Lecture 7.
Graphs: DFS, BFS

Algorithms and Data Structures
Ivan Solomatin
Harbour.Space@UTCC Bangkok

Outline

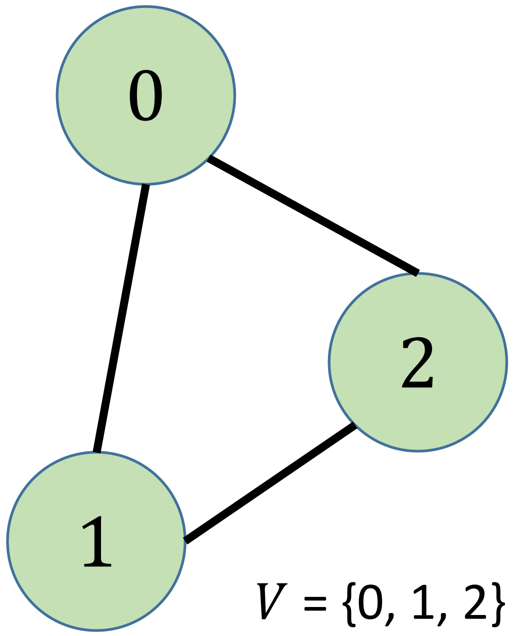
- Graph basics
    - Basic graph theory definitions
    - Ways to store graphs:
        - Adjacency matrix
        - Edges list
        - Adjacency list
        - Adjacency dict
- DFS
    - Graph traversal
    - Connected components search
    - Cycles search
    - Topological Sort
- BFS
    - Graph traversal
    - Shortest paths

Graph basics

- **Basic graph theory definitions**
- Ways to store graphs:
  - Adjacency matrix
  - Edges list
  - Adjacency list
  - Adjacency dict

Basic graph theory definitions

Definitions



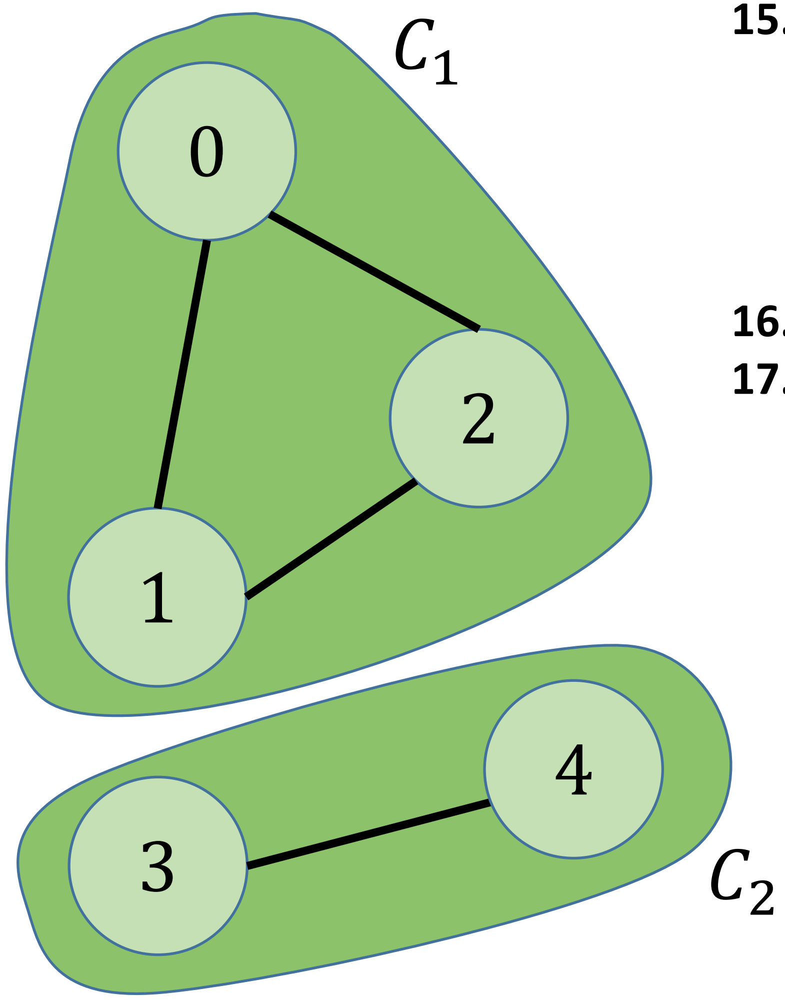
V = {0, 1, 2}

E = {(0, 1),
(0, 2),
(1, 2)}

1. **Graph:** $G = (V, E): E \subseteq \{\{v, u\}: v, u \in V\}$
2. **Edge:** $e \in E; \quad e = (v, u)$
3. **Adjacent vertices:** $v, u: (v, u) \in E$
4. **Vertex v is incident to edge e:** $v \in e$
5. **Loop:** $e = \{v, v\}$
6. **Multiple edges:** $e_1, e_2 \in E: e_1 = (v, u), e_2 = (v, u)$
7. **Simple graph:** graph without loops and without multiple edges
8. **Path:** sequence of vertices:
   $(v_0, v_1, \cdots, v_{n-1}): v_i \in V\ \forall i \in [0, n),$
   $(v_i, v_{i+1}) \in E\ \forall i \in [0, n-1).$
9. **Simple path:** path, with no repeated vertices
   $(v_0, v_1, \cdots, v_{n-1}):\ v_\mathrm{i} \neq v_\mathrm{j}\ \ \forall i \neq \mathrm{j}$
10. **Cycle:** path $(v_0, v_1, \cdots, v_{n-1}): v_0 = v_{\mathrm{n}-1}$
11. **Simple cycle:** cycle $(v_0, v_1, \cdots, v_{n-1})$:
    such that $(v_0, \cdots, v_{n-2})$ – simple path.
12. **Degree of vertex:** $\deg v = |\{e \in E: v \in e\}|$
13. **Vertex u is reachable from v:** $v \rightsquigarrow u\ \exists$ path $(v, \ldots, u)$.
14. **Directed graph:**
    $G = (V, E): E \subseteq V \times V$

Basic graph theory definitions

Definitions



15. Connected component: $\boldsymbol{C} \subseteq \boldsymbol{V}$:

- $\forall v, u \in C$: $v \rightsquigarrow u$ ($\exists$ path $v_1 \to v_2$)
- $\forall\, v \in C, u \in V \setminus C$: $v \not\rightsquigarrow u$ ($\nexists$ path $v \to u$)

Connected components are equivalence classes of reachability relation (13.) $\rightsquigarrow$.

