ETH zürich

Departement Informatik

Programming and Problem-Solving

Graphs and Graph Algorithms

Manuela Fischer and Dennis Komm

Spring 2021 – May 27, 2021

Graphs

Searching in Networks

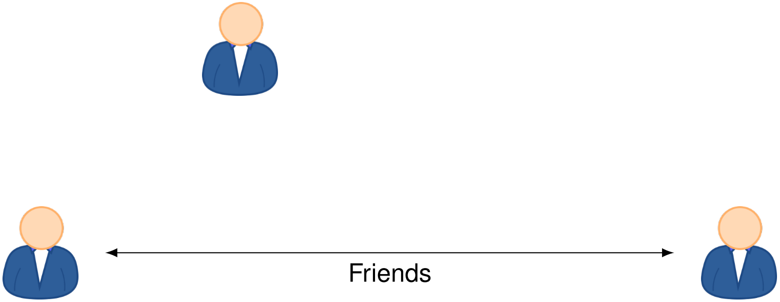
Social Network

Social Network

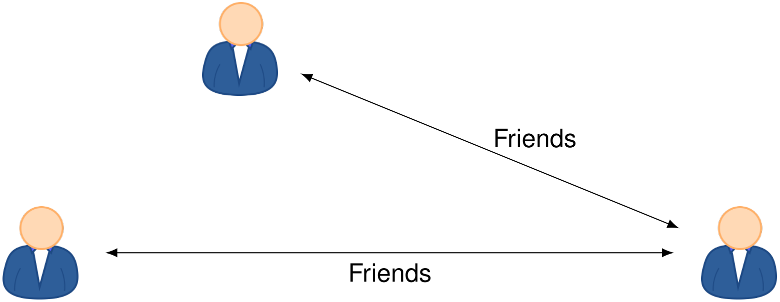
Social Network

Friends

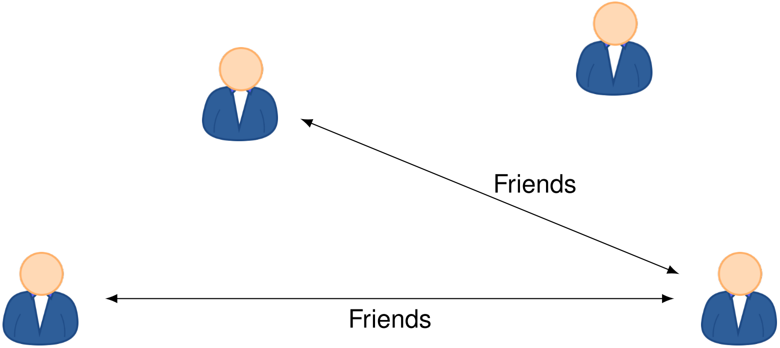
Social Network



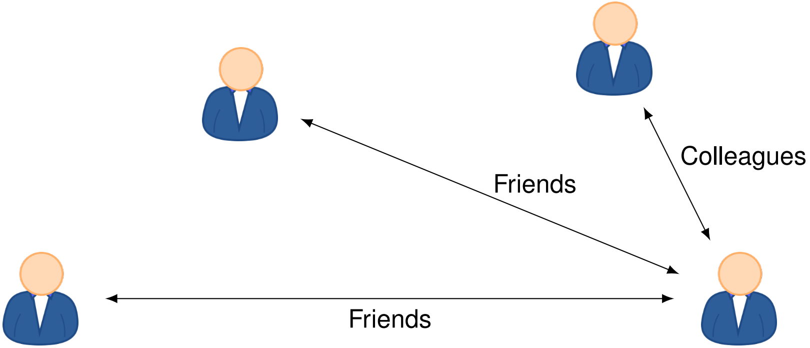
Social Network



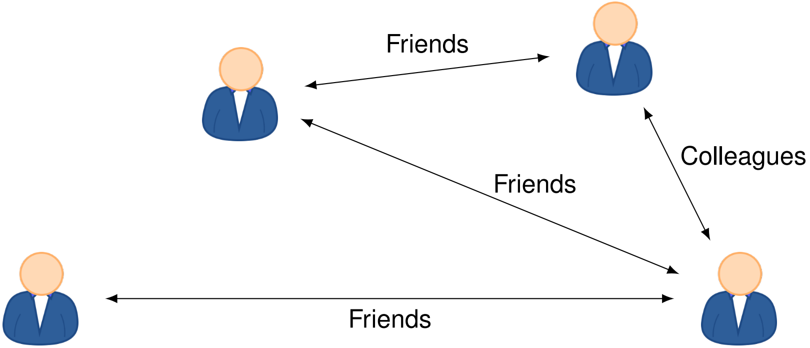
Social Network



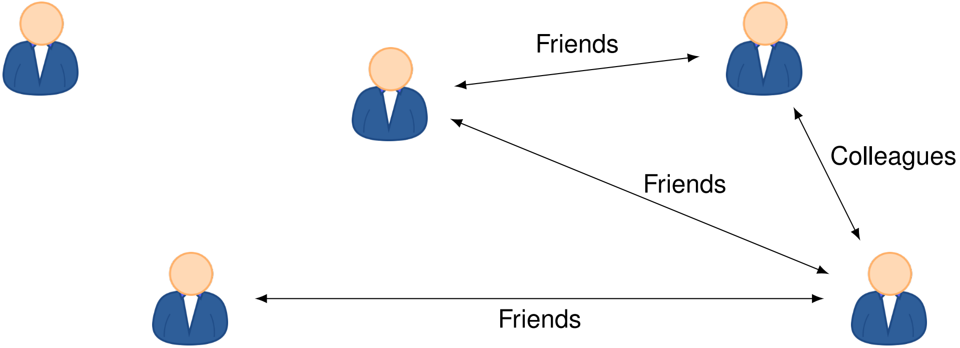
Social Network



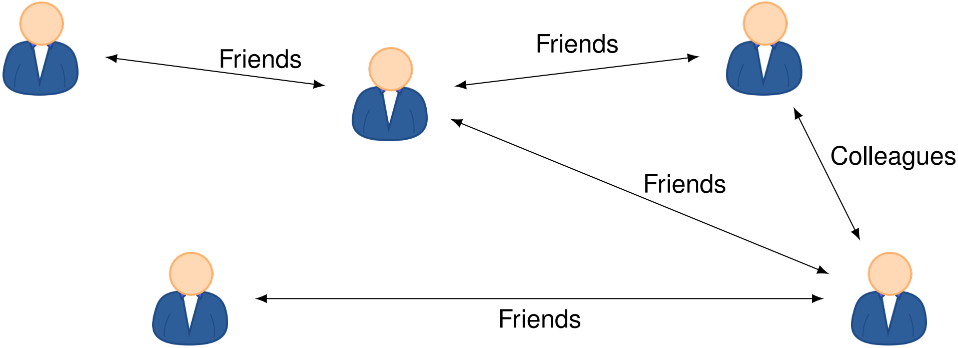
Social Network



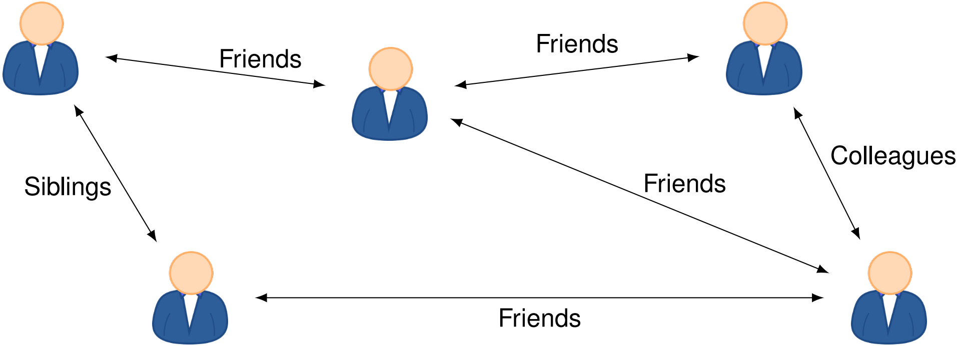
Social Network



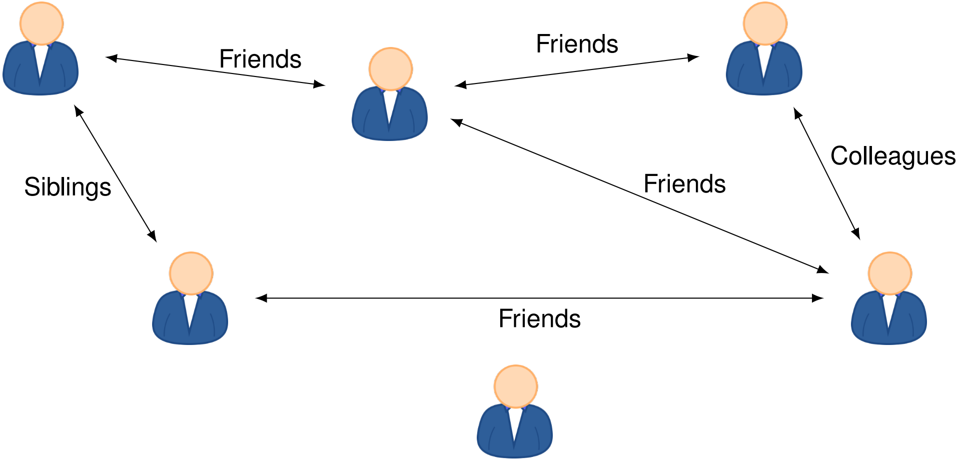
Social Network



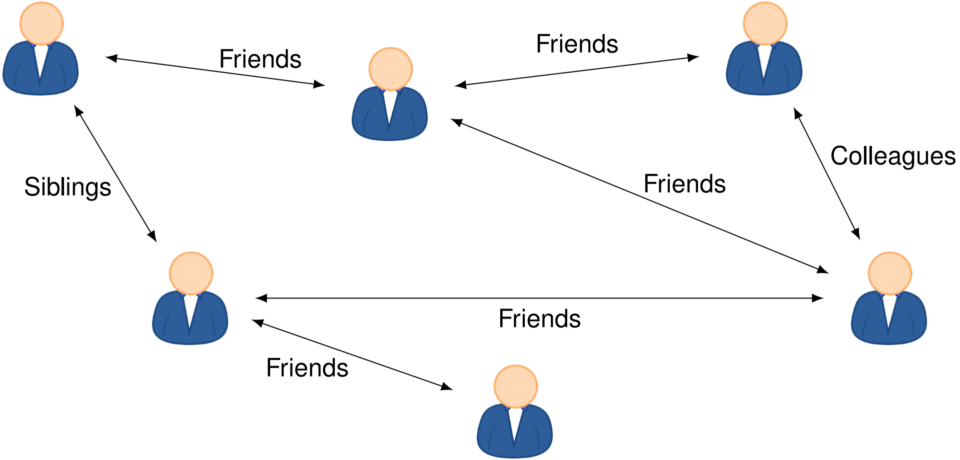
Social Network



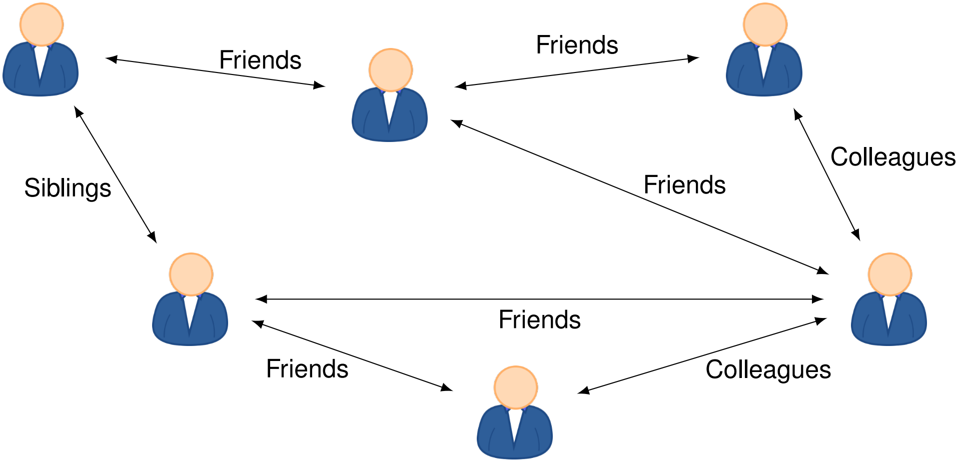
Social Network



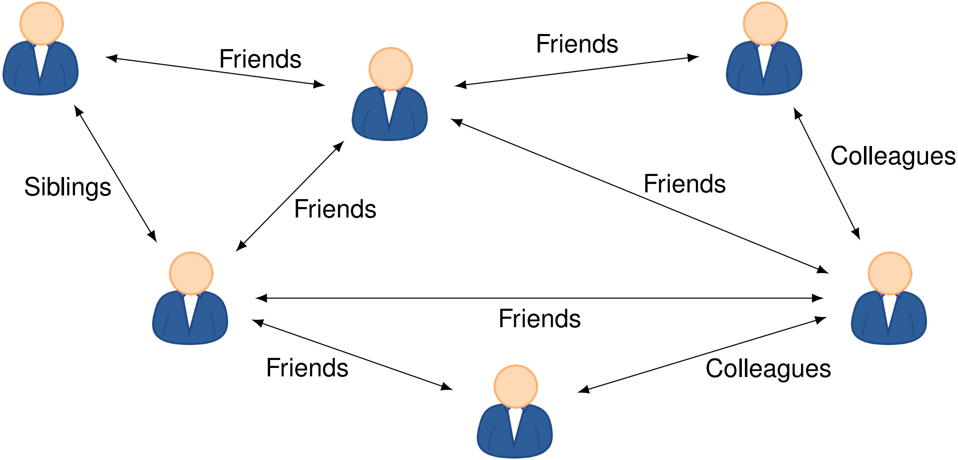
Social Network



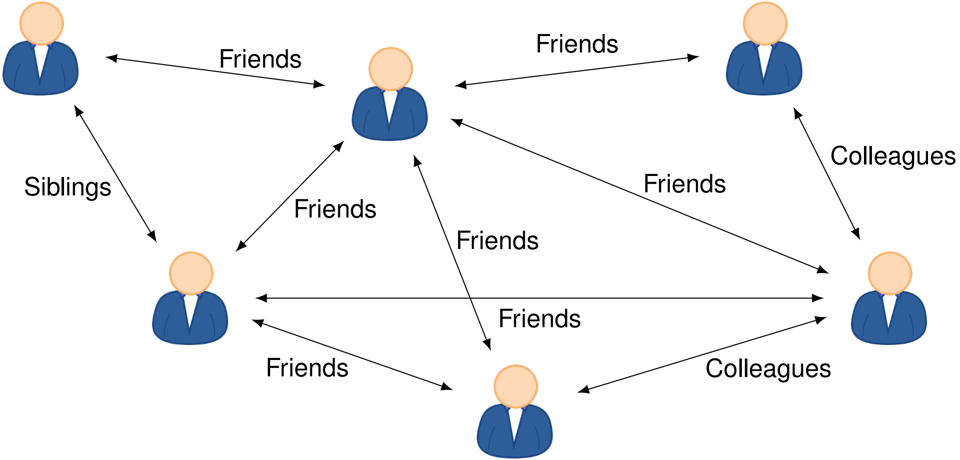
Social Network



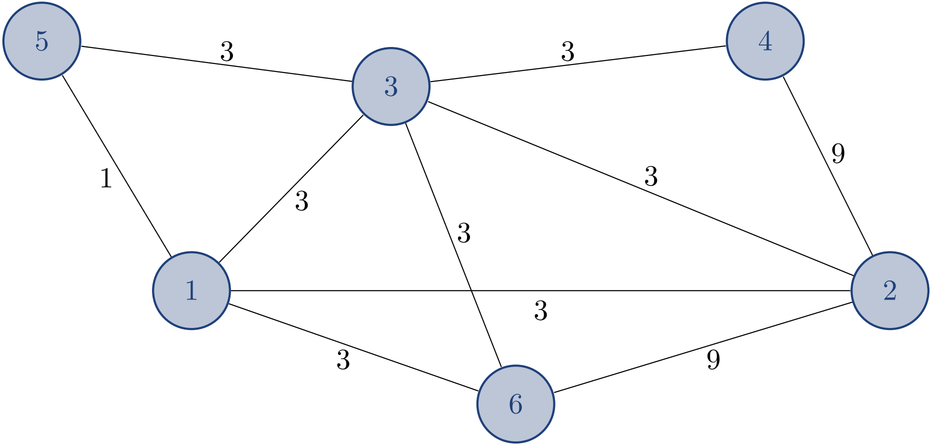
Social Network



Social Network



Abstract Modelling



Abstract Modelling

A **graph** $G = (V, E, w)$ consists of

1. a set V of vertices
2. a set E of edges between some of the vertices
3. (a weight function w)

Abstract Modelling

A **graph** $G = (V, E, w)$ consists of

1. a set V of vertices
2. a set E of edges between some of the vertices
3. (a weight function w)

- Vertices are called $v_0, v_1, v_2, \ldots$

Abstract Modelling

A **graph** $G = (V, E, w)$ consists of

1. a set V of vertices
2. a set E of edges between some of the vertices
3. (a weight function w)

- Vertices are called $v_0, v_1, v_2, \ldots$
- Graphs are either **weighted** or **unweighted**

Abstract Modelling

A **graph** $G = (V, E, w)$ consists of

1. a set V of vertices
2. a set E of edges between some of the vertices
3. (a weight function w)

- Vertices are called $v_0, v_1, v_2, \ldots$
- Graphs are either **weighted** or **unweighted**
- Graphs are either **directed** or **undirected**

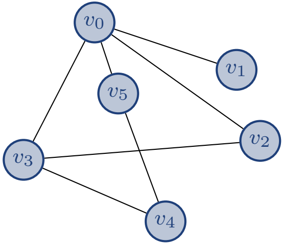
Abstract Modelling

A **graph** $G = (V, E, w)$ consists of

1. a set V of vertices
2. a set E of edges between some of the vertices
3. (a weight function w)

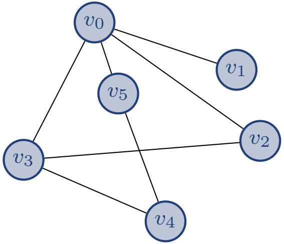
