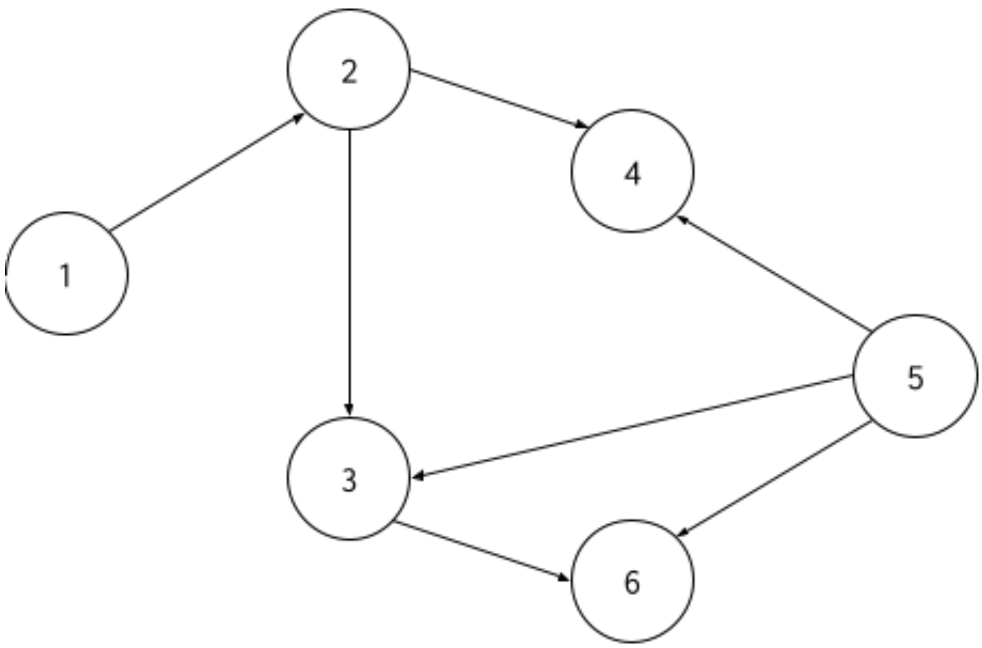
ICC4101 - Algorithms and Competitive Programing

Javier Correa

Graphs

A graph is a structure composed of nodes/vertexes (V) which are related between each other with edged ($(u, v) \in E$):

$$G = (V, E).$$



We call the graph G undirected if $(u, v) \in E \Leftrightarrow (v, u) \in E$, i.e. the order of the vertexes defining an edge **doesn't matter**.

If the order is important, the graph is directed.

Additionally, vertexes or edges could have weights associated:

$$w_v = f(v), w_e = f((u, v))$$

Working with graphs

In general, we need to traverse the graph:

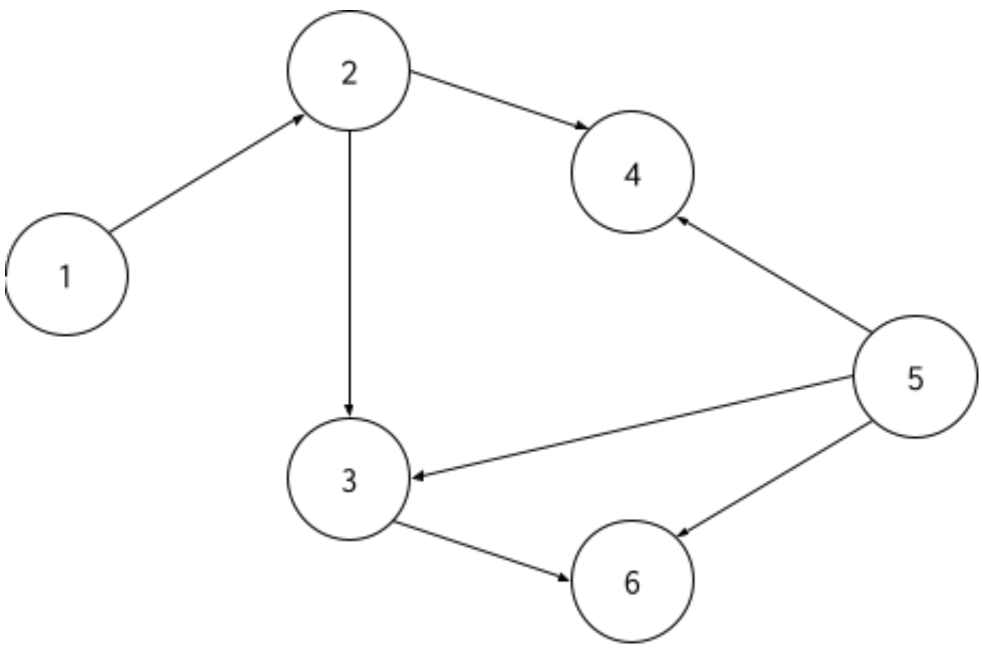
- Iterate nodes
- Given a node, iterate outgoing/incoming edges
- Given an edge, extract the associated nodes

Adjacency Matrix

A possible representation of a graph is using an Adjacency Matrix. A graph is represented with a matrix M such as, if nodes v_i and v_j are connected, e.g. $(v_i, v_j) \in V$, then $M[i,j] \neq 0$, otherwise $M[i,j] = 0$.

- For an undirected graph, the matrix M is symmetric.
- Edge weights can be stored in entries of the adjacency matrix.