16. Connected graph: $\exists!$ Connected component = V

17. Tree:

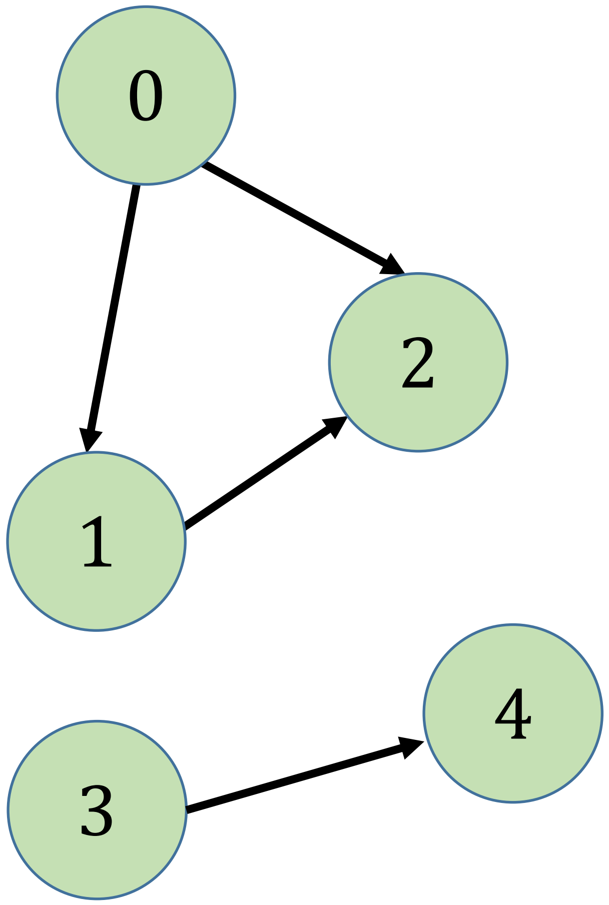
- Connected acyclic graph
- Connected graph: $|V| = |E| + 1$
- Acyclic graph: $|V| = |E| + 1$

Graph basics

- Basic graph theory definitions
- **Ways to store graphs:**
  - **Adjacency matrix**
  - Edges list
  - Adjacency list
  - Adjacency dict

Ways to store graphs

Adjacency matrix



Let's enumerate elements of V with integer numbers within $[0, |V|)$. To store $E \subseteq V \times \mathrm{V}$, we can use matrix $A \in \{0, 1\}^{|V| \times |V|}$, which shows adjacency relation.

$$A_{i,j} = 1 \iff (v_i, v_j) \in E$$

If graph is undirected, A is symmetric.

	v_0	v_1	v_2	v_3	v_4
v_0	$-$	$+$	$+$	$-$	$-$
v_1	$-$	$-$	$+$	$-$	$-$
v_2	$-$	$-$	$-$	$-$	$-$
v_3	$-$	$-$	$-$	$-$	$+$
v_4	$-$	$-$	$-$	$-$	$-$

Memory usage:

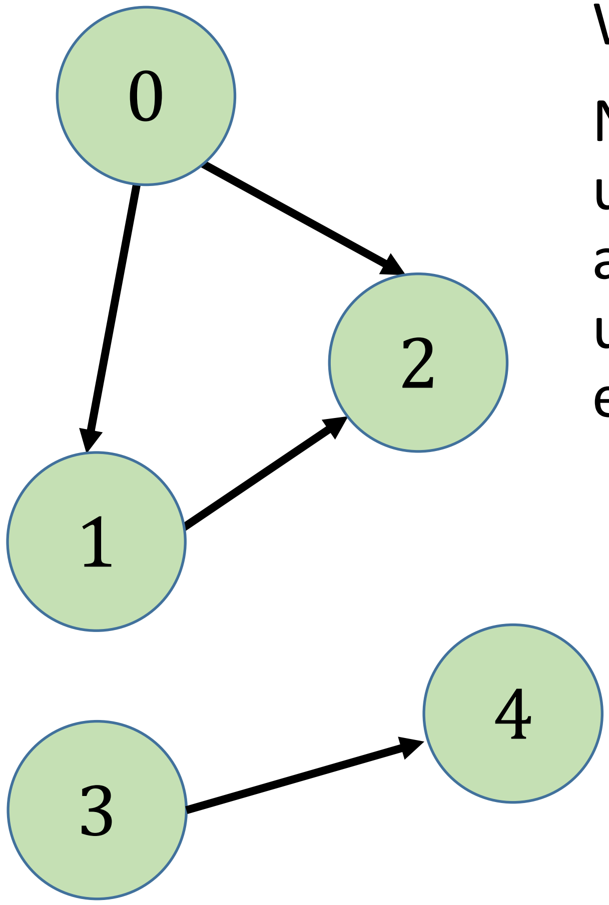
$O(|V|^2)$

Graph basics

- Basic graph theory definitions
- **Ways to store graphs:**
  - Adjacency matrix
  - **Edges list**
  - Adjacency list
  - Adjacency dict

Ways to store graphs

Edge list



We can store list of edges directly.

Number of edges: $|E| \leq |V|^2$, so memory usage is asymptotically not worse than for adjacency matrix. But in real world, $|E|$ is usually sufficiently less than $|V|^2$. In this case, edge list is better in terms on memory usage.

```
0, 1
0, 2
1, 2
3, 4
```

But it's not very convenient for obtaining adjacency relation. E.g. to obtain list of vertices adjacent with v, we need to iterate over whole list, which takes $O(|E|)$ operations.

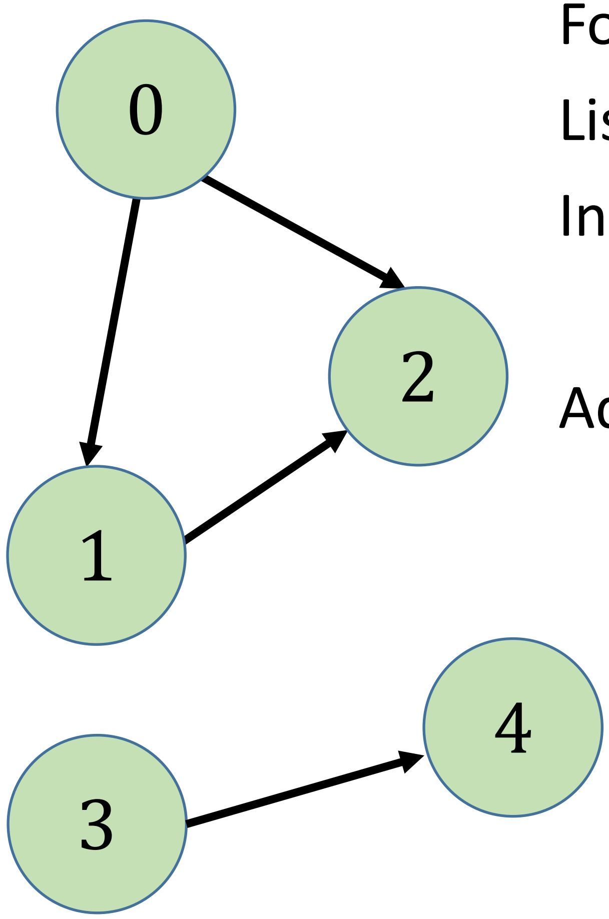
Memory usage: $O(|E|)$

Graph basics

- Basic graph theory definitions
- **Ways to store graphs:**
  - Adjacency matrix
  - Edges list
  - **Adjacency list**
  - Adjacency dict

Ways to store graphs

Adjacency list



For each node let's store list of adjacent nodes.