- Vertices are called $v_0, v_1, v_2, \ldots$
- Graphs are either **weighted** or **unweighted**
- Graphs are either **directed** or **undirected**
- Graphs are either **connected** or **unconnected**

Abstract Modelling

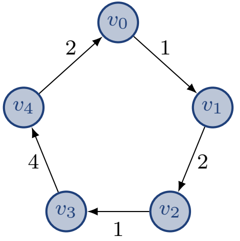


Undirected unweighted graph

Abstract Modelling

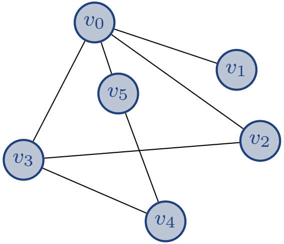


Undirected unweighted graph

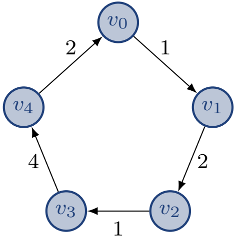


Undirected weighted graph

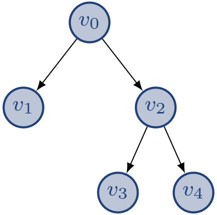
Abstract Modelling



Undirected unweighted graph

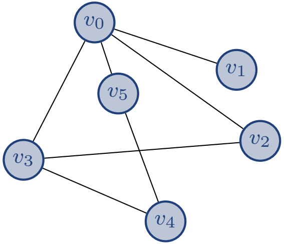


Undirected weighted graph

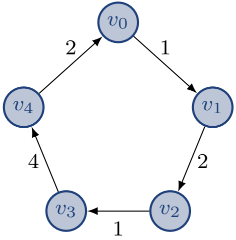


Directed unweighted graph

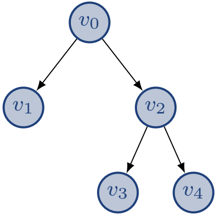
Abstract Modelling



Undirected unweighted graph



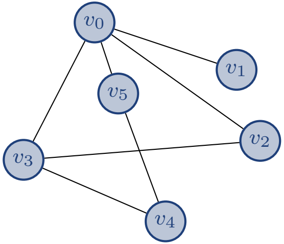
Undirected weighted graph



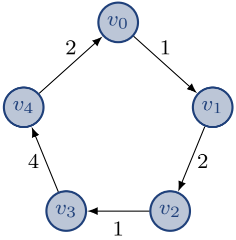
Directed unweighted graph

Which type of graph is used depends on what we want to model

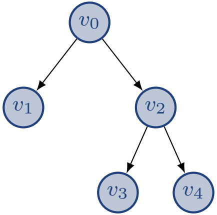
Abstract Modelling



Undirected unweighted graph



Undirected weighted graph



Directed unweighted graph

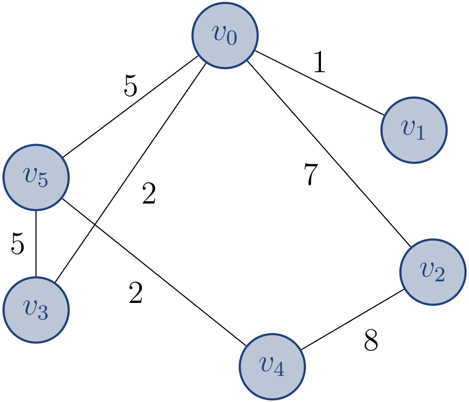
Which type of graph is used depends on what we want to model

We mostly consider undirected, unweighted, connected graphs

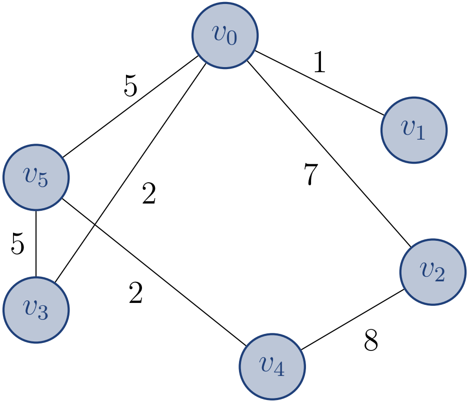
Graphs

On the Computer

Adjacency Matrices – Undirected Weighted Graphs

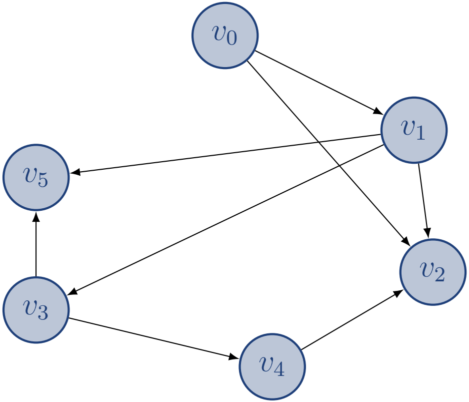


Adjacency Matrices – Undirected Weighted Graphs

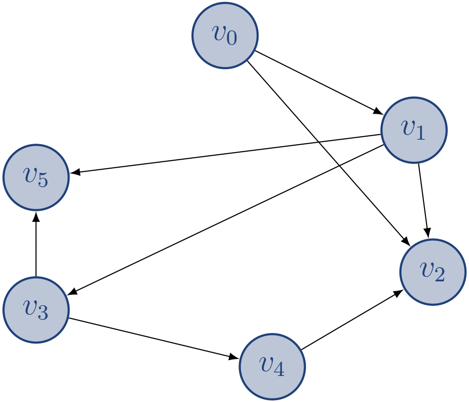


$$\begin{pmatrix} 0 & 1 & 7 & 2 & 0 & 5 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 8 & 0 \\ 2 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 8 & 0 & 0 & 2 \\ 5 & 0 & 0 & 5 & 2 & 0 \end{pmatrix}$$

Adjacency Matrices – Directed Unweighted Graphs



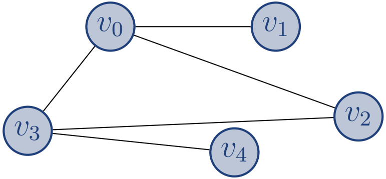
Adjacency Matrices – Directed Unweighted Graphs



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Adjacency Matrices – Directed / Undirected Graphs

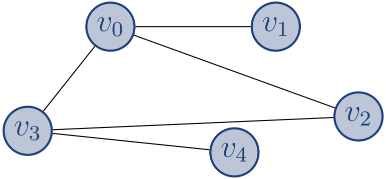
Matrices of undirected graphs
are symmetric



$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

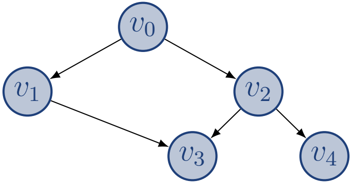
Adjacency Matrices – Directed / Undirected Graphs