For example the graph



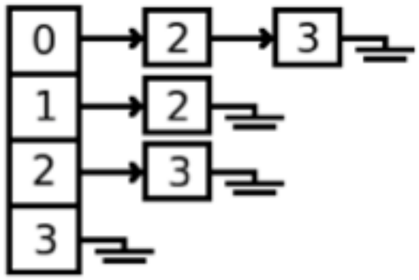

```
G = [
    [0, 1, 0, 0, 0, 0],
    [0, 0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 1],
    [0, 0, 0, 0, 0, 0],
]
```

Adjacency list

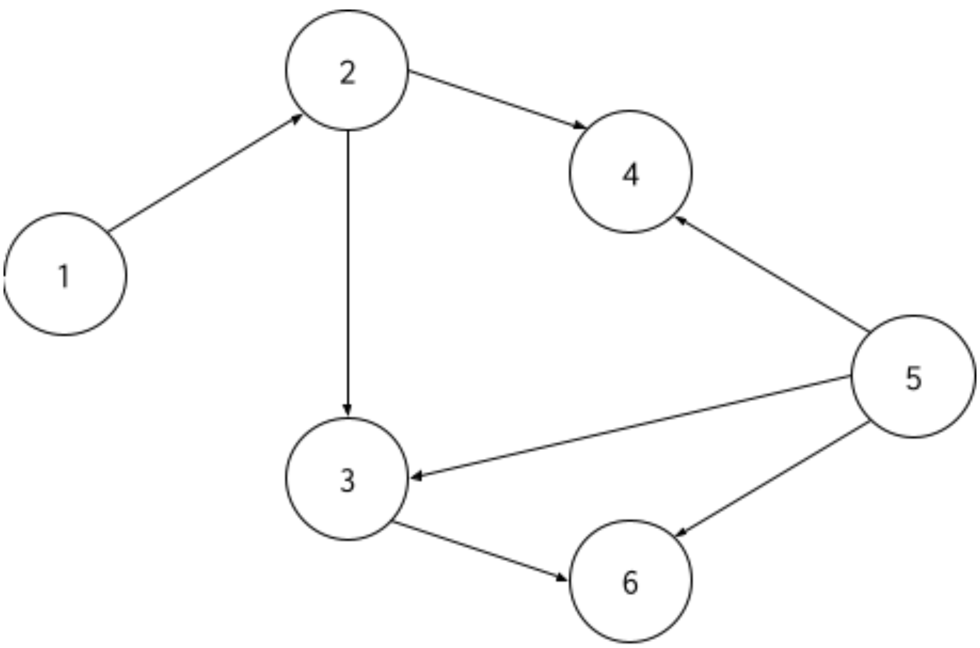
A different representation of a graph is by adjacency lists.

- An array of lists $L[]$, with the number of elements equal the number of nodes in the graph
- The ith entry of the list corresponds to a list of all node j such that $(v_i, v_j) \in V$

$$L[i] = list(\dots, j, \dots) \Rightarrow (v_i, v_j) \in E$$



For example the graph




```
G = [
    [1],
    [2, 3],
    [5],
    [],
    [2, 5],
    [],
]
```

Or to make it more efficient:

```
G = {
    0: {1},
    1: {2, 3},
    3: {5},
    4: set(),
    5: {2, 5},
    6: set(),
}
```

Adjacency matrix vs Adjacency lists

1. Matrix
   - Check if two vertexes are connected takes $O(1)$ time.
   - Store the graph requires $O(|V|^2)$ memory space (worst case).
2. Lists
   - Check if two nodes are connected takes $O(|V|)$ time.
   - Store the graph requires $O(|V| + |E|)$ memory space.