List of edges adjacent with v: `G[`v`].`

Initializing:

```
    G = [[] for i in range(V)]
```

Adding edge $v \rightarrow u$:

    G[v].append(u)

```
       G = [
0:          [1, 2],
1:          [2],
2:          [],
3:          [4],
4:          [] ]
```

Memory usage:

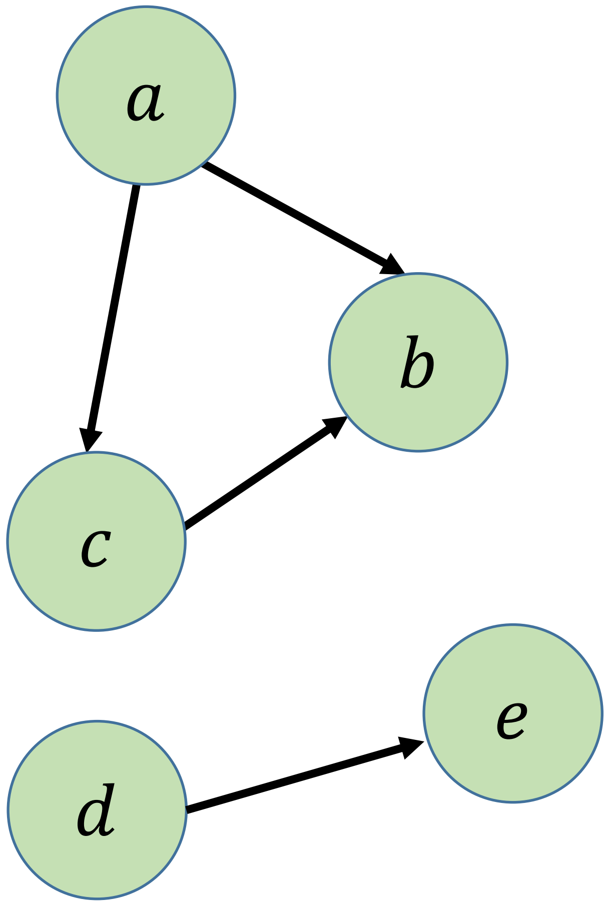
$O(|V| + |E|)$

Graph basics

- Basic graph theory definitions
- **Ways to store graphs:**
  - Adjacency matrix
  - Edges list
  - Adjacency list
  - **Adjacency dict**

Ways to store graphs

Adjacency dict



If vertices are not enumerated with numbers within $[0, |V|)$, it can be easier to use dict instead of list:

List of edges adjacent with v: `G[v]`

Initialization:

```
G = {v: [] for v in V}
```

Adding edge $v \to u$:

```
G[v].append(u)
```

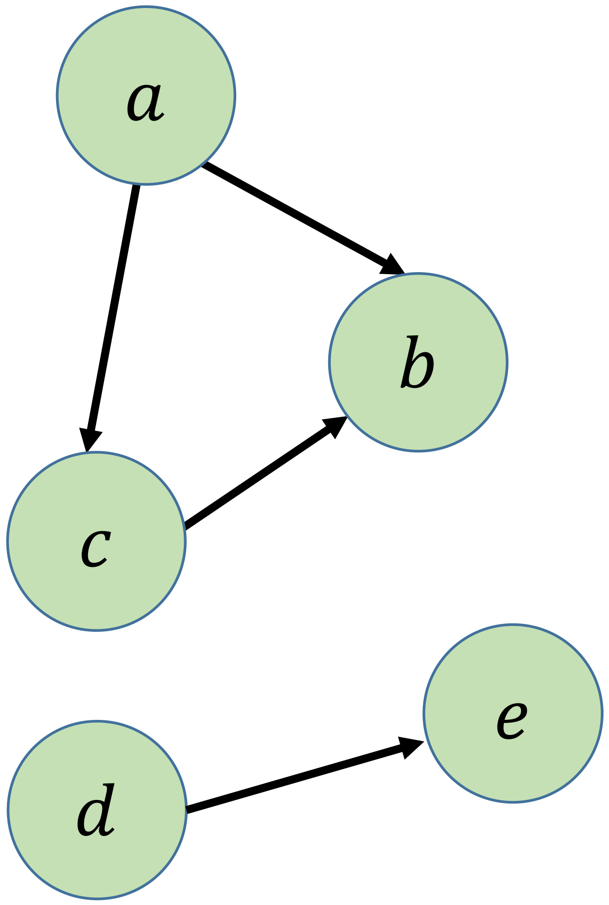
```
G = [
      'a': ['b', 'c'],
      'b': [],
      'c': ['b'],
      'd': ['e'],
      'e': [] ]
```

Memory usage:

$$O(|V| + |E|)$$

Ways to store graphs

Adjacency dict



If vertices are not enumerated with numbers within $[0, |V|)$, it can be easier to use dict instead of list:

```
G = {v: {} for v in V}
```

Adding edge $v \to u$:

```
G[v][u] = True
# or any additional information, e.g. – weight
```

List of edges adjacent with v:

```
G[v].keys()
```

Memory usage:

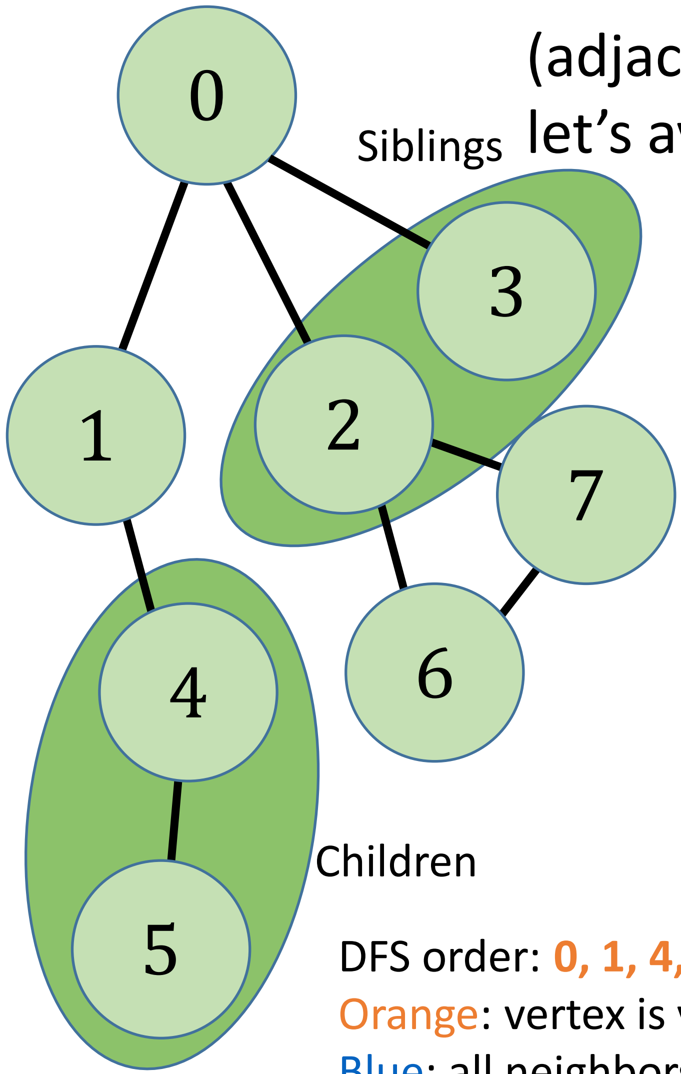
$O(|V| + |E|)$

DFS – Depth First Search

- **Graph traversal**
- Connected components search
- Cycles search
- Topological Sort

DFS. Graph Traversal

Idea



For a vertex, let's iterate over all it's neighbors (adjacent vertices) and go to each vertex, also, let's avoid repeated visiting (returning is ok).