Matrices of undirected graphs are symmetric



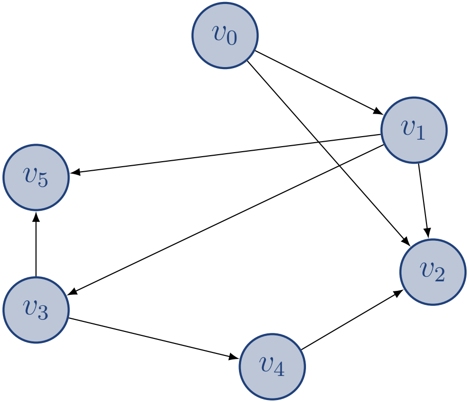
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Matrices of directed graphs are not (always) symmetric

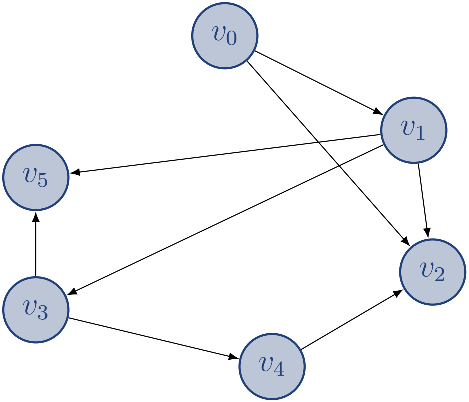


$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Adjacency Lists – Directed Unweighted Graphs



Adjacency Lists – Directed Unweighted Graphs



$$\begin{aligned}&((1,2),\\&\ (2,3,5),\\&\ (),\\&\ (4,5),\\&\ (2),\\&\ ())\end{aligned}$$

Adjacency Matrices and Lists in Python

Use 2-dimensional lists

Adjacency Matrices and Lists in Python

Use 2-dimensional lists

Matrix: Weighted

```
G = [ [ 0, 1, 7, 2, 0, 5 ],
      [ 1, 0, 0, 0, 0, 0 ],
      [ 7, 0, 0, 0, 8, 0 ],
      [ 2, 0, 0, 0, 0, 5 ],
      [ 0, 0, 8, 0, 0, 2 ],
      [ 5, 0, 0, 5, 2, 0 ] ]
```

Adjacency Matrices and Lists in Python

Use 2-dimensional lists

Matrix: Weighted

```
G = [ [ 0, 1, 7, 2, 0, 5 ],
      [ 1, 0, 0, 0, 0, 0 ],
      [ 7, 0, 0, 0, 8, 0 ],
      [ 2, 0, 0, 0, 0, 5 ],
      [ 0, 0, 8, 0, 0, 2 ],
      [ 5, 0, 0, 5, 2, 0 ] ]
```

Matrix: Unweighted

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

Adjacency Matrices and Lists in Python

Use 2-dimensional lists

Matrix: Weighted

```
G = [ [ 0, 1, 7, 2, 0, 5 ],
      [ 1, 0, 0, 0, 0, 0 ],
      [ 7, 0, 0, 0, 8, 0 ],
      [ 2, 0, 0, 0, 0, 5 ],
      [ 0, 0, 8, 0, 0, 2 ],
      [ 5, 0, 0, 5, 2, 0 ] ]
```

Matrix: Unweighted

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

List: Unweighted

```
G = [ [1,2], [0,2,3,5], [0,1,4], [1,4,5], [2,3], [1,3] ]
```

Graph Algorithms

Breadth-First and Depth-First Search

Breadth-First (BFS) and Depth-First Search (DFS)

Many applications need the systematic exploration of a given graph

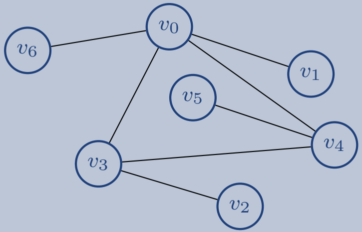
- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

Breadth-First (BFS) and Depth-First Search (DFS)

Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices

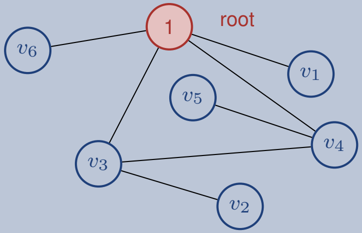


Breadth-First (BFS) and Depth-First Search (DFS)

Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices

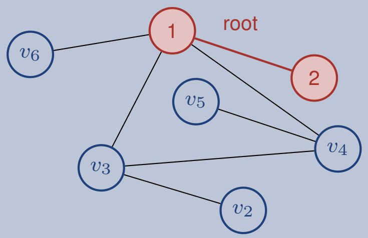


Breadth-First (BFS) and Depth-First Search (DFS)

Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices

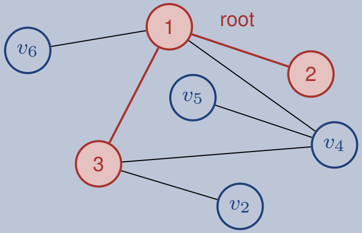


Breadth-First (BFS) and Depth-First Search (DFS)

Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices

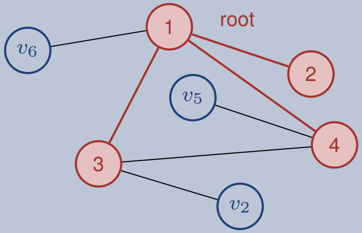


Breadth-First (BFS) and Depth-First Search (DFS)

Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices

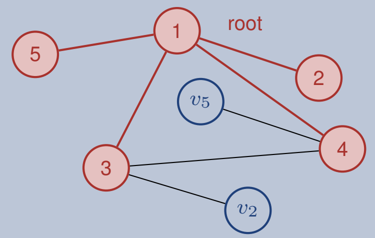


Breadth-First (BFS) and Depth-First Search (DFS)

Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices

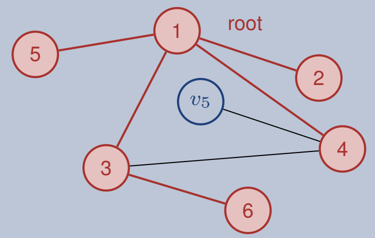


Breadth-First (BFS) and Depth-First Search (DFS)

Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices

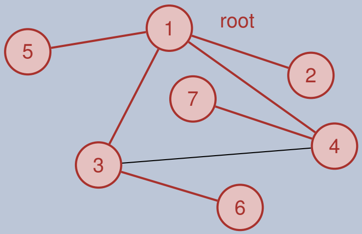


Breadth-First (BFS) and Depth-First Search (DFS)

Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices

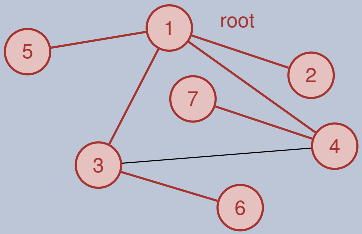


Breadth-First (BFS) and Depth-First Search (DFS)

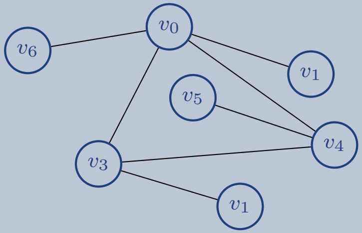
Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices



DFS: Go into the graph as deep as possible, then broadly; again break ties in favor of smaller indices

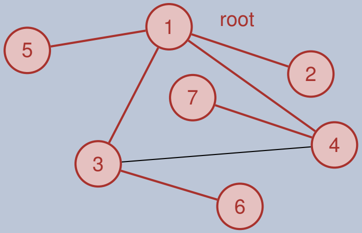


Breadth-First (BFS) and Depth-First Search (DFS)

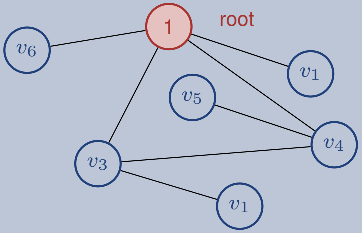
Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices



DFS: Go into the graph as deep as possible, then broadly; again break ties in favor of smaller indices

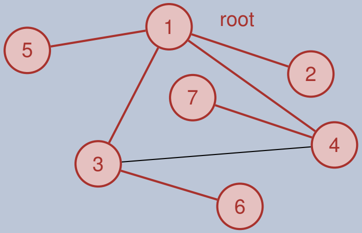


Breadth-First (BFS) and Depth-First Search (DFS)

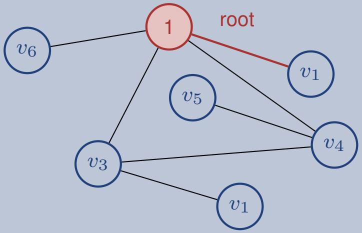
Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices



DFS: Go into the graph as deep as possible, then broadly; again break ties in favor of smaller indices

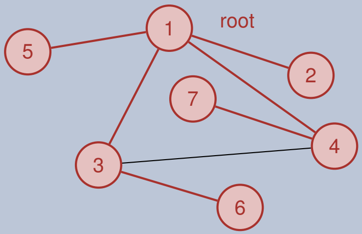


Breadth-First (BFS) and Depth-First Search (DFS)

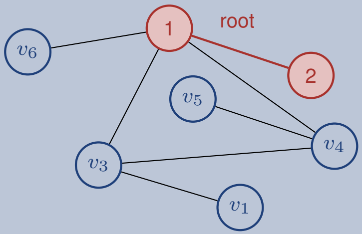
Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices



DFS: Go into the graph as deep as possible, then broadly; again break ties in favor of smaller indices

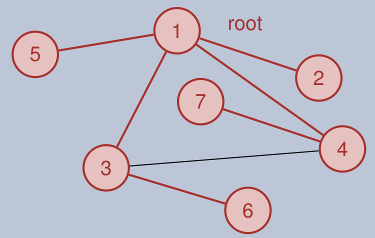


Breadth-First (BFS) and Depth-First Search (DFS)

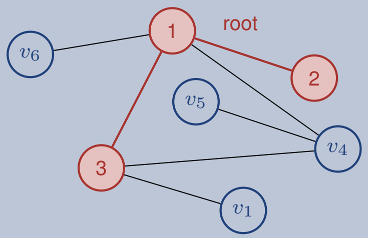
Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices



DFS: Go into the graph as deep as possible, then broadly; again break ties in favor of smaller indices

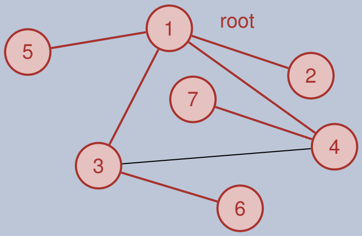


Breadth-First (BFS) and Depth-First Search (DFS)

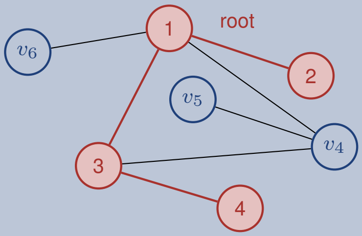
Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices



DFS: Go into the graph as deep as possible, then broadly; again break ties in favor of smaller indices

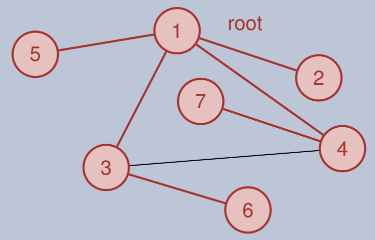


Breadth-First (BFS) and Depth-First Search (DFS)

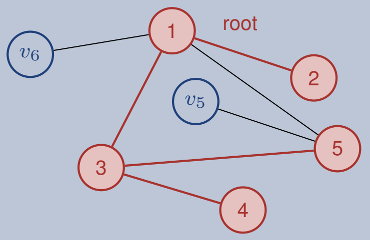
Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices



DFS: Go into the graph as deep as possible, then broadly; again break ties in favor of smaller indices

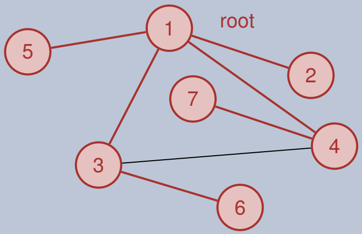


Breadth-First (BFS) and Depth-First Search (DFS)

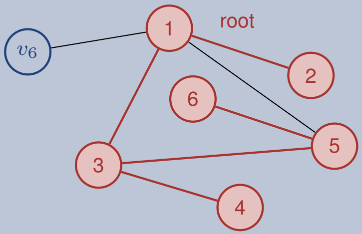
Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices



DFS: Go into the graph as deep as possible, then broadly; again break ties in favor of smaller indices

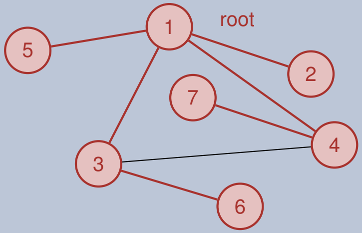


Breadth-First (BFS) and Depth-First Search (DFS)