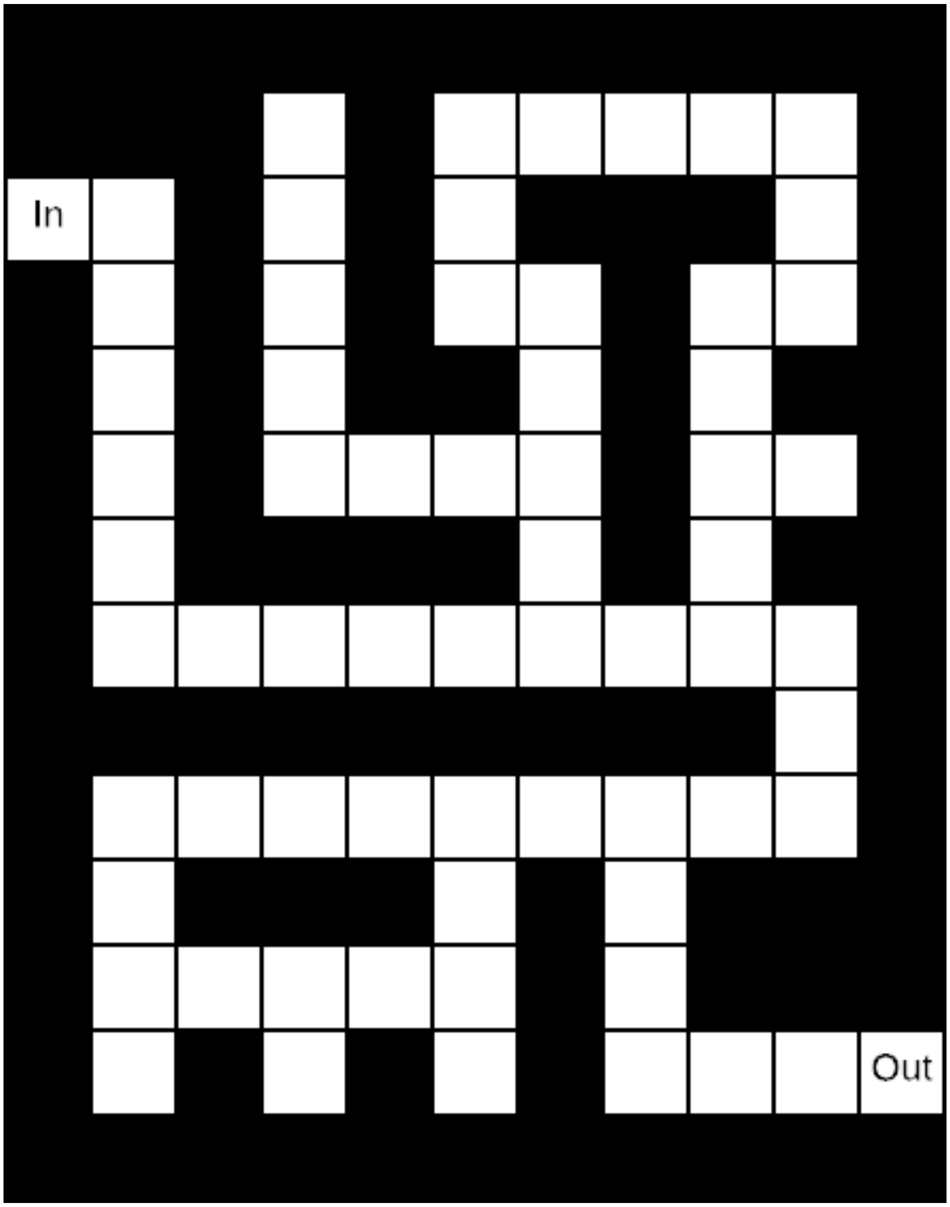
Implicit Graph

In cases where the graph is too big or even infinite, it is convenient to calculate the neighbors of a node "on-the-fly" rather than storing them directly.

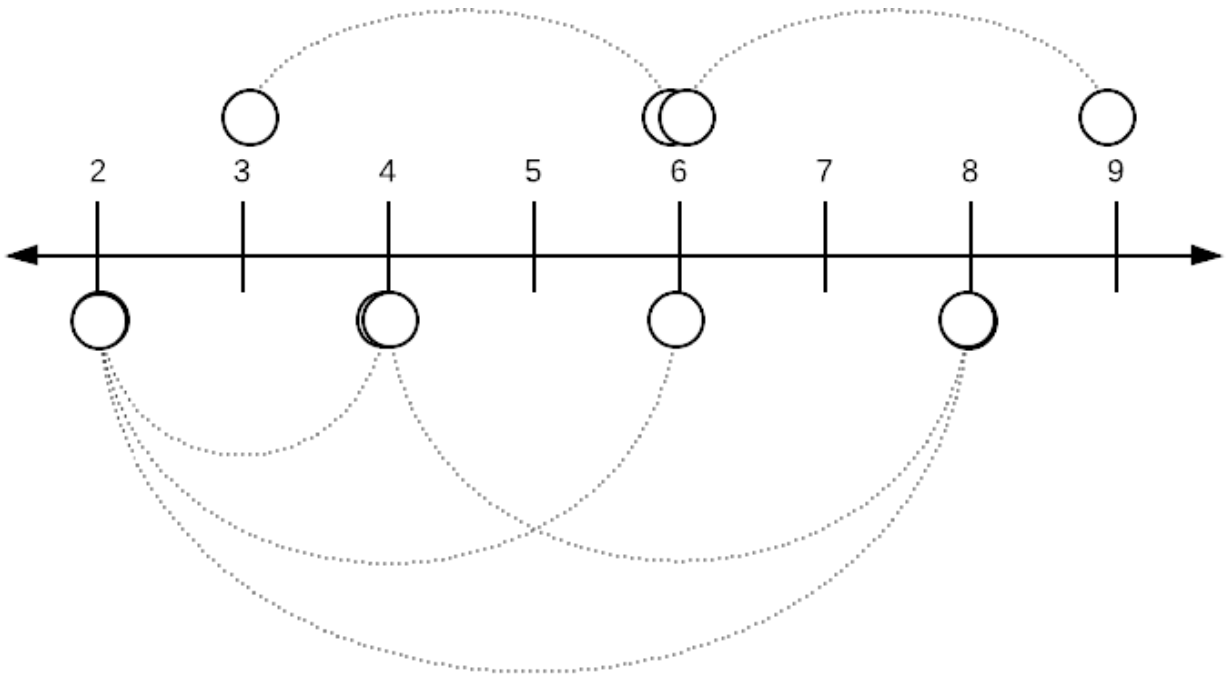
Other case of implicit graph is when the nodes and edges can be calculated following some problem specific rule.

Implicit graph examples

- A 2D maze composed $M \times N$ cells, in which we know if a cell (i, j) is occupied or free.



- Numbers in the an interval $[m, n]$ in which two numbers are related if the are multiples.



Representing a General Graph in Python

```
class node:
    def __init__(self, key, data=None, color=None):
        self.key = key
        self.data = data
        self.color = color

class graph:
    def __init__(self, ...):
        """Initialize the necessary parameters"""
        pass

    def nodes(self):
        """Yield all possible nodes"""
        pass

    def neighbors(self, v):
        """Yield all neighbors starting at node v"""
        for ... in ...:
            do_something(...)
            yield node(k, d, c)
```

Or a `dict`:

```
node = dict(key="...", data=..., color=...)
```

Or just the `key`.