DFS explores depth first. This means that it visits **children** of vertex before it's **siblings**.

```
def dfs($v$):
    visited[$v$] = True
    for $u$: $(v, u) \in E$:
        if not visited[$u$]:
            dfs($u$)
```

DFS order: **0, 1, 4, 5, 5, 4, 1, 0, 2, 6, 7, 7, 6, 2, 3, 3, 0**

Orange: vertex is visited first

Blue: all neighbors are visited

DFS. Graph Traversal

Implementation

```
import sys
sys.setrecursionlimit(128 * 10**5)


def dfs(G, visited, v):
 # v is visited first time
    visited[v] = True
    for u in G[v]:
        if not visited[u]:
            dfs(G, visited, u)
 # all neighbours of v are visited
```

DFS. Graph Traversal

Non-recursive implementation

```
def dfs(G, visited, s):
    vertex_stack = [s]
    while vertex_stack:
        v = vertex_stack.pop()
        if not visited[v]:
            # v is visited first time
            visited[v] = True
            for u in G[v]:
                if not visited[u]:
                    vertex_stack.append(u)
```

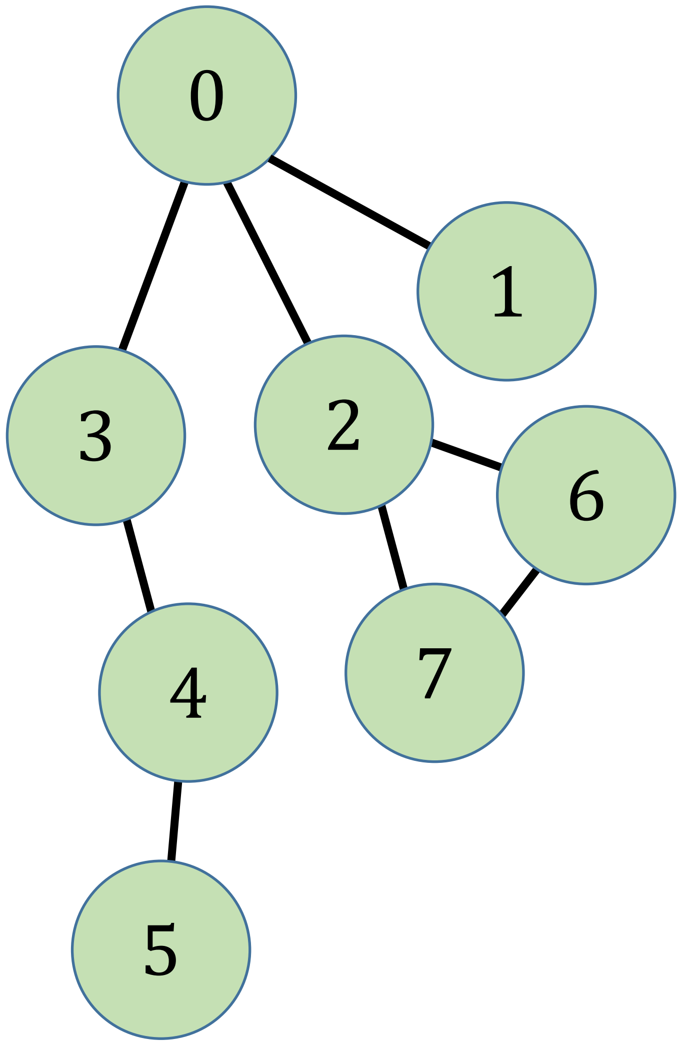
DFS. Graph Traversal

Non-recursive implementation

```
def dfs(G, visited, s):
    vertex_stack = [s]
    while vertex_stack:
        v = vertex_stack[-1]
        if not visited[v]:
            # v is visited first time
            visited[v] = True
            for u in G[v]:
                if not visited[u]:
                    vertex_stack.append(u)
        else:
            # all neighbours of v were visited
            vertex_stack.pop()
```

DFS. Graph Traversal

Example



vertex_stack:

0	1	2	6	7	6

DFS – Depth First Search

- Graph traversal
- **Connected components search**
- Cycles search
- Topological Sort

Connected components search

Idea

Let’s notice that after calling $\mathrm{DFS}(v)$ we can guarantee that all vertices which are reachable from v are visited.

In case of undirected graph this means that it traverses all the vertices which are in the same connected component as v.

This idea can be used to determine connected components. E.g. to calculate number of connected components, you can just calculate minimum number of DFS calls you need to visit all the nodes:

```python
visited = [False] * len(G)
n_components = 0
for i in range(len(G)):
    if not visited[i]:
        dfs(G, visited, i)
        n_components += 1
```

Connected components search

Directed graph case

In case of directed graph, connected components are not defined. Because, reachability is not a symmetric relation any more, so, it doesn’t have equivalence classes.

But for directed graphs we still can guarantee that after calling $DFS(v)$ all vertices reachable from v are visited. If you need to traverse all the vertices of the graph, you should use the same approach:

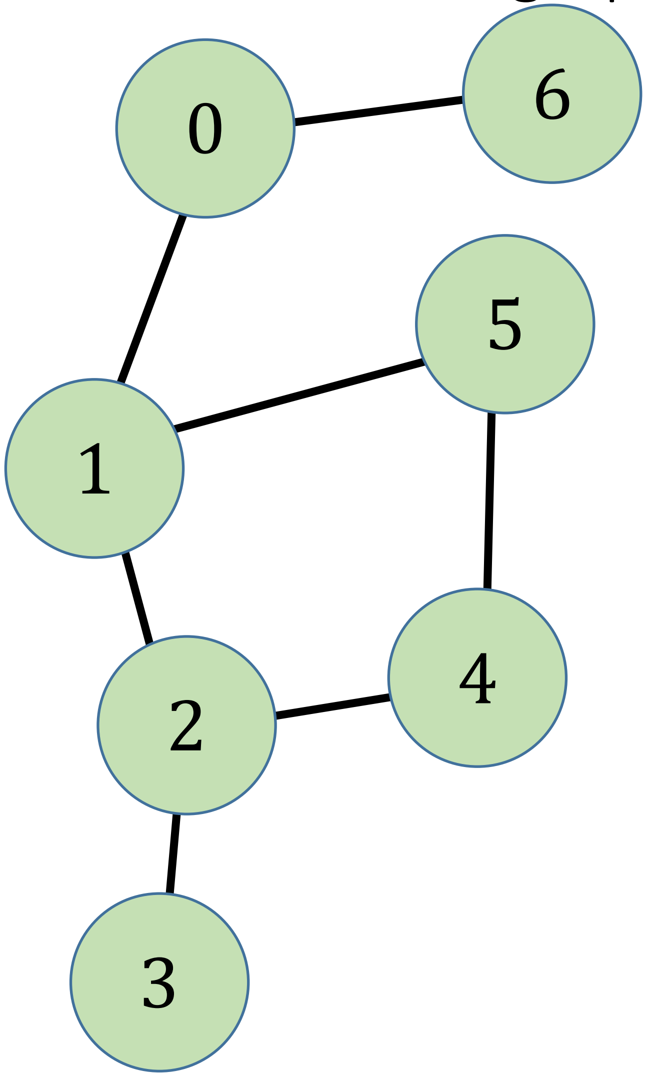
```python
visited = [False] * len(G)
for i in range(len(G)):
    if not visited[i]:
        dfs(G, visited, i)
```

