due to symmetries triangles counted multiple times
count / 6
```

Out[11]: 2

COUNT TRIANGLES

Problem to solve: Consider all triples of vertices. What is the number of those in which all vertices are connected?

Different implementation of solution 2

In [12]:
```
fn walk(current:usize,destination:usize,steps:usize,adjacency_list:&Vec<Vec<usize>>) -> u32 {
    match steps {
        0 => if current == destination {1} else {0},
        _ => {
            let mut count = 0;
            for v in &adjacency_list[current] {
                count += walk(*v,destination,steps-1,adjacency_list);
            }
            count
        }
    }
}
```

COUNT TRIANGLES

Problem to solve: Consider all triples of vertices. What is the number of those in which all vertices are connected?

Different implementation of solution 2

In [12]:
```
fn walk(current:usize,destination:usize,steps:usize,adjacency_list:&Vec<Vec<usize>>) -> u32 {
    match steps {
        0 => if current == destination {1} else {0},
        _ => {
            let mut count = 0;
            for v in &adjacency_list[current] {
                count += walk(*v,destination,steps-1,adjacency_list);
            }
            count
        }
    }
}
```

In [13]:
```
let mut count = 0;
for v in 0..n {
    count += walk(v,v,3,&graph_list);
}
count / 6
```

Out[13]: 2

COUNT TRIANGLES

Problem to solve: Consider all triples of vertices. What is the number of those in which all vertices are connected?

Solution 3: For each vertex try all pairs of neighbors (via adjacency lists) and see if they are connected (via adjacency matrix)

In [14]:
```
let mut count: u32 = 0;
for u in 0..n {
    let neighbors = &graph_list[u];
    for v in neighbors {
        for u in neighbors {
            if graph_matrix[*v][*u] {
                count += 1;
            }
        }
    }
}
count / 6
```

Out[14]: 2

1. REPRESENTING GRAPHS: EXAMPLES IN RUST

2. SAMPLE GRAPH ALGORITHMS

3. MODULES

MODULES

Up to now: **our** functions and data types (mostly) in the same namespace

- **exception:** functions in structs and enums

MODULES

Up to now: **our** functions and data types (mostly) in the same namespace

- **exception:** functions in structs and enums

One can create a namespace, using `mod`

In [15]:
```
mod things_to_say {
    fn say_hi() {
        say("Hi");
    }

    fn say_bye() {
        say("Bye");
    }

    fn say(what: &str) {
        println!("{}!",what);
    }
}
```

MODULES

Up to now: **our** functions and data types (mostly) in the same namespace

- **exception:** functions in structs and enums

One can create a namespace, using `mod`

In [15]:
```
mod things_to_say {
    fn say_hi() {
        say("Hi");
    }

    fn say_bye() {
        say("Bye");
    }

    fn say(what: &str) {
        println!("{}!",what);
    }
}
```

You have to use the module name to refer to access a function.

In [16]:
```
things_to_say::say_hi();
```

```
things_to_say::say_hi();
               ^^^^^^ private function
function `say_hi` is private
```

MODULES

- By default, all definitions in the namespace are private.
- Advantage: Can hide all internally used code
- Use `pub` to make functions or types public

In [17]:
```
mod things_to_say {
    pub fn say_hi() {
        say("Hi");
    }

    pub fn say_bye() {
        say("Bye");
    }

    fn say(what: &str) {
        println!("{}!",what);
    }
}
```

In [18]:
```
things_to_say::say_hi();
```

```
Hi!
```

TO BE CONTINUED...