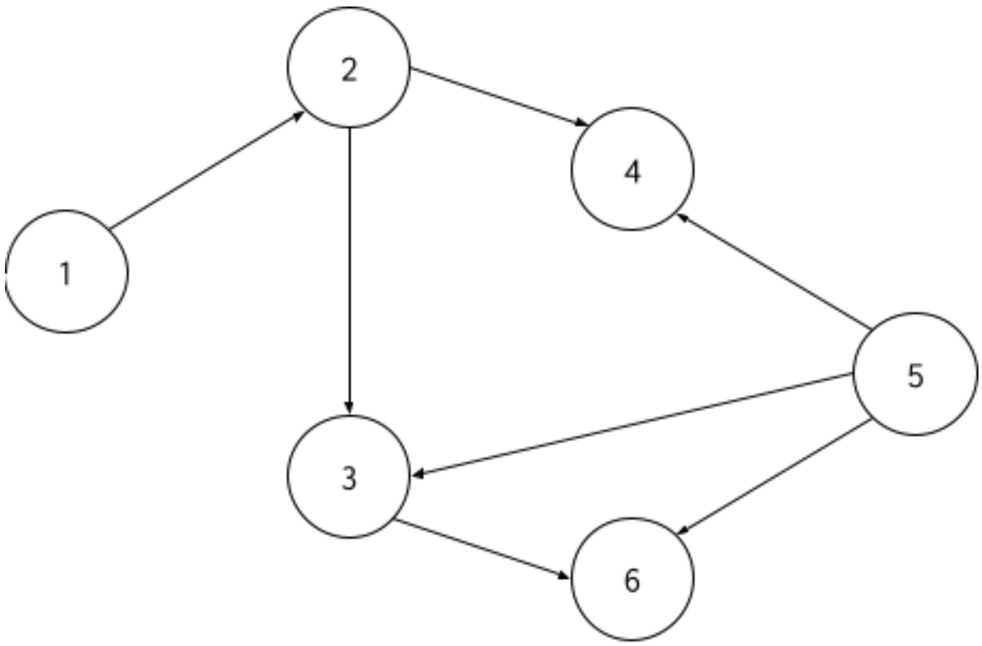
And the graph modeled by a function `neighbors`:

```
def neighbors(v):
    """Yield all neighbors starting at node v"""
    for ... in ...:
        do_something(...)
        yield dict(...)
```

And the properties modeled using `dict`:

```
color = {1: "RED", 2: "BLUE", ...}
```

For example the graph




```
# nodes = 0, 1, 2, 3, 4, 5
def neighbors(v):
    for n in G[v]:
        yield n
```

Graph exploring

Most graph problems are about finding a particular graph or a structure within the graph that has some particular property P.

There are 2 main search strategies:

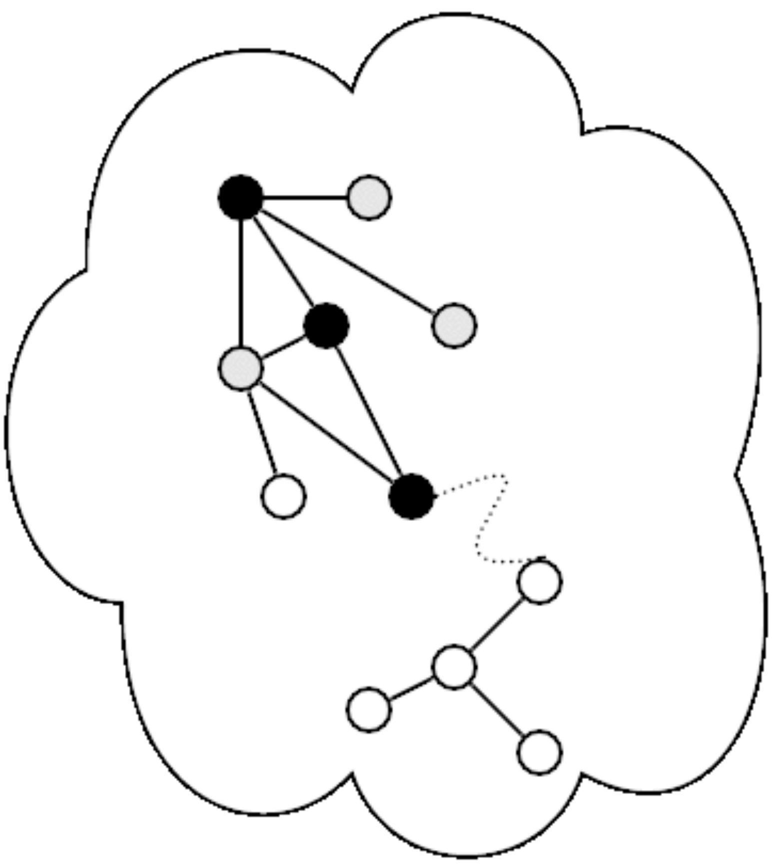
- Deep first search
- Breath first search

And a third for special cases:

- Best first search

Deep First Search (DFS)

Explore the graph, searching for property P first following edges until no further edge is available.