DFS – Depth First Search

- Graph traversal
- Connected components search
- **Cycles search**
- Topological Sort

DFS. Cycles search

Non directed graph



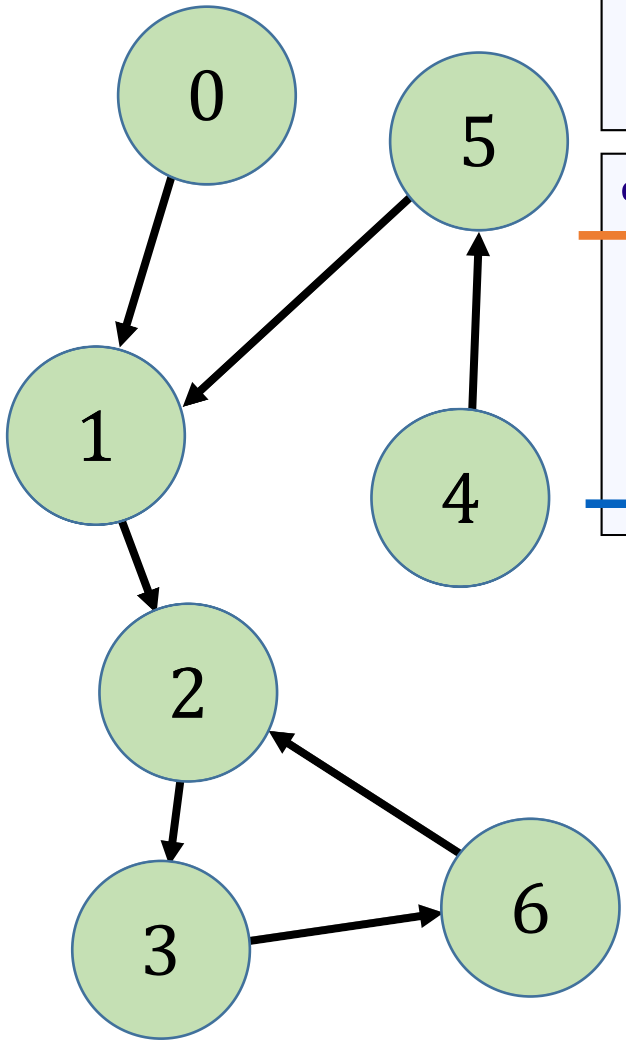
```
visited = [False] * len(G)
for i in range(len(G)):
    if not visited[i]:
        dfs(G, visited, i)
```

```
def dfs(G, visited, v, p=-1):
    visited[v] = True
    for u in G[v]:
        if not visited[u]:
            dfs(G, visited, u, v)
        elif u != p:
            print('Cycle found!')
```

```
def dfs(G, visited, s):
    vertex_stack = [(s, -1)]
    while vertex_stack:
        v, p = vertex_stack[-1]
        if not visited[v]:
            visited[v] = True
            for u in G[v]:
                if not visited[u]:
                    vertex_stack.append((u, v))
                elif u != p:
                    print('Cycle found!')
        else:
            vertex_stack.pop()
```

DFS. Cycles search

Directed graph



```
visited = [0] * len(G)
for i in range(len(G)):
    if not visited[i]:
        dfs(G, visited, i)
```

```
def dfs(G, visited, v):
    visited[v] = 1
    for u in G[v]:
        if not visited[u]:
            dfs(G, visited, u)
        elif visited[u] == 1:
            print('Cycle found!')
    visited[v] = 2
```

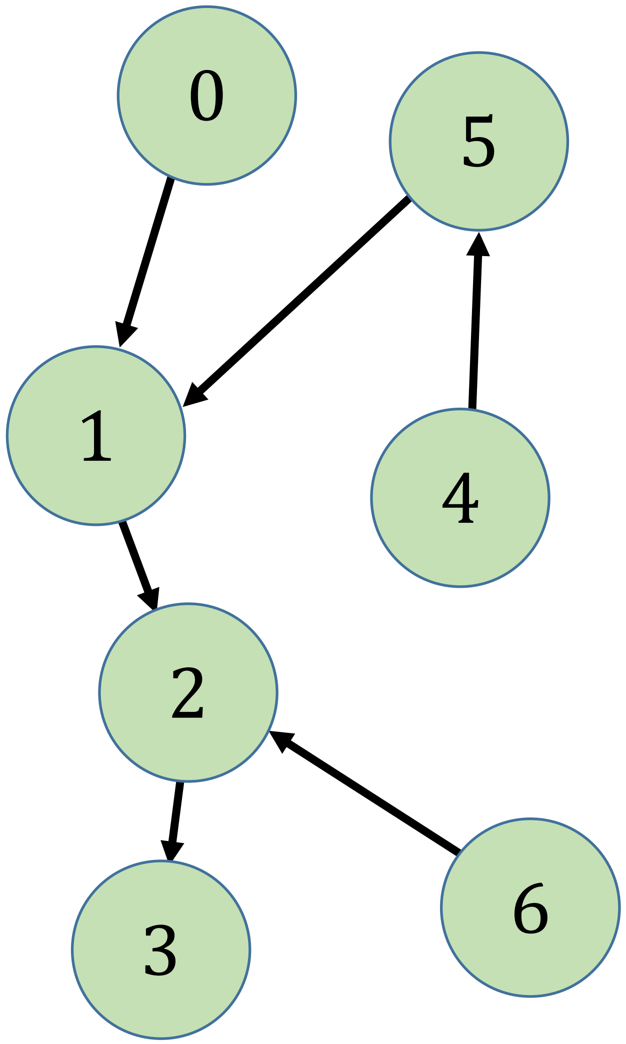
```
def dfs(G, visited, s):
    vertex_stack = [s]
    while vertex_stack:
        v = vertex_stack[-1]
        if not visited[v]:
            visited[v] = 1
            for u in G[v]:
                if not visited[u]:
                    vertex_stack.append((u))
                elif visited[v] == 1:
                    print('Cycle found!')
        else:
            visited[v] = 2
            vertex_stack.pop()
```

DFS – Depth First Search

- Graph traversal
- Connected components search
- Cycles search
- **Topological Sort**