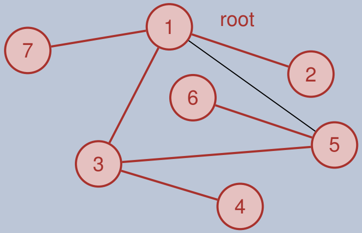
Many applications need the systematic exploration of a given graph

- Start and an arbitrary vertex
- Follow edges through graph
- Store vertices in the respective order

BFS: First go broadly and than deeply, just as with the Heap; break ties in favor of smaller indices



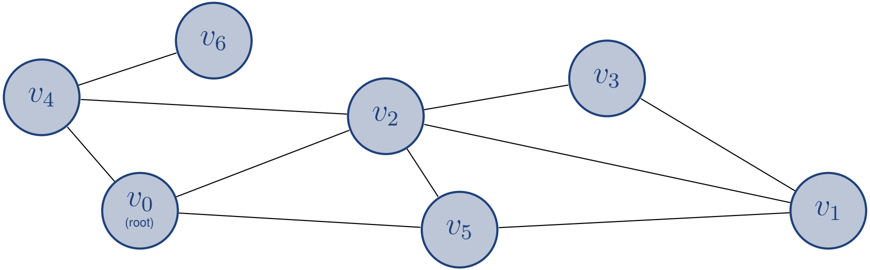
DFS: Go into the graph as deep as possible, then broadly; again break ties in favor of smaller indices



Breadth-First Search

Iteratively with a Queue

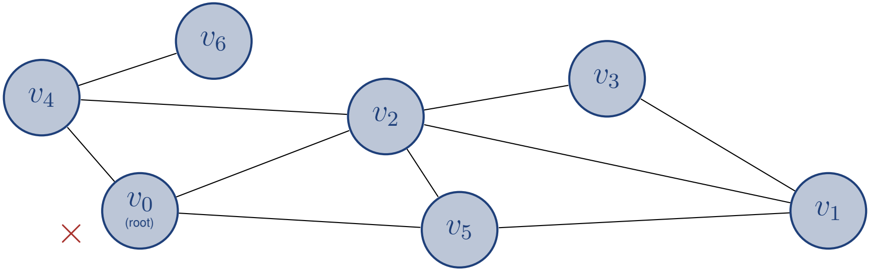
BFS with a Queue



Queue:

Output:

BFS with a Queue

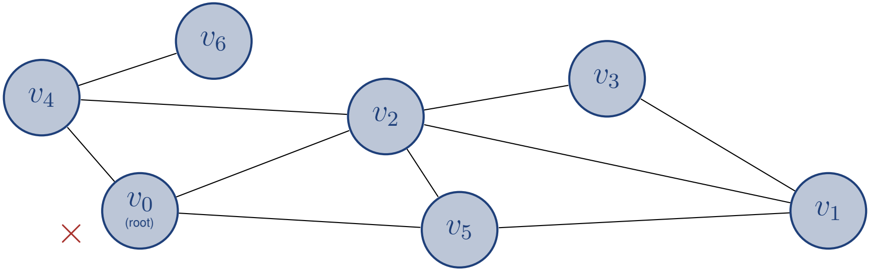


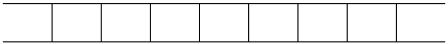
Queue:

v_0								

Output:

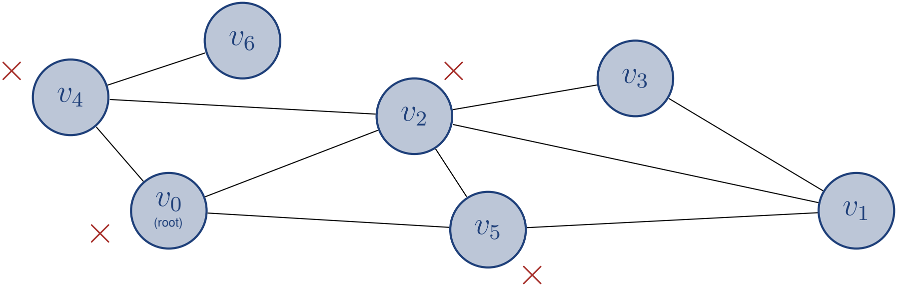
BFS with a Queue




Queue:

Output: v_0

BFS with a Queue

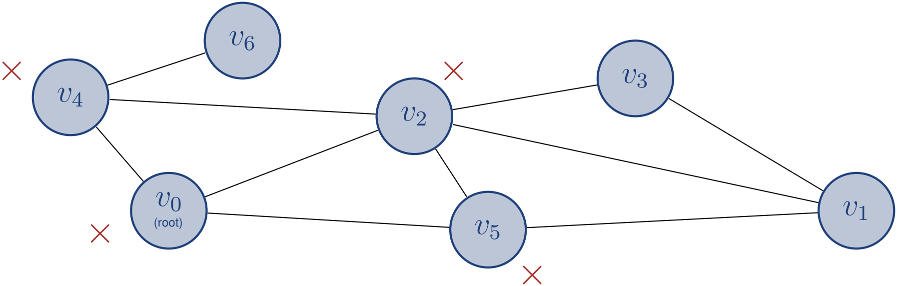


Queue:

v_2	v_4	v_5						

Output: v_0

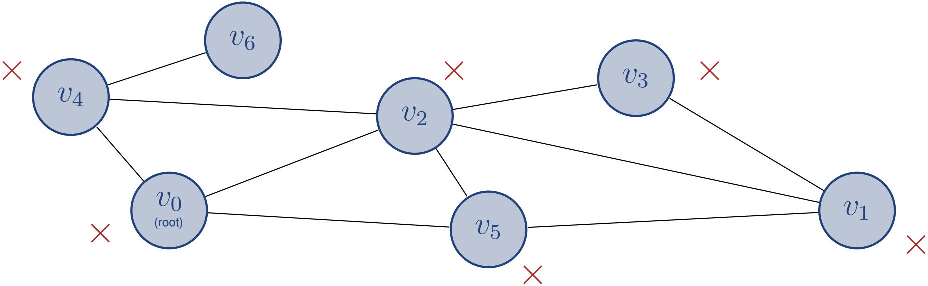
BFS with a Queue



Queue: v_4 v_5

Output: v_0 v_2

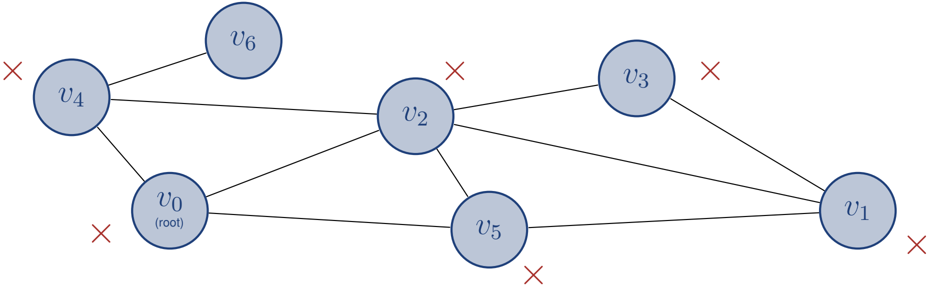
BFS with a Queue



Queue: v_4 v_5 v_1 v_3

Output: v_0 v_2

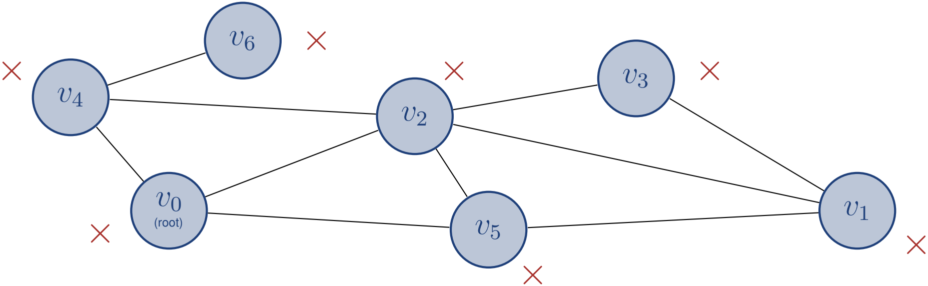
BFS with a Queue



Queue:	v_5	v_1	v_3						

Output: v_0 v_2 v_4

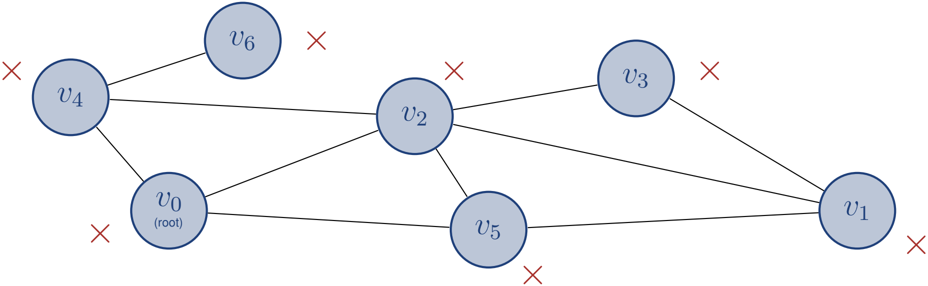
BFS with a Queue



Queue:	v_5	v_1	v_3	v_6					

Output: v_0 v_2 v_4

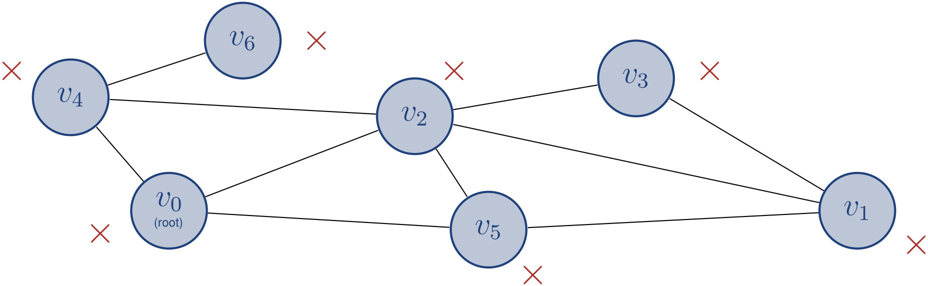
BFS with a Queue



Queue:	v_1	v_3	v_6						

Output: v_0 v_2 v_4 v_5

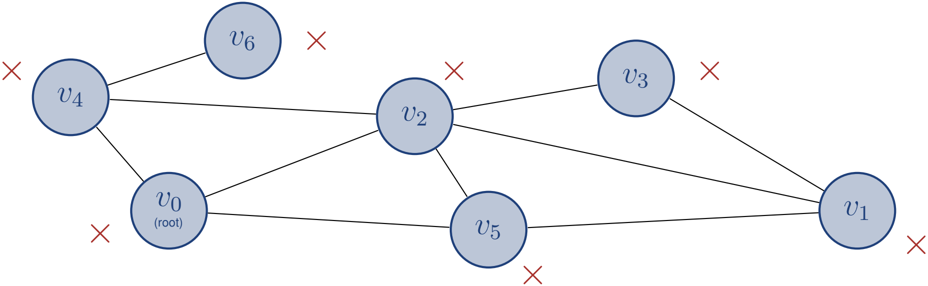
BFS with a Queue



Queue:	v_3	v_6							

Output: v_0 v_2 v_4 v_5 v_1

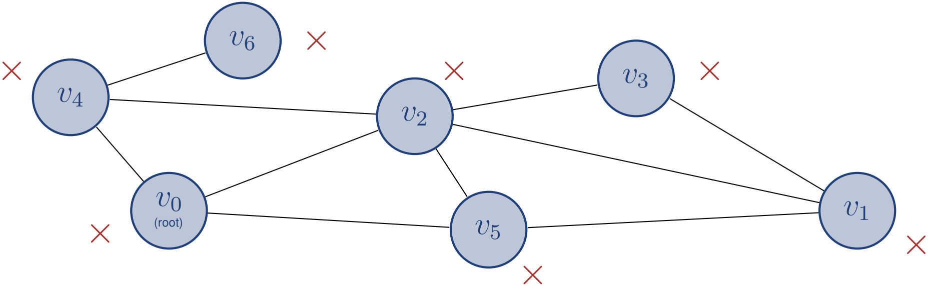
BFS with a Queue



Queue: v_6

Output: v_0 v_2 v_4 v_5 v_1 v_3

BFS with a Queue



Queue:

Output: v_0 v_2 v_4 v_5 v_1 v_3 v_6

BFS with Queue and Adjacency Matrix

```
G = [ [0,0,1,0,1,1,0], [0,0,1,1,0,1,0], [1,1,0,1,1,1,0], [0,1,1,0,0,0,0],
      [1,0,1,0,0,0,1], [1,1,1,0,0,0,0], [0,0,0,0,1,0,0] ]
queue = []
visited = [ 0 for i in range(len(G)) ]
```