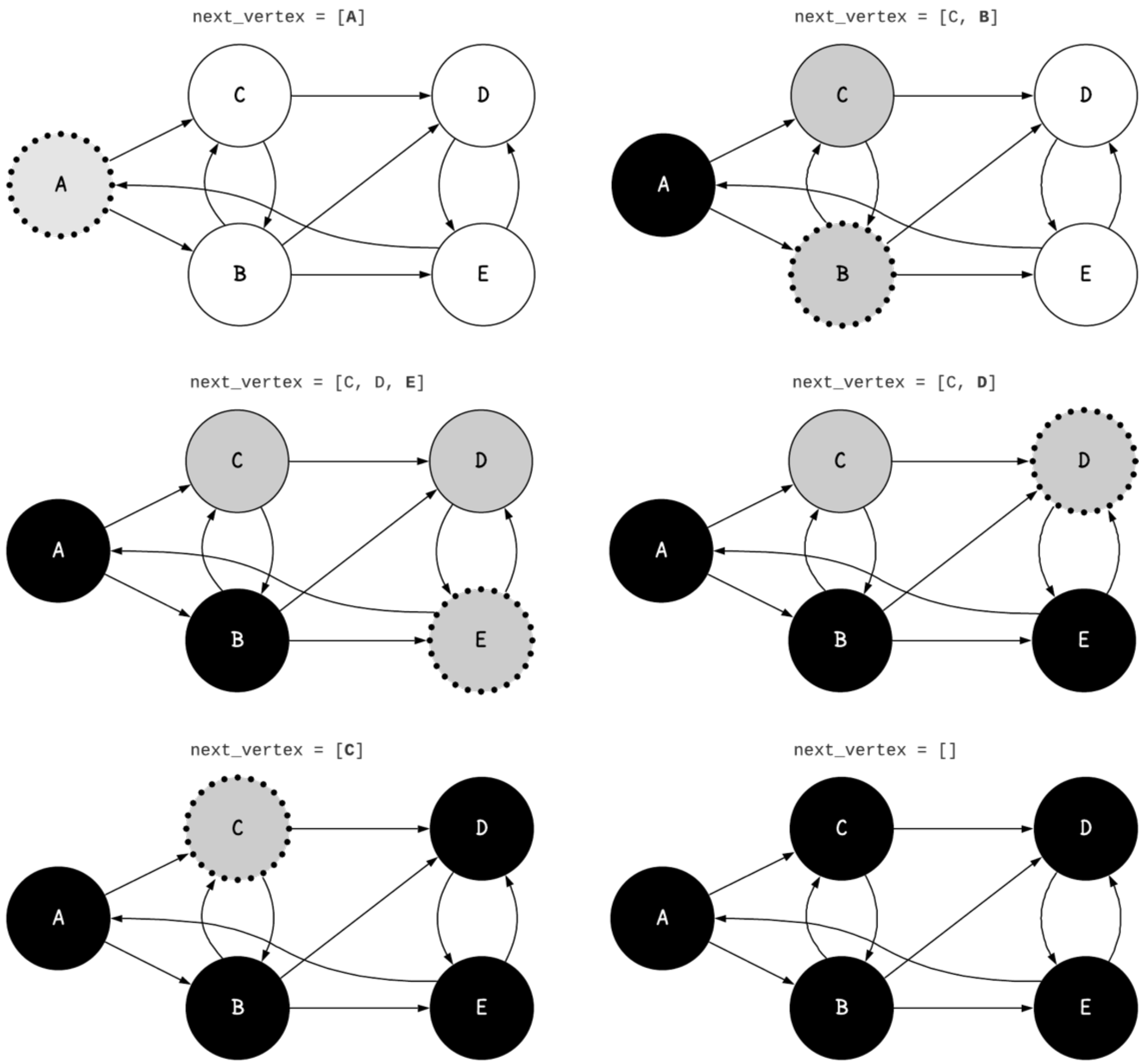
DFS (recursive implementation, Backtracking)

```python
import enum
class color(enum.Enum):
    WHITE = 0
    GREY = 1
    BLACK = 2

def DFS(G, v, P):
    v.color = color.GREY
    for u in G.neighbors(v):
        if u.color == color.WHITE:
            if P(G, u):
                return True # Or "return u" if we are searching for a node
            res = DFS(G, u, P)
            if res:
                return True # Or "return res" if we are searching for a node
    v.color = color.BLACK
    return False # Or "return None" if we are searching for a node
```

DFS (iterative)

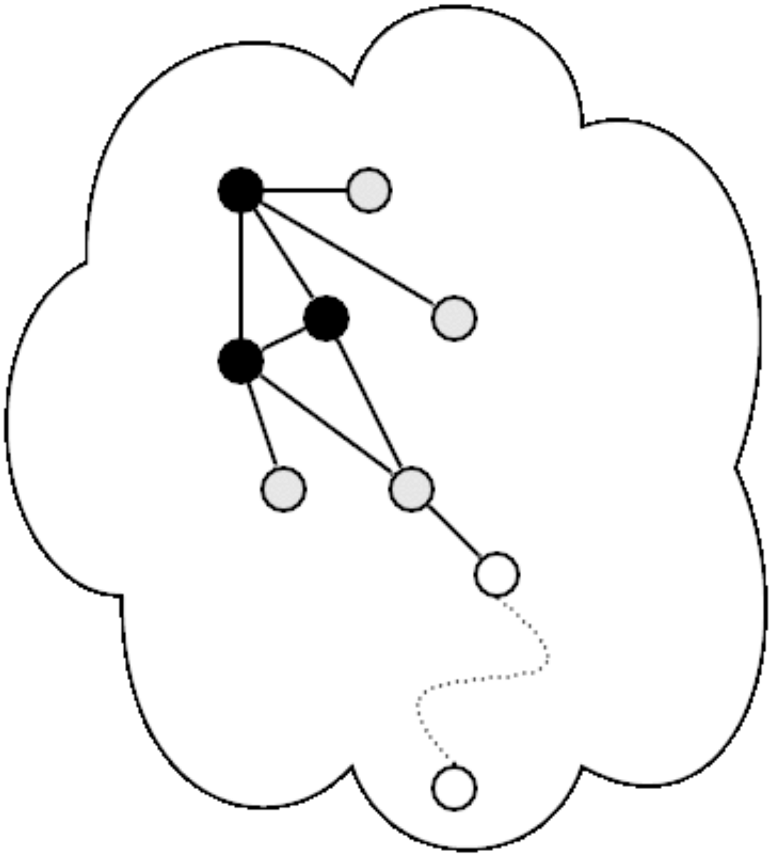
```python
def DFS(G, s, P):
    next_vertexes = [s]
    while len(next_vertexes) > 0:
        v = next_vertexes.pop()
        v.color = color.GREY
        if P(G, v):
            return True # or "return v" if we are searching for a node
        else:
            for u in G.neighbors(v):
                if u.color == color.WHITE:
                    next_vertexes.append(u)
        v.color = color.BLACK
    return False # or "return None" if we are searching for a node
```