DFS. Topological sort

Problem statement



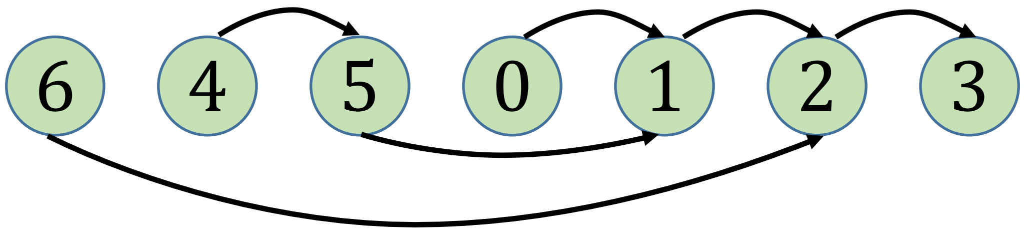
Graph $G = (V, E)$ defines partial order on elements of V:

$$v \prec u \quad \Leftrightarrow \quad (v, u) \in E$$

Task is to sort items according to this partial order:

$$v_{i_0}, v_{i_1}, \dots, v_{i_{|V|-1}}$$

$$\forall j_1, j_2: \; v_{i_{j_1}} \prec v_{i_{j_2}} \; \Rightarrow j_1 < j_2$$

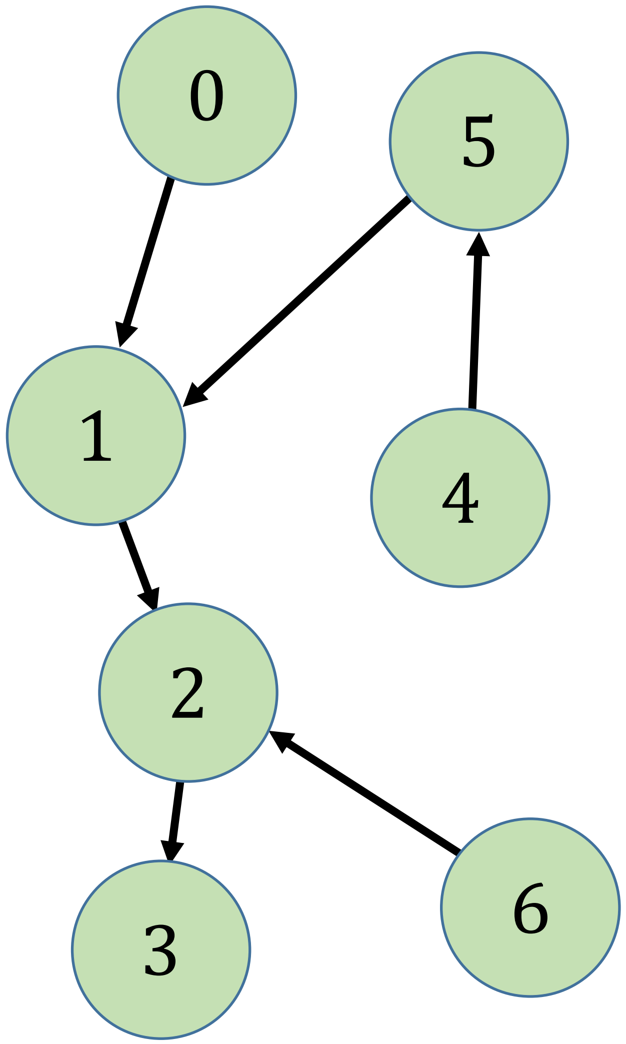


If graph contains a cycle, it can't be topologically sorted.

Example: Course plan.
Vertices – topics to be covered in the course.
$(v, u) \in E \Leftrightarrow$ topic u should be explained before v.
Topics in the course should be topologically sorted.

DFS. Topological sort

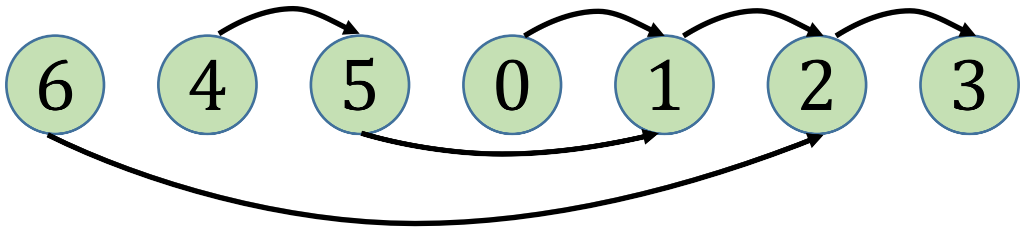
Idea



Vertex v in sorted array should go before all vertices reachable from v (by transitivity).

Idea: Let's build sorted array in reversed order. For each vertex v let's traverse all vertices reachable from v with $DFS(v)$, and append all these vertices to array. After that we can append v to array.

This procedure will guarantee the following: each vertex v is added only after adding all vertices reachable from v. That's a reversed topological sort order.



DFS. Topological sort

Implementation

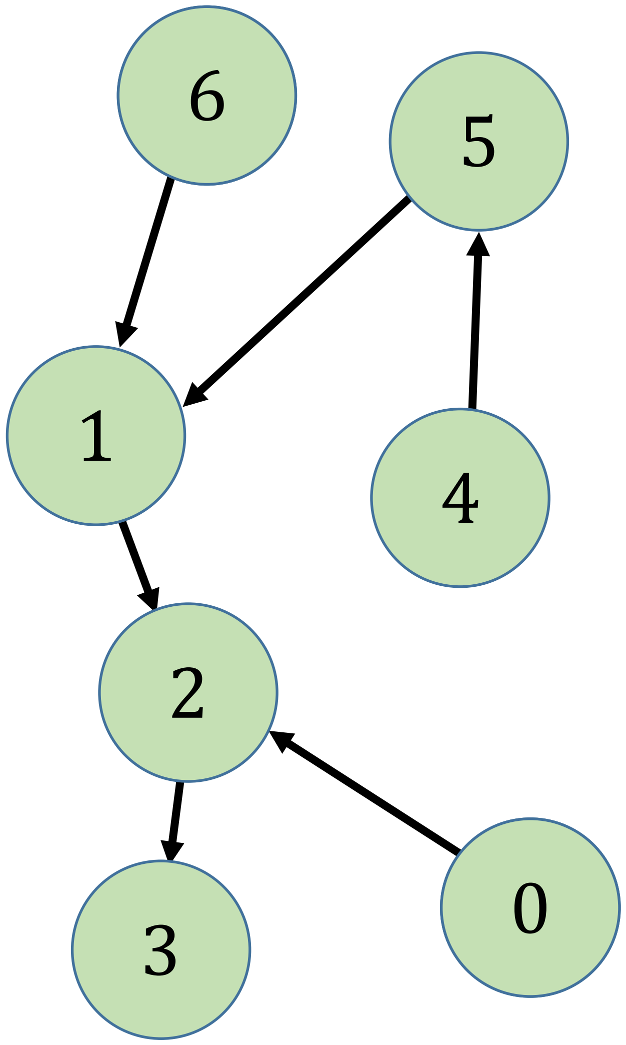
```python
def dfs(G, visited, topsort, v):
    visited[v] = 1
    for u in G[v]:
        if not visited[u]:
            dfs(G, visited, topsort, u)
        elif visited[u] == 1:
            print('Impossible')
    topsort.append(v)
    visited[v] = 2
```