BFS with Queue and Adjacency Matrix

```
G = [ [0,0,1,0,1,1,0], [0,0,1,1,0,1,0], [1,1,0,1,1,1,0], [0,1,1,0,0,0,0],
      [1,0,1,0,0,0,1], [1,1,1,0,0,0,0], [0,0,0,0,1,0,0] ]
queue = []
visited = [ 0 for i in range(len(G)) ]
```

- Consider first vertex in queue and print it

BFS with Queue and Adjacency Matrix

```
G = [ [0,0,1,0,1,1,0], [0,0,1,1,0,1,0], [1,1,0,1,1,1,0], [0,1,1,0,0,0,0],
      [1,0,1,0,0,0,1], [1,1,1,0,0,0,0], [0,0,0,0,1,0,0] ]
queue = []
visited = [ 0 for i in range(len(G)) ]
```

- Consider first vertex in queue and print it
- Add unvisited neighbors to queue

BFS with Queue and Adjacency Matrix

```
G = [ [0,0,1,0,1,1,0], [0,0,1,1,0,1,0], [1,1,0,1,1,1,0], [0,1,1,0,0,0,0],
      [1,0,1,0,0,0,1], [1,1,1,0,0,0,0], [0,0,0,0,1,0,0] ]
queue = []
visited = [ 0 for i in range(len(G)) ]
```

- Consider first vertex in queue and print it
- Add unvisited neighbors to queue
- `visited` stores which vertices have been visited

BFS with Queue and Adjacency Matrix

```
G = [ [0,0,1,0,1,1,0], [0,0,1,1,0,1,0], [1,1,0,1,1,1,0], [0,1,1,0,0,0,0],
      [1,0,1,0,0,0,1], [1,1,1,0,0,0,0], [0,0,0,0,1,0,0] ]
queue = []
visited = [ 0 for i in range(len(G)) ]
```

- Consider first vertex in queue and print it
- Add unvisited neighbors to queue
- `visited` stores which vertices have been visited
- Repeat as long as queue is not empty

Exercise – BFS with Queue and Adjacency Matrix

Implement BFS

- as a Python function
- with a 2-dimensional list as parameter
- using a queue
- and an adjacency matrix

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

BFS with Queue and a Adjacency Matrix

```
def BFS(G):
    queue = []
    visited = [ 0 for i in range(len(G)) ]
    queue.append(0)
    visited[0] = 1
    while len(queue) > 0:
        current = queue.pop(0)
        print(current, end=" ")
        for j in range(len(G)):
            if G[current][j] == 1 and visited[j] == 0:
                visited[j] = 1
                queue.append(j)
BFS([ [0,0,1,0,1,1,0], [0,0,1,1,0,1,0], [1,1,0,1,1,1,0], [0,1,1,0,0,0,0],
      [1,0,1,0,0,0,1], [1,1,1,0,0,0,0], [0,0,0,0,1,0,0] ])
```

Exercise – BFS with Queue and Adjacency List

Implement BFS

- as a Python function
- with a 2-dimensional list as parameter
- using a queue
- and an adjacency list

```
G = [ [2,4,5], [2,3,5], [0,1,3,4,5], [1,2],
      [0,2,6], [0,1,2], [4] ]
```

BFS with Queue and a Adjacency List

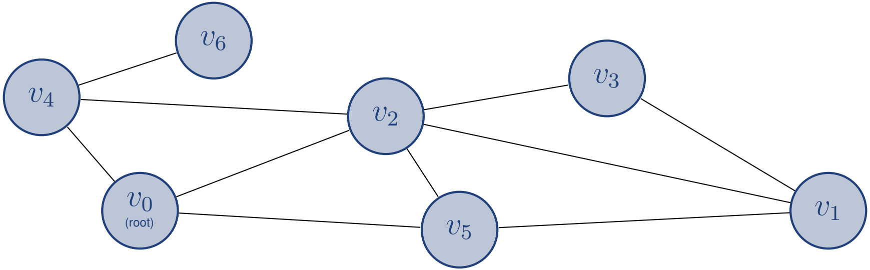
```
def BFS(G):
    queue = []
    visited = [ 0 for i in range(len(G)) ]
    queue.append(0)
    visited[0] = 1
    while len(queue) > 0:
        current = queue.pop(0)
        print(current, end=" ")
        for j in G[current]:
            if visited[j] == 0:
                visited[j] = 1
                queue.append(j)

BFS([ [2,4,5], [2,3,5], [0,1,3,4,5], [1,2], [0,2,6], [0,1,2], [4] ])
```

Depth-First Search

Iteratively with a Stack

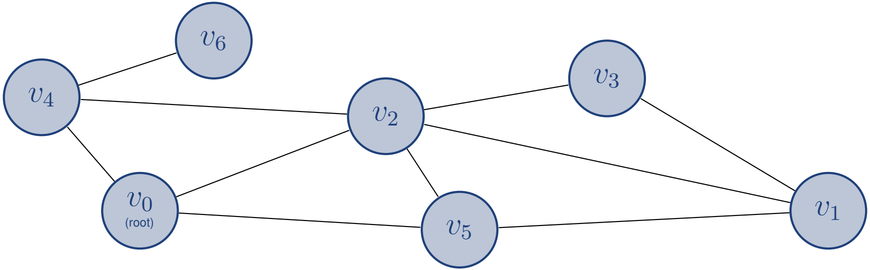
DFS with a Stack



Stack:

Output:

DFS with a Stack

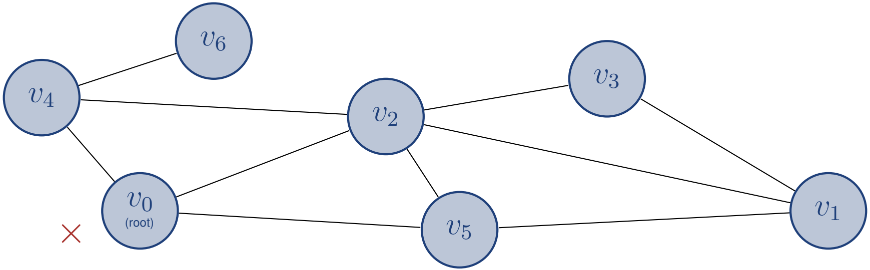


Stack:

v_0								

Output:

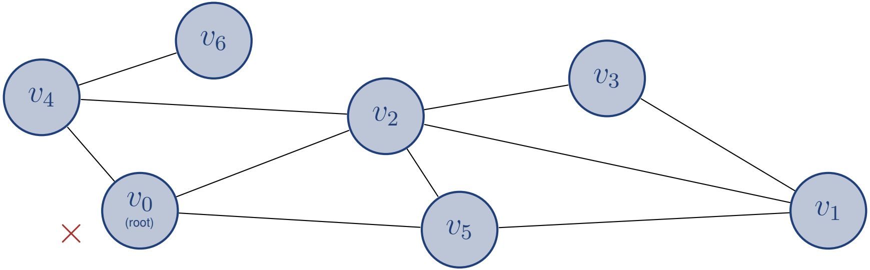
DFS with a Stack



Stack:

Output: v_0

DFS with a Stack

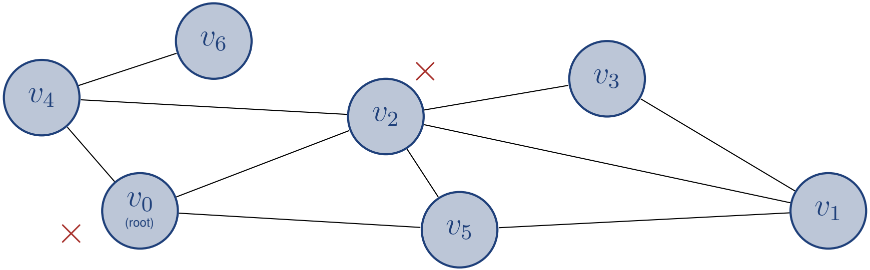


Stack:

v_5	v_4	v_2						

Output: v_0

DFS with a Stack

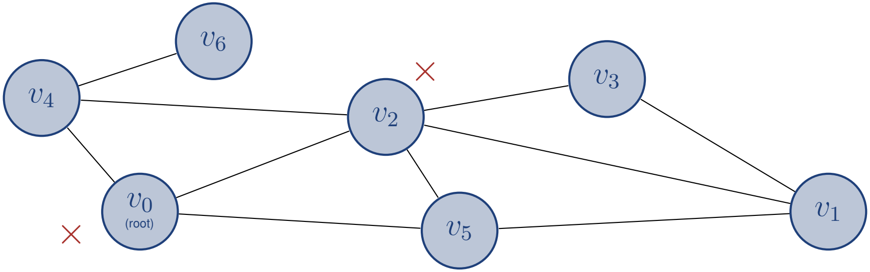


Stack:

v_5	v_4							

Output: v_0 v_2

DFS with a Stack

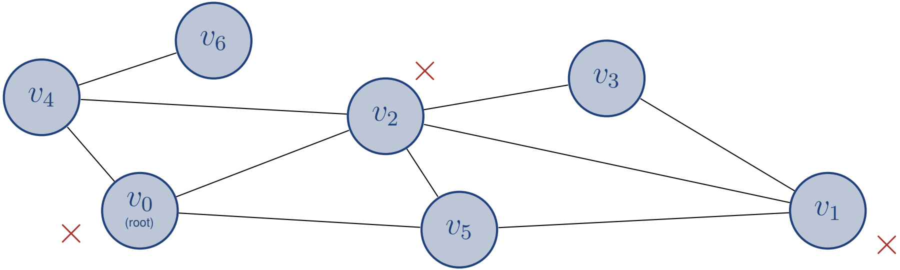


Stack:

v_5	v_4	v_5	v_4	v_3	v_1			

Output: v_0 v_2

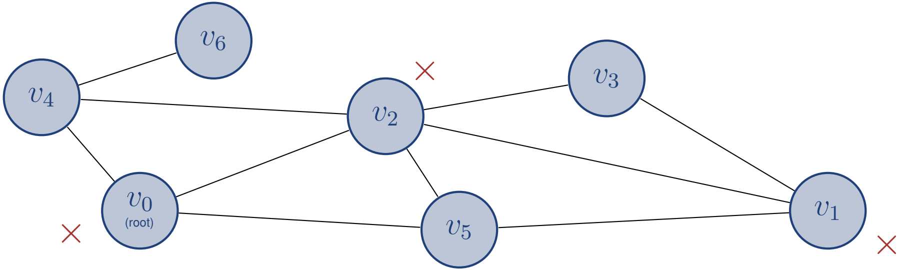
DFS with a Stack



Stack:	v_5	v_4	v_5	v_4	v_3				

Output: v_0 v_2 v_1

DFS with a Stack

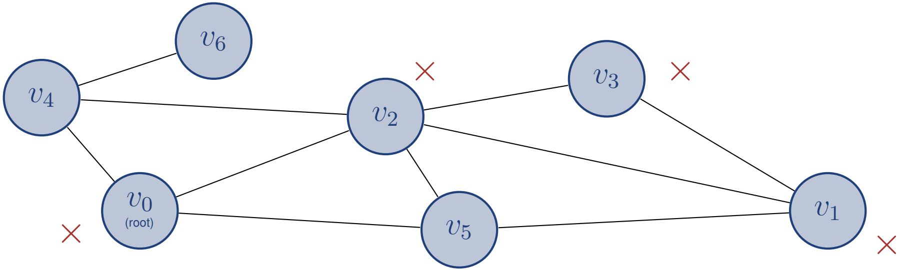


Stack:

v_5	v_4	v_5	v_4	v_3	v_5	v_3		

Output: v_0 v_2 v_1

DFS with a Stack

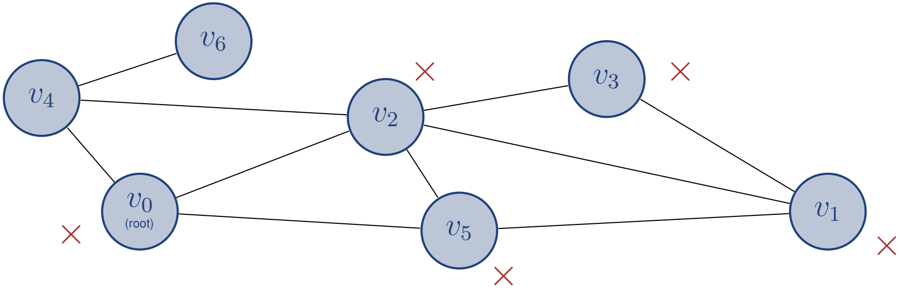


Stack:

v_5	v_4	v_5	v_4	v_3	v_5			

Output: v_0 v_2 v_1 v_3

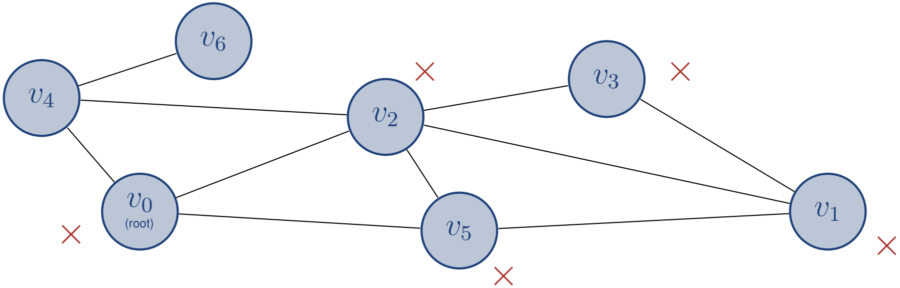
DFS with a Stack



Stack:	v_5	v_4	v_5	v_4	v_3				

Output: v_0 v_2 v_1 v_3 v_5

DFS with a Stack

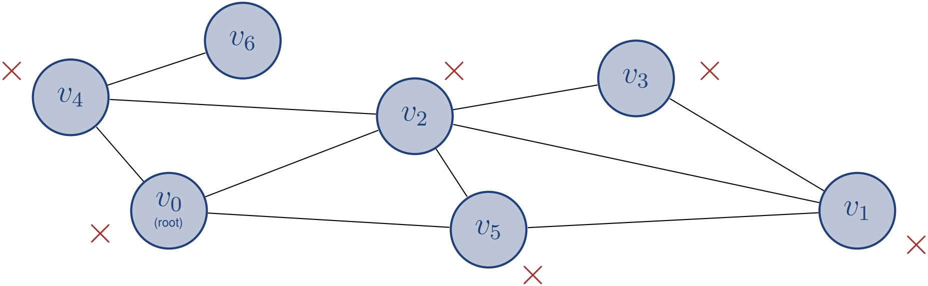


v_5	v_4	v_5	v_4					

Stack: v_5 v_4 v_5 v_4

Output: v_0 v_2 v_1 v_3 v_5

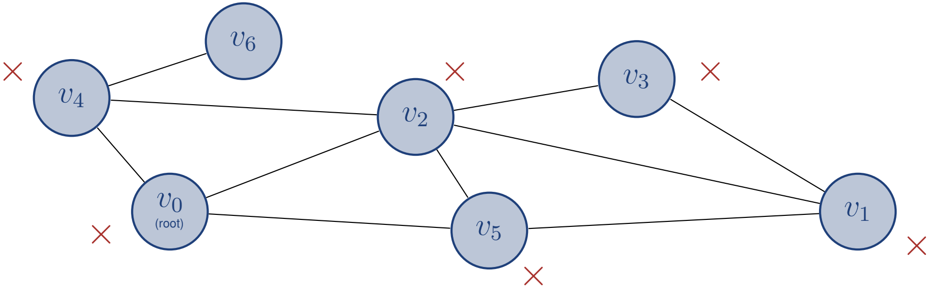
DFS with a Stack



Stack:	v_5	v_4	v_5						

Output: v_0 v_2 v_1 v_3 v_5 v_4

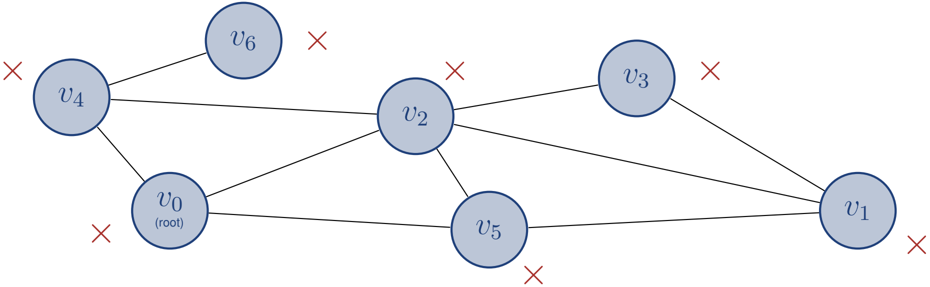
DFS with a Stack



Stack:	v_5	v_4	v_5	v_6					

Output: v_0 v_2 v_1 v_3 v_5 v_4

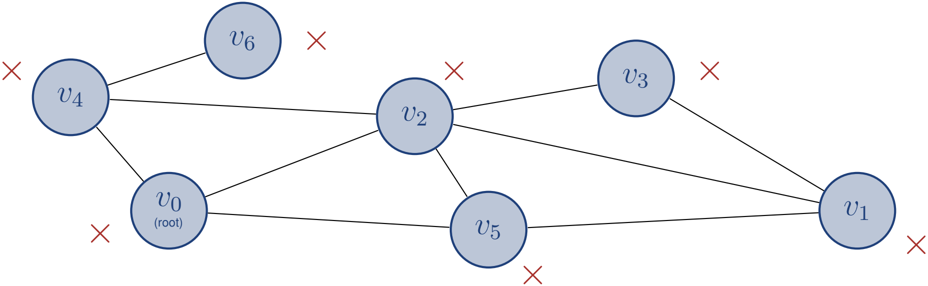
DFS with a Stack



Stack:	v_5	v_4	v_5						

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

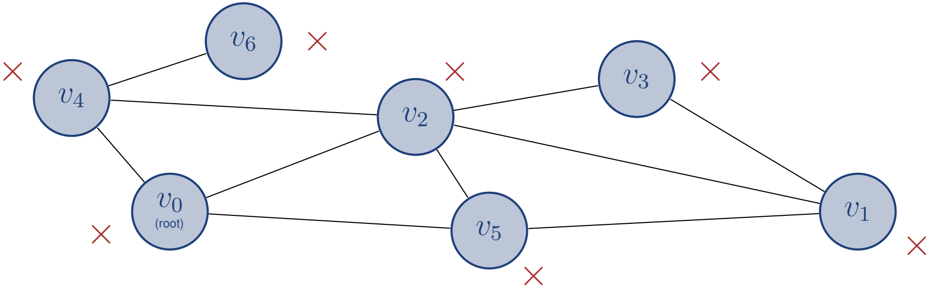
DFS with a Stack



Stack:	v_5	v_4							

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

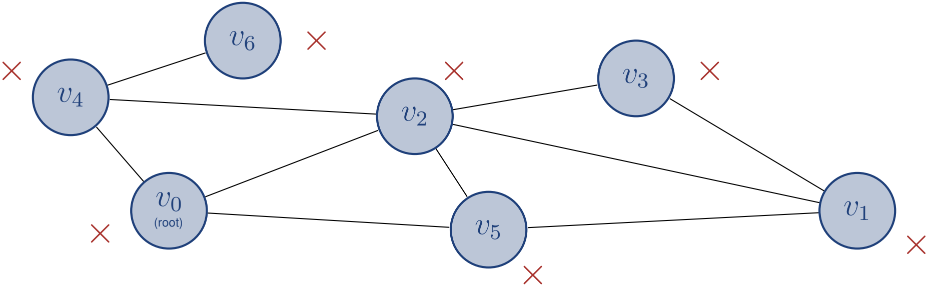
DFS with a Stack



Stack:	v_5								

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

DFS with a Stack



Stack:

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

DFS with Stack and Adjacency Matrix

```
G = [ [0, 0, 1, 0, 1, 1, 0],
      [0, 0, 1, 1, 0, 1, 0],
      [1, 1, 0, 1, 1, 1, 0],
      [0, 1, 1, 0, 0, 0, 0],
      [1, 0, 1, 0, 0, 0, 1],
      [1, 1, 1, 0, 0, 0, 0],
      [0, 0, 0, 0, 1, 0, 0] ]
stack = []
visited = [ 0 for i in range(len(G)) ]
```

DFS with Stack and Adjacency Matrix

```
G = [ [0, 0, 1, 0, 1, 1, 0],
      [0, 0, 1, 1, 0, 1, 0],
      [1, 1, 0, 1, 1, 1, 0],
      [0, 1, 1, 0, 0, 0, 0],
      [1, 0, 1, 0, 0, 0, 1],
      [1, 1, 1, 0, 0, 0, 0],
      [0, 0, 0, 0, 1, 0, 0] ]
stack = []
visited = [ 0 for i in range(len(G)) ]
```

- Consider first vertex in stack and print it

DFS with Stack and Adjacency Matrix

```
G = [ [0, 0, 1, 0, 1, 1, 0],
      [0, 0, 1, 1, 0, 1, 0],
      [1, 1, 0, 1, 1, 1, 0],
      [0, 1, 1, 0, 0, 0, 0],
      [1, 0, 1, 0, 0, 0, 1],
      [1, 1, 1, 0, 0, 0, 0],
      [0, 0, 0, 0, 1, 0, 0] ]
stack = []
visited = [ 0 for i in range(len(G)) ]
```

- Consider first vertex in stack and print it
- Add unvisited neighbors to stack

DFS with Stack and Adjacency Matrix

```
G = [ [0, 0, 1, 0, 1, 1, 0],
      [0, 0, 1, 1, 0, 1, 0],
      [1, 1, 0, 1, 1, 1, 0],
      [0, 1, 1, 0, 0, 0, 0],
      [1, 0, 1, 0, 0, 0, 1],
      [1, 1, 1, 0, 0, 0, 0],
      [0, 0, 0, 0, 1, 0, 0] ]
stack = []
visited = [ 0 for i in range(len(G)) ]
```

- Consider first vertex in stack and print it
- Add unvisited neighbors to stack
- `visited` stores which vertices have been visited

DFS with Stack and Adjacency Matrix

```
G = [ [0, 0, 1, 0, 1, 1, 0],
      [0, 0, 1, 1, 0, 1, 0],
      [1, 1, 0, 1, 1, 1, 0],
      [0, 1, 1, 0, 0, 0, 0],
      [1, 0, 1, 0, 0, 0, 1],
      [1, 1, 1, 0, 0, 0, 0],
      [0, 0, 0, 0, 1, 0, 0] ]
stack = []
visited = [ 0 for i in range(len(G)) ]
```

- Consider first vertex in stack and print it
- Add unvisited neighbors to stack
- `visited` stores which vertices have been visited
- Repeat as long as stack is not empty

Exercise – DFS with Stack and Adjacency Matrix

Implement DFS

- as a Python function
- with a 2-dimensional list as parameter
- using a stack
- and an adjacency matrix

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

DFS with Stack and Adjacency Matrix

```python
def DFS(G):
    stack = []
    visited = [ 0 for i in range(len(G)) ]
    stack.append(0)
    while len(stack) > 0:
        current = stack.pop()
          if visited[current] == 0:
              visited[current] = 1
              print(current, end=" ")
              for j in reversed(range(len(G))):
                  if G[current][j] == 1 and visited[j] == 0:
                      stack.append(j)

DFS( [ [0,0,1,0,1,1,0], [0,0,1,1,0,1,0], [1,1,0,1,1,1,0], [0,1,1,0,0,0,0],
      [1,0,1,0,0,0,1], [1,1,1,0,0,0,0], [0,0,0,0,1,0,0] ] )
```

Exercise – DFS with Stack and Adjacency List

Implement DFS

- as a Python function
- with a 2-dimensional list as parameter
- using a stack
- and an adjacency matrix

```
G = [ [2,4,5], [2,3,5], [0,1,3,4,5], [1,2], [0,2,6], [0,1,2], [4] ]
```

DFS with Stack and Adjacency List

```
def DFS(G):
    stack = []
    visited = [ 0 for i in range(len(G)) ]
    stack.append(0)
    while len(stack) > 0:
        current = stack.pop()
        if visited[current] == 0:
            visited[current] = 1
            print(current, end=" ")
            for j in reversed(G[current]):
                if visited[j] == 0:
                    stack.append(j)
DFS([ [2,4,5], [2,3,5], [0,1,3,4,5], [1,2], [0,2,6], [0,1,2], [4] ])
```

Depth-First Search

Recursively

Recursive DFS

- Global list `visited`

Recursive DFS

- Global list `visited`
- Function `DFS`, which is called recursively

Recursive DFS

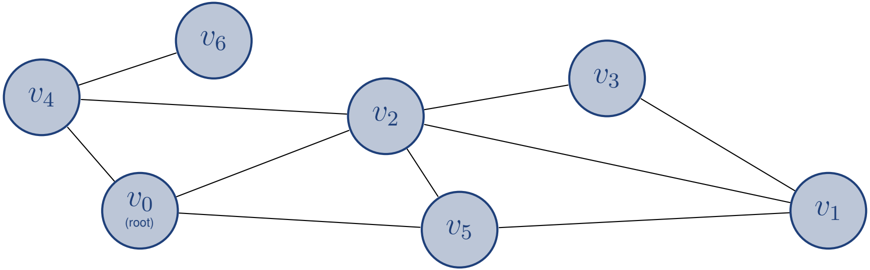
- Global list `visited`
- Function `DFS`, which is called recursively
- Two parameters
  1. graph `G`
  2. Start vertex `current`