Deep First Search Example



Breath first seach

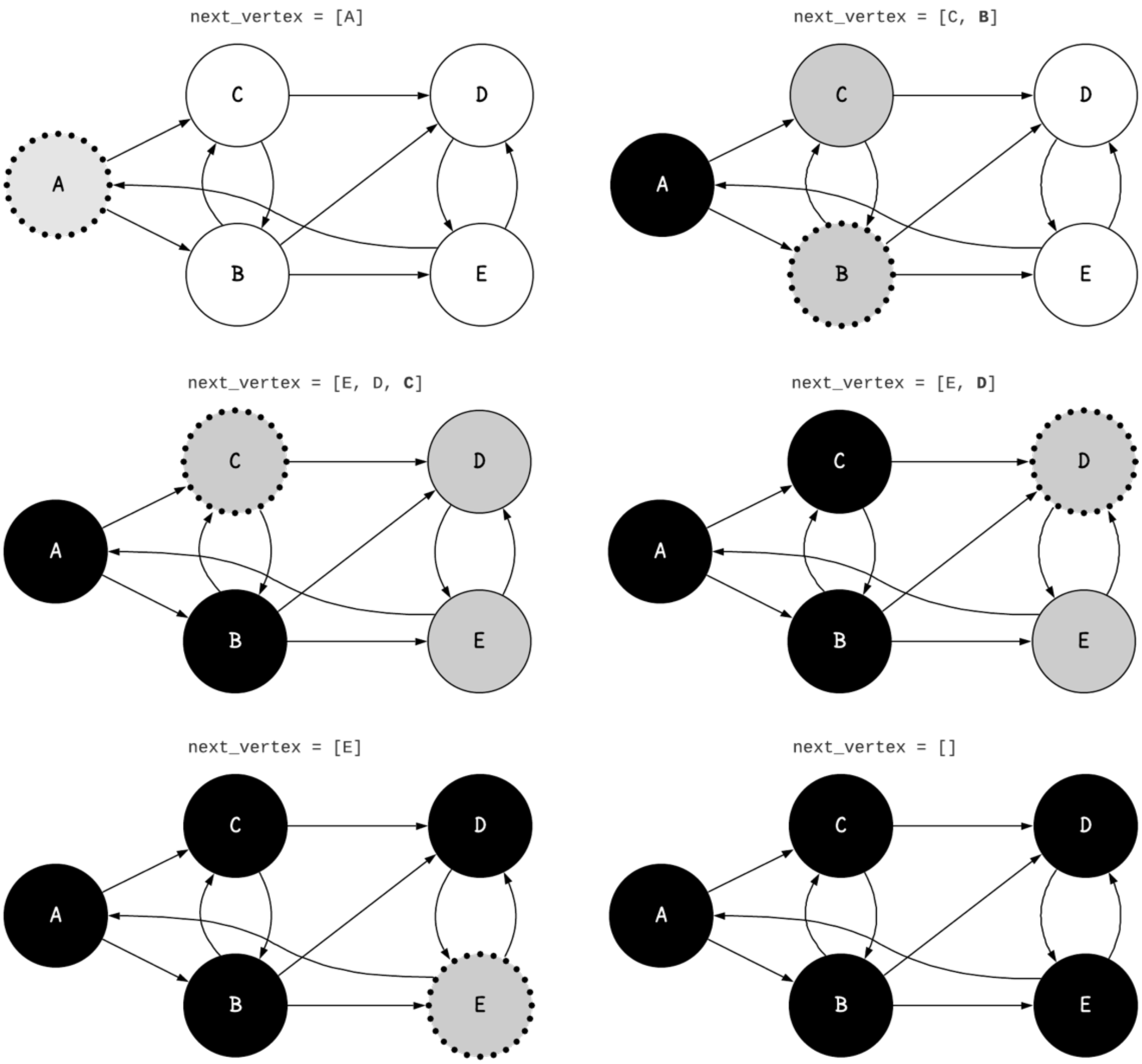
Evaluates all sibling nodes and then expand to their child nodes.



BFS

```
def BFS(G, s, P):
    next_vertexes = [s]
    while len(next_vertexes) > 0:
        v = next_vertexes.pop()
        v.color = color.GREY
        if P(G, v):
            return True # or "return v" if we are searching for a
node
        else:
            for u in G.neighbors(v):
                if u.color == color.WHITE:
                    u.color = color.GREY
                    next_vertexes.prepend(u)
        v.color = color.BLACK
    return False # or "return None" if we are searching for a node
```

Breath First Search Example



Related problems

- Find connected components.
- Flood fill.
- Topological sort.
- Articulation points.
- Strongly connected components.
- Check for a bipartite graph.