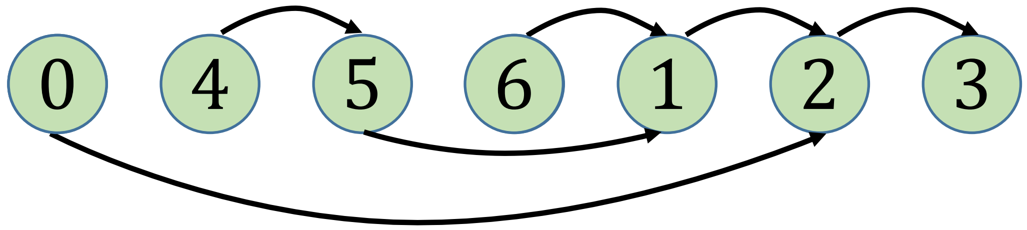
```python
def dfs(G, visited, topsort, s):
    vertex_stack = [s]
    while vertex_stack:
        v = vertex_stack[-1]
        if not visited[v]:
            visited[v] = 1
            for u in G[v]:
                if not visited[u]:
                    vertex_stack.append(u)
                elif visited[u] == 1:
                    print('Impossible')
        else:
            # all neighbors of v were visited
            if visited[v] == 1:
                # this happened first time
                topsort.append(v)
            visited[v] = 2
            vertex_stack.pop()
```

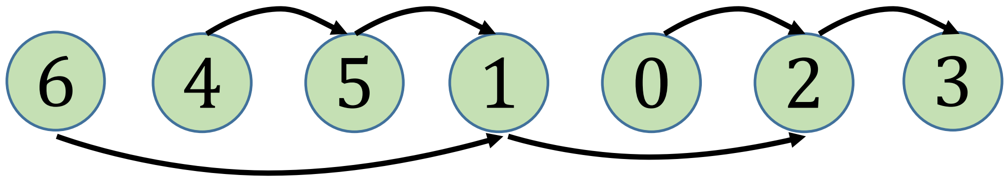
```python
topsort = []
for i in range(N):
    if not visited[i]:
        dfs(G, visited, topsort, i)
topsort = topsort[::-1]
```

DFS. Topological sort

Example







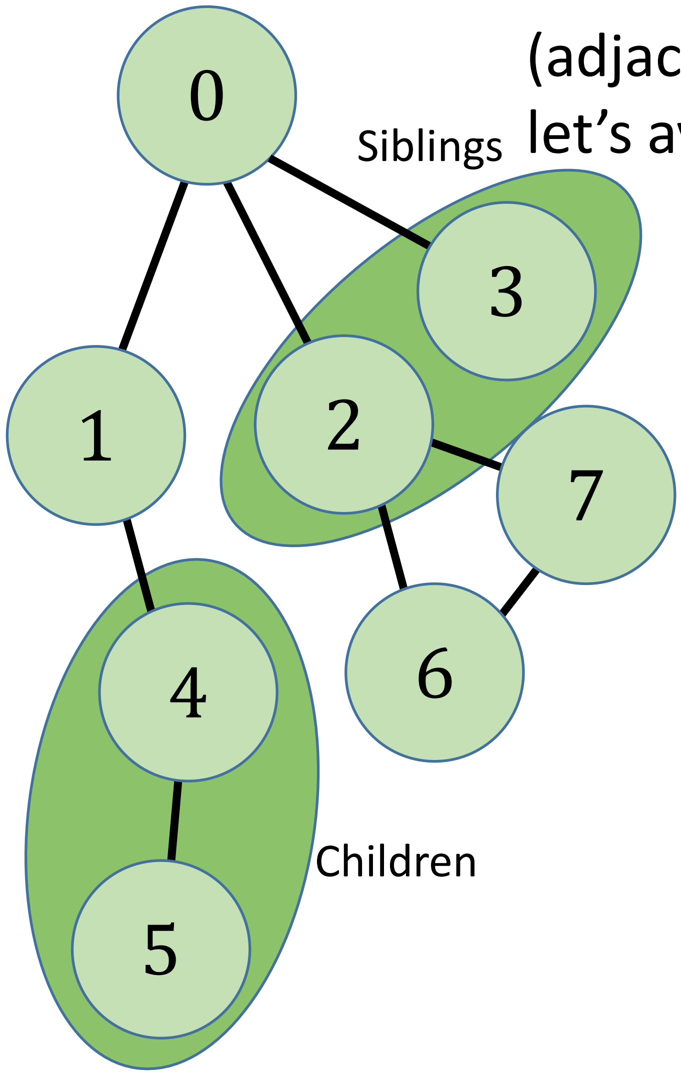
BFS – Breadth First Search

- **Graph traversal**
- Shortest paths

BFS: Graph traversal

Idea

For a vertex, let’s iterate over all it’s neighbors (adjacent vertices) and go to each vertex, also, let’s avoid repeated visiting.



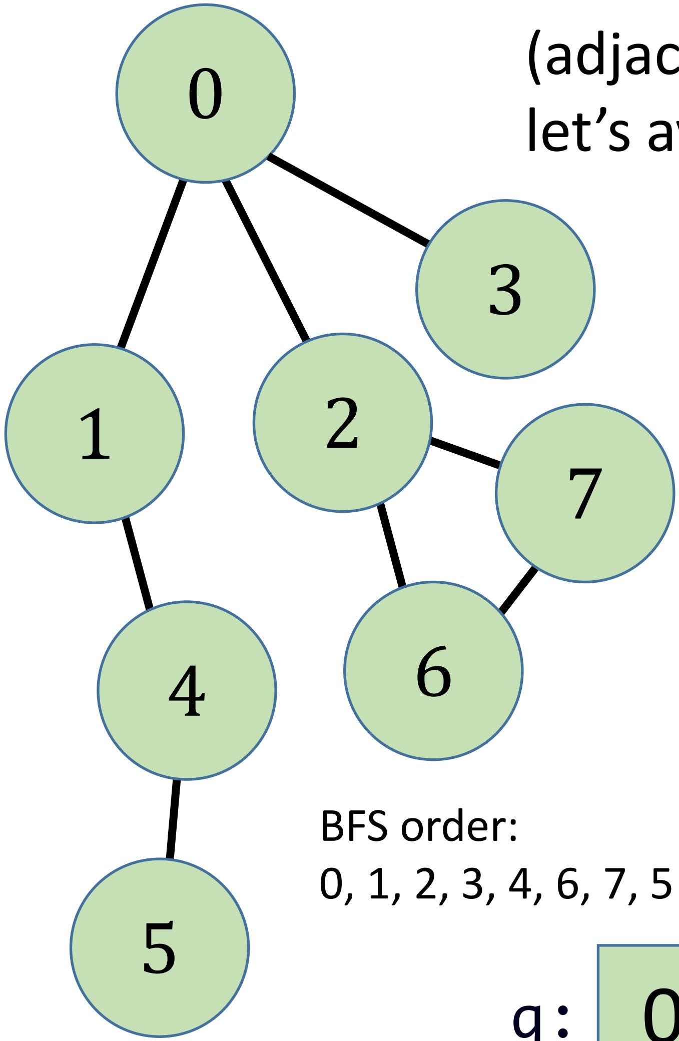
BFS explores breadth first. This means that it visits **siblings** of vertex before it’s **children**.

```
def bfs($s$):
    q = queue([$s$])
    while q:
        $v$ = q.pop()
        if not visited[$v$]:
            visited[$v$] = True
            for $u$: $(v,u) \in E$:
                if not visited[$u$]:
                    q.push($u$)
```

BFS: Graph traversal

Idea

For a vertex, let’s iterate over all it’s neighbors (adjacent vertices) and go to each vertex, also, let’s avoid repeated visiting.