Recursive DFS

- Global list `visited`
- Function `DFS`, which is called recursively
- Two parameters
    1. graph `G`
    2. Start vertex `current`

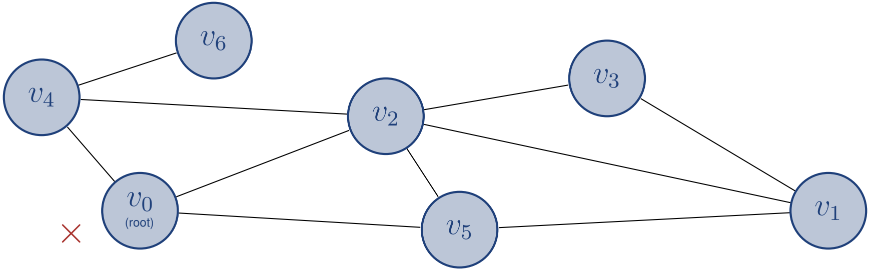
```
visited = [ 0 for i in range(len(G)) ]
def DFS(G, current):
    visited[current] = 1
    print(current, end=" ")
    for i in range(len(G)):
        if G[current][i] == 1 and visited[i] == 0:
            DFS(G, i)
```

Recursive DFS



Output:

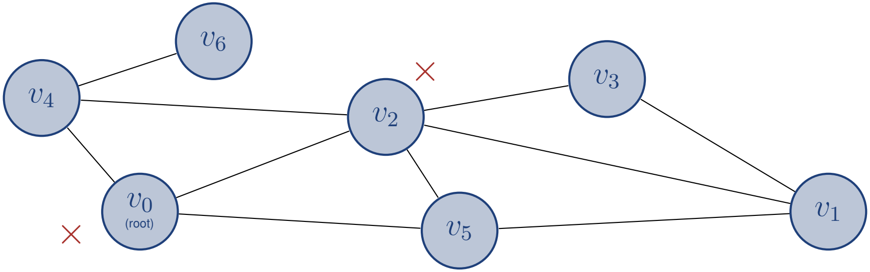
Recursive DFS



`DFS(G,0)`

Output: v_0

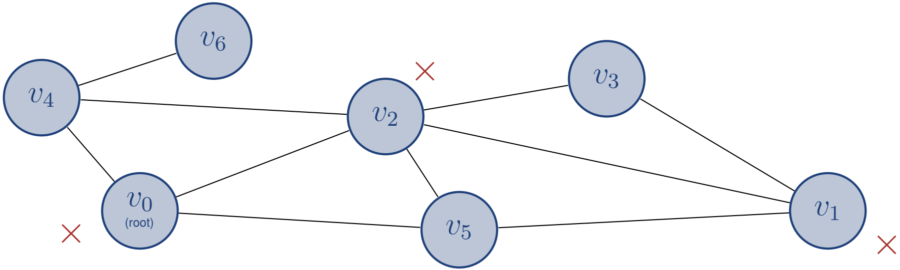
Recursive DFS



`DFS(G,0)` → `DFS(G,2)`

Output: v_0 v_2

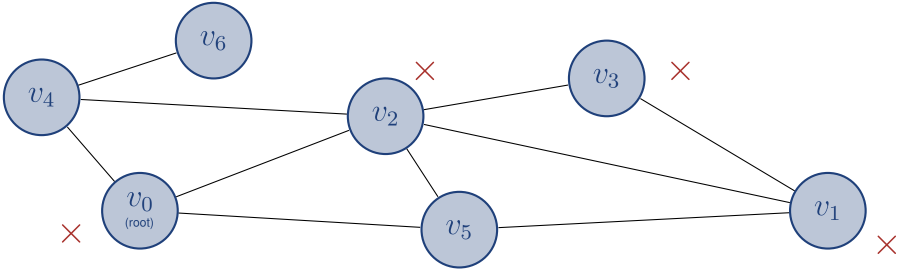
Recursive DFS



`DFS(G,0)` → `DFS(G,2)` → `DFS(G,1)`

Output: v_0 v_2 v_1

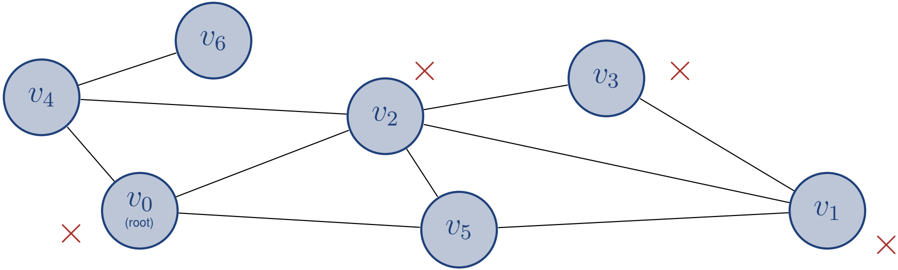
Recursive DFS



`DFS(G,0)` → `DFS(G,2)` → `DFS(G,1)` → `DFS(G,3)`

Output: v_0 v_2 v_1 v_3

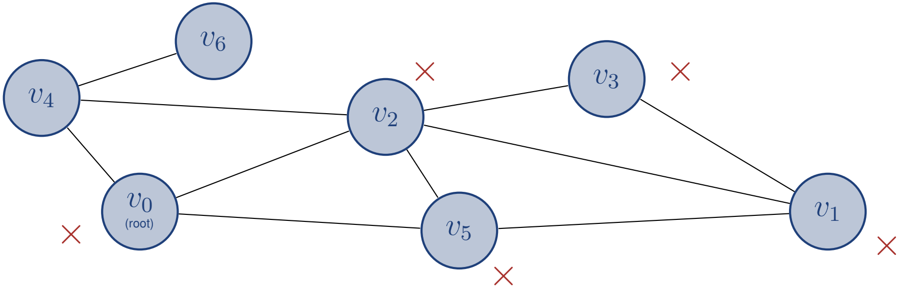
Recursive DFS



`DFS(G,0)` → `DFS(G,2)` → `DFS(G,1)`

Output: v_0 v_2 v_1 v_3

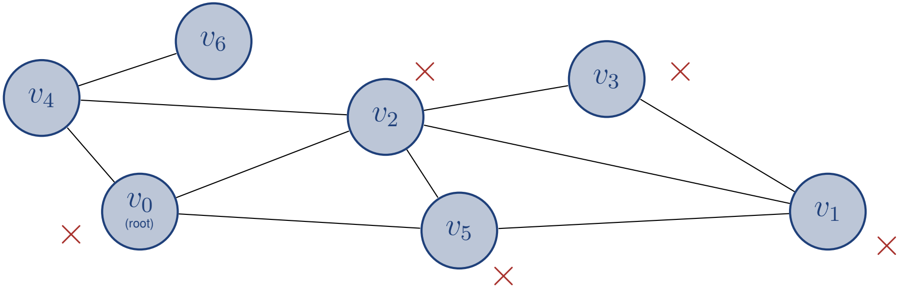
Recursive DFS



`DFS(G,0)` → `DFS(G,2)` → `DFS(G,1)` → `DFS(G,5)`

Output: v_0 v_2 v_1 v_3 v_5

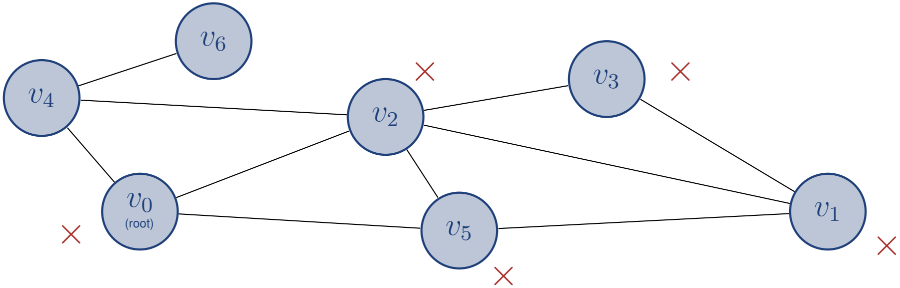
Recursive DFS



`DFS(G,0)` → `DFS(G,2)` → `DFS(G,1)`

Output: v_0 v_2 v_1 v_3 v_5

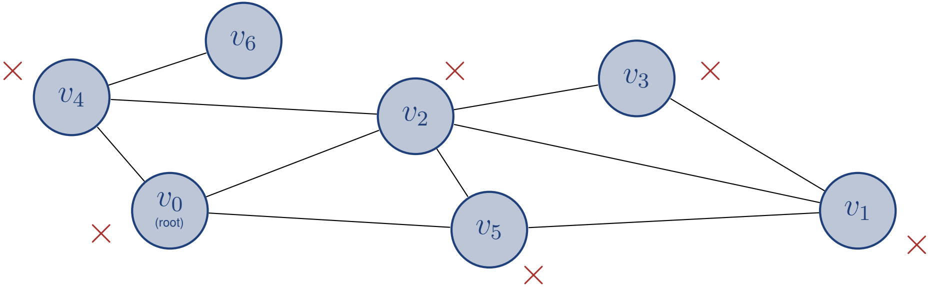
Recursive DFS



DFS(G,0) → DFS(G,2)

Output: v_0 v_2 v_1 v_3 v_5

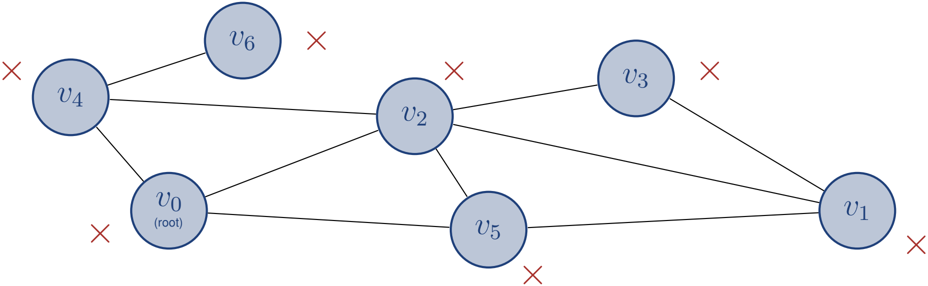
Recursive DFS



`DFS(G,0)` → `DFS(G,2)` → `DFS(G,4)`

Output: v_0 v_2 v_1 v_3 v_5 v_4

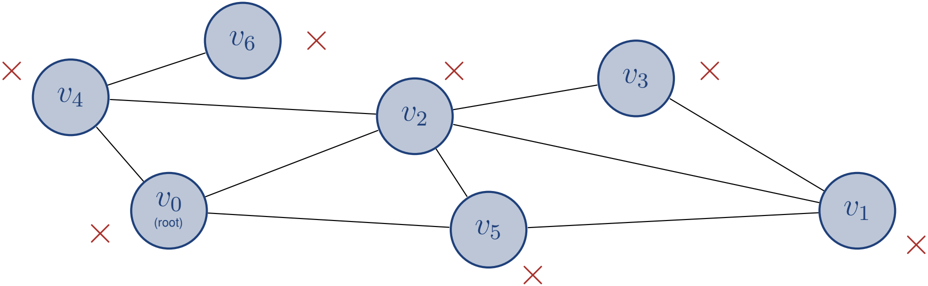
Recursive DFS




DFS(G,0) → DFS(G,2) → DFS(G,4) → DFS(G,6)

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

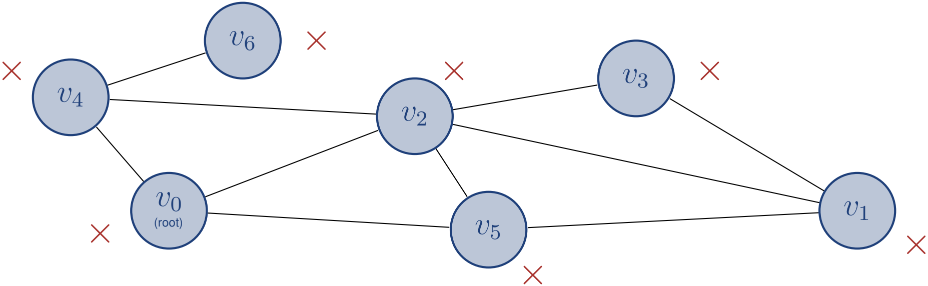
Recursive DFS



DFS(G,0) → DFS(G,2) → DFS(G,4)