Connected components

Run `dfs` or `bfs` from a starting node, if after finishing it's execution white nodes remains, a new components has been detected. Repeat until no white nodes remain.

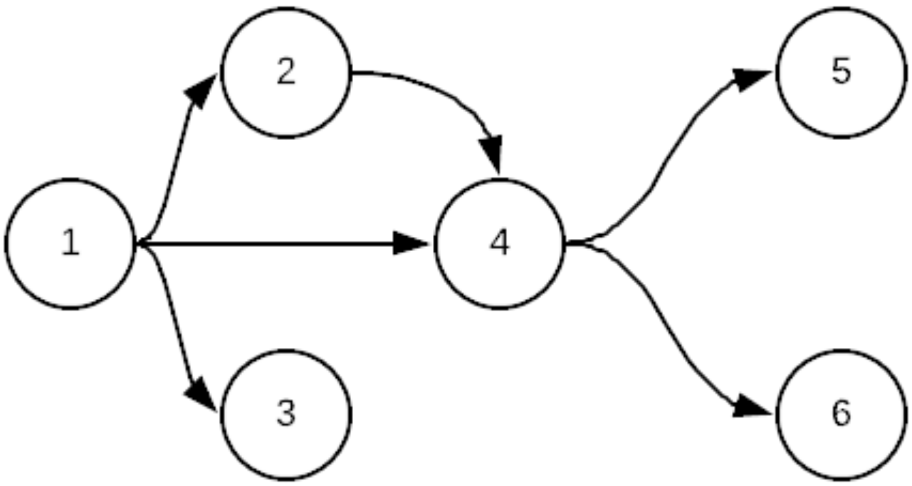
Flood fill, or count the size of the independent components.

Modify the recursive `dfs` to return the size of each component.

Problem to the reader, how to implement it on the iterative `dfs` or `bfs`?

Topological sort.

Given an acyclic directed graph, find some order of the nodes $u_1, u_2, \ldots, u_n$ such that if u_i appears after node u_j in the order, then there is a directed path between $u_i \to u_j$, or u_i and u_j are in independent components of the graph.



Save closed nodes in a "global" list. The topological order is given by reversing this list.

```
def topological_sort_dfs(G, L, v):
    v.color = color.GREY
    for u in G.neighbor(v):
        if u.color == color.WHITE:
            if not topological_sort_dfs(G, L, u):
                return False
        elif u.color == color.BLACK:
             return False
    v.color = color.BLACK
    L.prepend(v)
    return True
```

Articulation points

Simplest algorithm, use `dfs` (or `bfs`) to count for connected components. Then for each vertex `v`, remove the vertex and count the number of CC's. If it increases, then `v` is an articulation point! This method has a computational complexity of $O(V^2 + VE)$, but we can do better!

Using `dfs`:

- For every node `v`, track `num(v)` as the iteration number when the node `v` is first visited.
- For every node `v`, track `low(v)` as the lowest `num(u)` reachable from the exploration starting at `v`, not taking into account the parent of `v`!
- If at the end of the `dfs`, when we are at node `u` with neightbour `v`, if `low(v) ≥ num(u)` the `u` is an articulated point.
- There's a problem with this algorith, the starting point need to fullfil that is has more than one children in the spanning DFS tree.
- Similarly, the edge `uv` is a bridge if `low(v) > num(u)`

In [ ]:

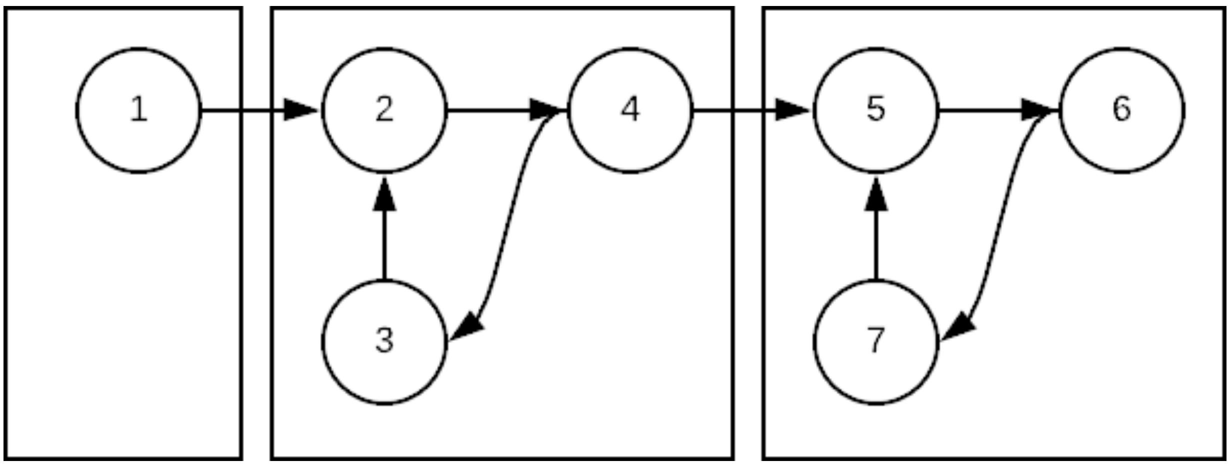
```
def articulation_dfs(neighbours, s):
    next_vertexes = [s]
    s.parent = None
    num = 0
    while len(next_vertexes) > 0:
        v = next_vertexes.pop()
        v.color = color.GREY
        v.num = v.low = num
        num += 1
        for u in neighbours(v):
            if u == v.parent:
                continue
            if u.color == color.WHITE:
                u.parent = v
                next_vertexes.append(u)
            else:
                v.low = min(v.low, u.low)

        v.color = color.BLACK
```