DFS explores breadth first. This means that it visits **siblings** of vertex before it’s **children**.

```
def bfs($s$):
    q = queue([$s$])
    while q:
        $v$ = q.pop()
        if not visited[$v$]:
            visited[$v$] = True
            for $u$: $(v, u) \in E$:
                if not visited[$u$]:
                    q.push($u$)
```

BFS order:
0, 1, 2, 3, 4, 6, 7, 5

q:

0	1	2	3	4	6	7	5	7

BFS : Graph traversal

Implementation

```
def dfs(G, visited, s):
    vertex_stack = [s]
    while vertex_stack:
        v = vertex_stack.pop()
        if not visited[v]:
            # v is visited first time
            visited[v] = True
            for u in G[v]:
                if not visited[u]:
                    vertex_stack.append(u)
```

stack → queue

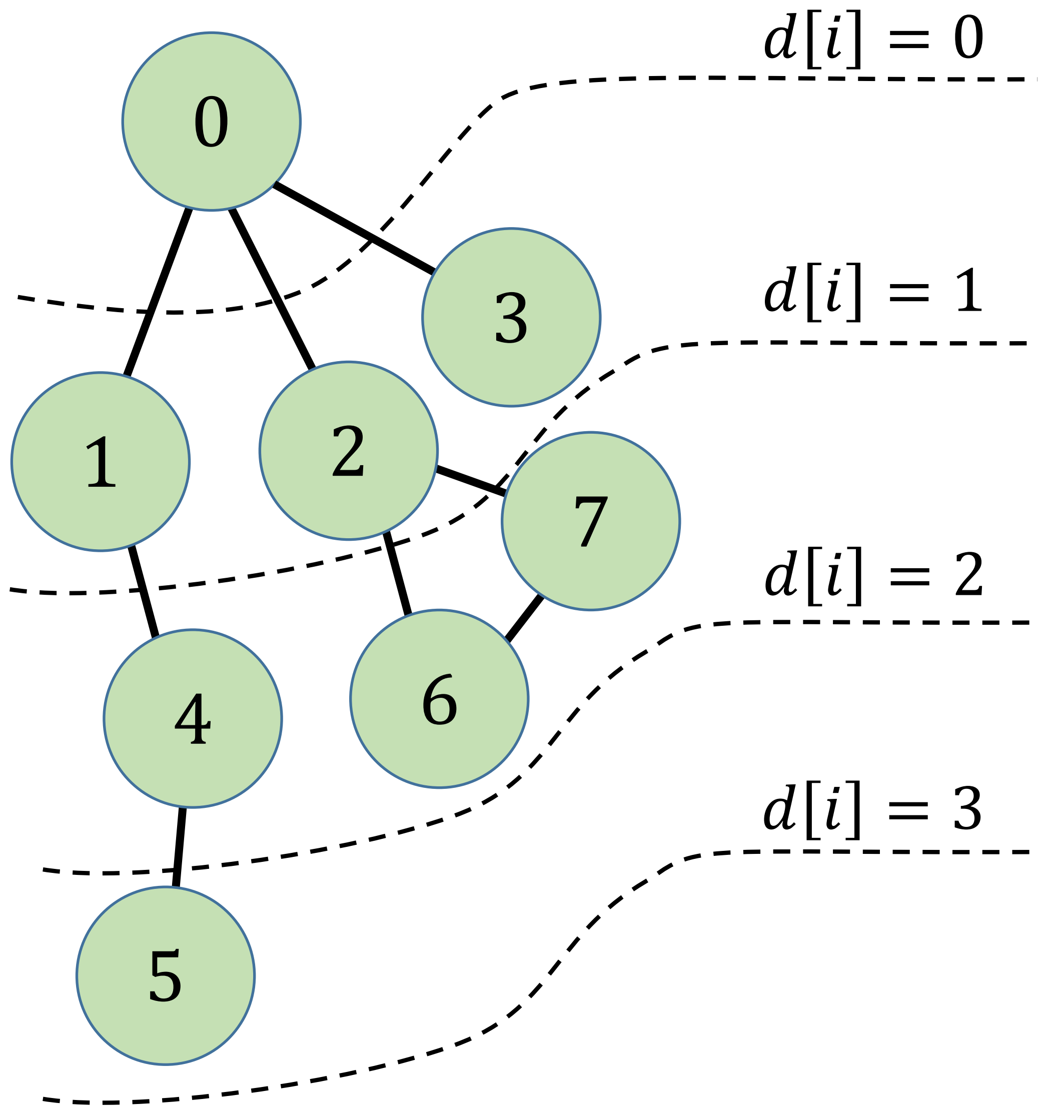
```
from collections import deque
def bfs(G, visited, s):
    vertex_queue = deque([s])
    while vertex_queue:
        v = vertex_queue.popleft()
        if not visited[v]:
            # v is visited first time
            visited[v] = True
            for u in G[v]:
                if not visited[u]:
                    vertex_queue.append(u)
```

BFS – Breadth First Search

- Graph traversal
- **Shortest paths**

BFS: Shortest paths

Idea



$d[i]$ – length shortest path $s \to i$

1. Initially, queue contains all vertices with $d[i] = 0$ (vertex s).

2. Then, adding all unvisited vertices adjacent to ones with $d[i] = 0$ gives us all vertices with $d[i] = 1$.

3. Then, adding all unvisited vertices adjacent to ones with $d[i] = 1$ gives us all vertices with $d[i] = 2$.

4. Etc…

BFS: Shortest paths

Idea

This means, BFS iterates over vertices in non-decreasing $d[i]$ order.

We can utilize this knowledge to calculate d for each value.

Also, for each vertex we can store previous vertex which gives optimal path (similarily to dp). That will allow to restore optimal path $s \to i$ for each i.

BFS: Shortest paths

Implementation

```
def bfs(G, visited, d, p, s):
    vertex_queue = deque([s])
    while vertex_queue:
        v = vertex_queue.popleft()
        visited[v] = 2
        for u in G[v]:
            if not visited[u]:
                visited[u] = 1
                vertex_queue.append(u)
                d[u] = d[v] + 1
                p[u] = v
```

BFS: Shortest paths

Small trick

In the beginning, we add start vertex to queue:

```
vertex_queue = deque([s])
```

So, we “start forest fire” in one point. What if we “start forest fire” in several vertices:

```
S = [s1, s2, …]
vertex_queue = deque(S)
```

The algorithm will traverse graph by paths, starting from each of these points.

This will allow to calculate:

$$d[i] = \min_{s \in S}(\rho(s, i)), \text{ where}$$

$$\rho(v, u) - \text{minimal length of path } v \rightarrow u$$

We just need to initialize previous algorithm with $d[s] = 0\ \forall s \in S$.

Conclusion

Thank you for watching!