In [ ]:

```
num = 0
def articulation_dfs(neighbours, v):
    global num
    v.color = color.GREY
    v.num = v.low = num
    num += 1
    for u in neighbours(v):
        if u.color == color.WHITE:
            u.parent = v
            articulation_dfs(neighbours, u)
            u.low = min(u.low, v.low)
        elif u.parent != v:
            u.low = min(u.low, v.low)
        if v.low >= u.num:
            u.articulation = True
    v.color = color.BLACK
```

In [ ]:
```
def check_articulating(nodes, s):
    for _,u in nodes.items():
        if u == s:
            childs = sum(1 for _,v in nodes.items() if v != s and v.parent =
            if childs > 1:
                yield u
        else:
            for v in  neighbours(u):
                if v.low >= u.num:
                    yield u
                    break
```

Strongly connected components.



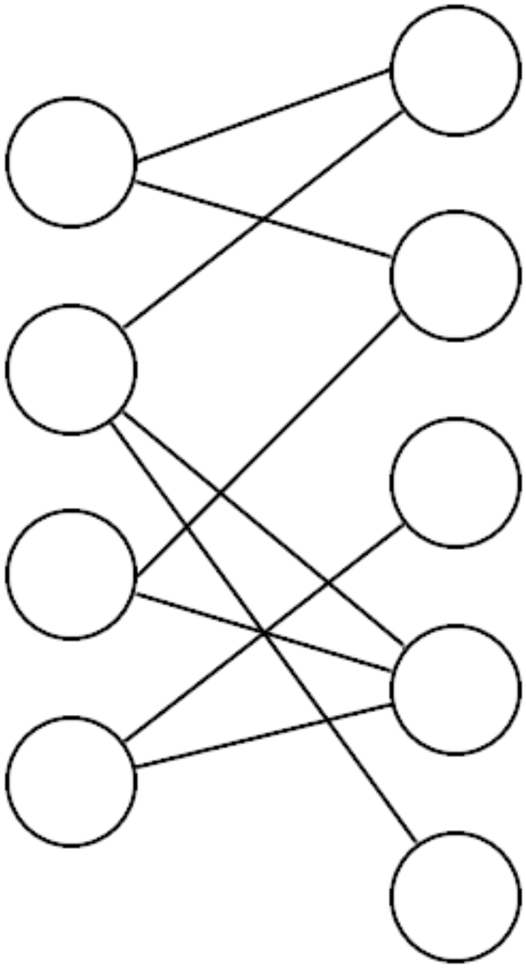
Number the nodes and use `dfs` to traverse the graph keeping track of the lowest number seen by a node.

```
node_count = 0
def stronly_cc_dfs(G, v):
    global node_count
    v.color = color.GREY
    v.min_seen = v.key = node_count
    node_count += 1
    for u in G.neighbors(v):
        if u.color == color.WHITE:
            stronly_cc_dfs(G, u)
        v.min_seen = min(v.min_seen, u.min_seen)
    v.color = color.BLACK
```

How to implement it iteratively?

Bipartite (or 2 colorable) graph check.

A bipartite graph can be dividing into two components with edges only between the components and not within the components.



Use `bfs` with two new colors, coloring neighbor nodes with different colors. If two adjacent nodes share the same color the graph is not 2-colorable/bipartite.

```
def bipartite_BFS(G, s, P):
    s.color = color.RED
    next_vertexes = [s]
    while len(next_vertexes) > 0:
        v = next_vertexes.pop()
        for u in G.neighbors(v):
            if u.color == color.WHITE:
                u.color = (color.BLUE if v.color == color.RED else
color.RED)
                next_vertexes.prepend(u)
            elif u.color == v.color:
                return False
    return True
```

(My personal) Recommendations

Most of the time you do not need classes to model nodes/graphs. Use the node's name (`int` or `str`) as its identifier, i.e. a string, an int or a tuple and use dictionaries to model the adjacency list/matrix and other properties of the nodes/edges.

For implicit graphs, use `collections.defaultdict` to hold the properties of nodes/edges.

From the graph structure, the main requirement is the `neighbtor` method, which can be implemented as a function instead.

For example:

In [1]:
```
import enum
import math

WHITE = 0
GREY = 1
BLACK = 2

# Crete a dictionary with the properties of the nodes. All
# nodes start with color white and dist property to infinite
nodes = dict((i, dict(name=i, color=WHITE, dist=math.inf))
             for i in range(8))
## Or using defaultdict
#from collections import defaultdict
#nodes = defaultdict(lambda: dict(color=color.WHITE, dist=math.inf))
```

In [ ]:
```
neighbors = {
    0: [1],
    1: [3],
    2: [1],
    3: [2, 4],
    4: [5],
    5: [7],
    6: [4],
    7: [6]}
```

In [ ]:
```
def dfs(i):
    v = nodes[i]
    v["color"] = GREY
    for j in neighbors[i]:
        u = nodes[j]
        if u["color"] == WHITE:
            dfs(j)
    v["color"] = BLACK
    print(i, v)
```