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

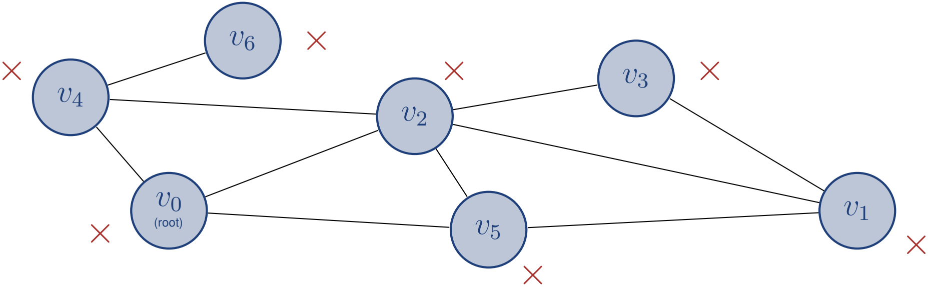
Recursive DFS



`DFS(G,0)` → `DFS(G,2)`

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

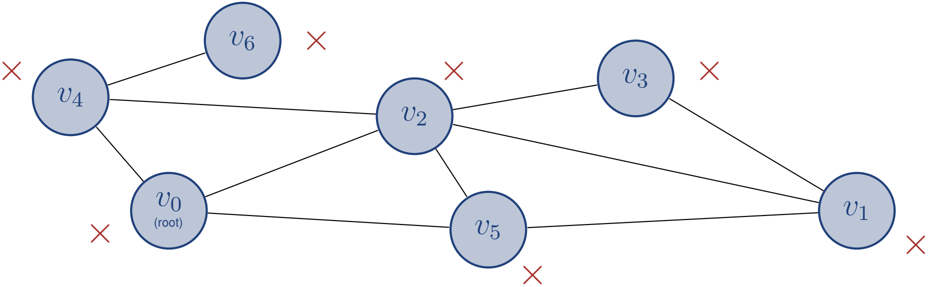
Recursive DFS



`DFS(G,0)`

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

Recursive DFS



Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

Recursive DFS

Applications

Applications

Is graph connected?

Applications

Is graph connected?

⇨ DFS from arbitrary vertex; are all vertices visited when done?

Applications

Is graph connected?

⇨ DFS from arbitrary vertex; are all vertices visited when done?

Is vertex w reachable from vertex v?

Applications

Is graph connected?

⇨ DFS from arbitrary vertex; are all vertices visited when done?

Is vertex w reachable from vertex v?

⇨ DFS from v; is w visited when done?

Applications

Is graph connected?

⇨ DFS from arbitrary vertex; are all vertices visited when done?

Is vertex w reachable from vertex v?

⇨ DFS from v; is w visited when done?

Is a graph 2-colorable?

Applications

Is graph connected?

⇨ DFS from arbitrary vertex; are all vertices visited when done?

Is vertex w reachable from vertex v?

⇨ DFS from v; is w visited when done?

Is a graph 2-colorable?

⇨ DFS from arbitrary vertex and color levels differently

Applications

Is graph connected?

⇨ DFS from arbitrary vertex; are all vertices visited when done?

Is vertex w reachable from vertex v?

⇨ DFS from v; is w visited when done?

Is a graph 2-colorable?

⇨ DFS from arbitrary vertex and color levels differently

Does a graph contain a cycle?

Applications

Is graph connected?

⇨ DFS from arbitrary vertex; are all vertices visited when done?

Is vertex w reachable from vertex v?

⇨ DFS from v; is w visited when done?

Is a graph 2-colorable?

⇨ DFS from arbitrary vertex and color levels differently

Does a graph contain a cycle?

⇨ DFS from arbitrary vertex; is there a back edge?

Recursive DFS

Graph Coloring

Graph Coloring

Consider arbitrary graph

- Can it be colored with two colors?
- Connected vertices (“neighbors”) have different color

Graph Coloring

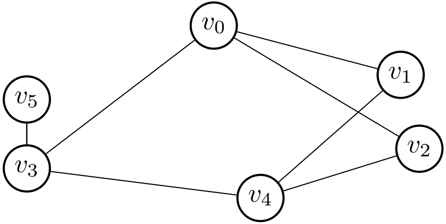
Consider arbitrary graph

- Can it be colored with two colors?
- Connected vertices (“neighbors”) have different color
- Compute recursively

Graph Coloring

Consider arbitrary graph

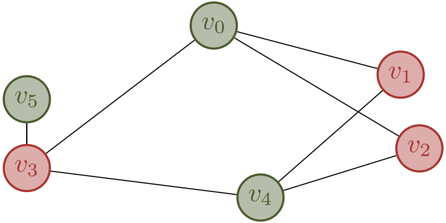
- Can it be colored with two colors?
- Connected vertices (“neighbors”) have different color
- Compute recursively



Graph Coloring

Consider arbitrary graph

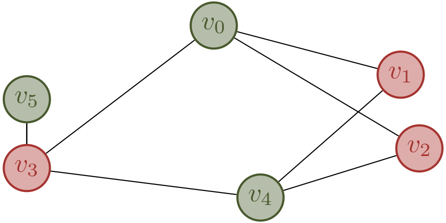
- Can it be colored with two colors?
- Connected vertices (“neighbors”) have different color
- Compute recursively



Graph Coloring

Consider arbitrary graph

- Can it be colored with two colors?
- Connected vertices (“neighbors”) have different color
- Compute recursively

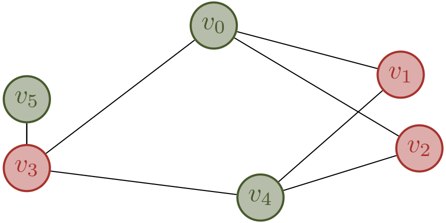


- List `color` instead of `visited`

Graph Coloring

Consider arbitrary graph

- Can it be colored with two colors?
- Connected vertices (“neighbors”) have different color
- Compute recursively

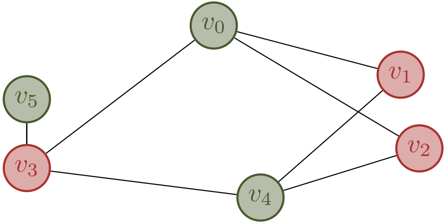


- List `color` instead of `visited`
- 0: not yet visited

Graph Coloring

Consider arbitrary graph

- Can it be colored with two colors?
- Connected vertices (“neighbors”) have different color
- Compute recursively

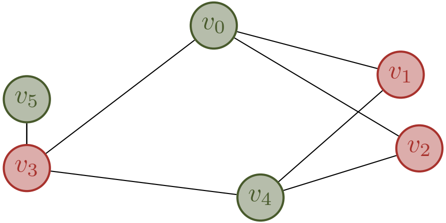


- List `color` instead of `visited`
- 0: not yet visited
- 1: colored green

Graph Coloring

Consider arbitrary graph

- Can it be colored with two colors?
- Connected vertices (“neighbors”) have different color
- Compute recursively



- List `color` instead of `visited`
- 0: not yet visited
- 1: colored green
- 2: colored red

Graph Coloring

We use recursive DFS

- All neighbors of `current` get a color different from that of `current`

Graph Coloring

We use recursive DFS

- All neighbors of `current` get a color different from that of `current`
- If neighbor already has same color as `current`, coloring is invalid

Graph Coloring

We use recursive DFS

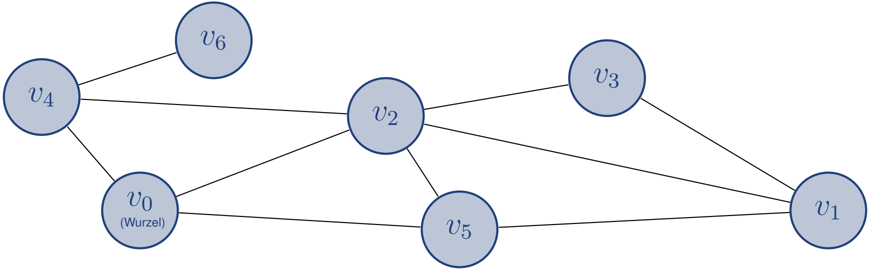
- All neighbors of `current` get a color different from that of `current`
- If neighbor already has same color as `current`, coloring is invalid

```python
def coloring(G, current):
    for i in range(len(G)):
        if G[current][i] == 1 and color[i] == 0:
            color[i] = 3 - color[current]
            coloring(G, i)
        elif G[current][i] == 1 and color[i] == color[current]:
            print("Coloring impossible.")
            return
```

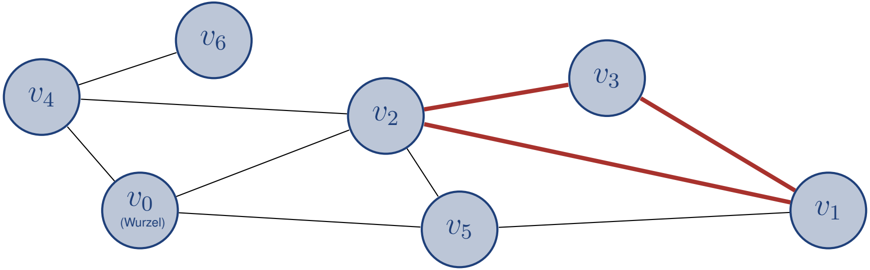
Recursive DFS

Finding Cycles

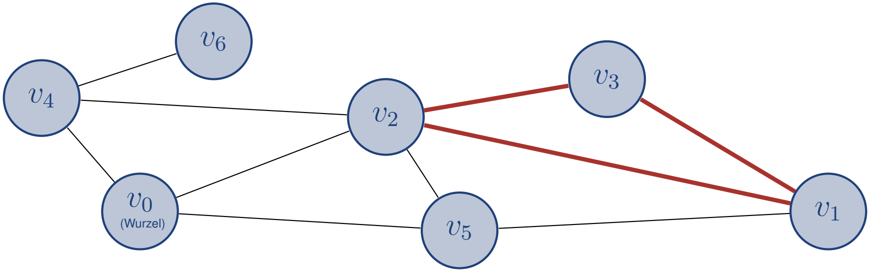
Finding Cycles



Finding Cycles

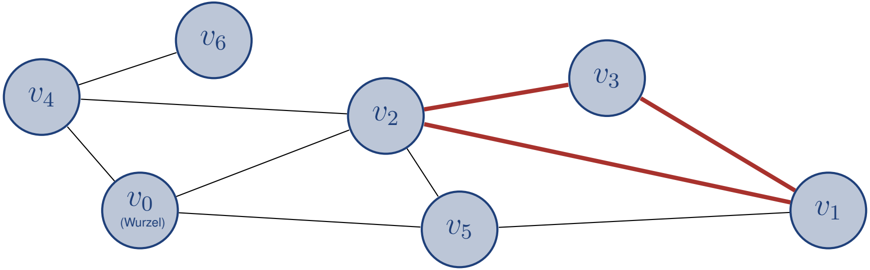


Finding Cycles



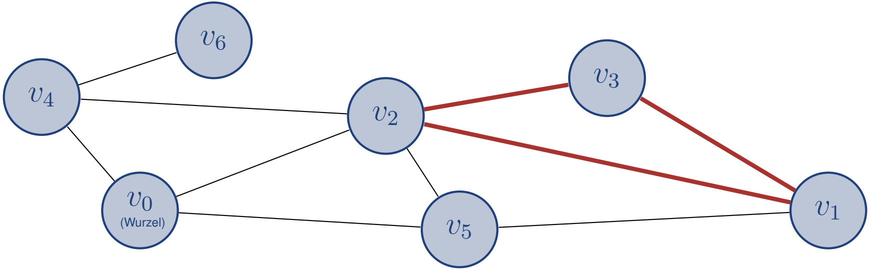
- DFS computes this

Finding Cycles



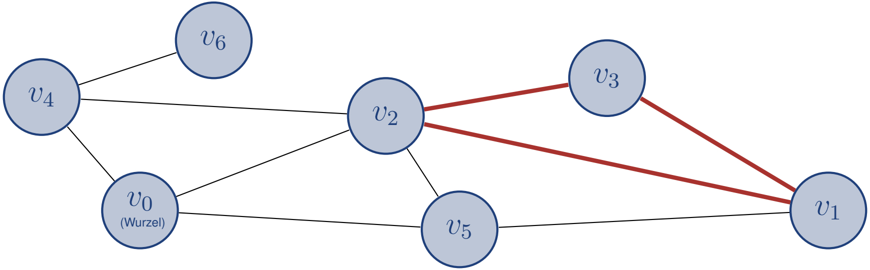
- DFS computes this
- Traverse graph as before

Finding Cycles



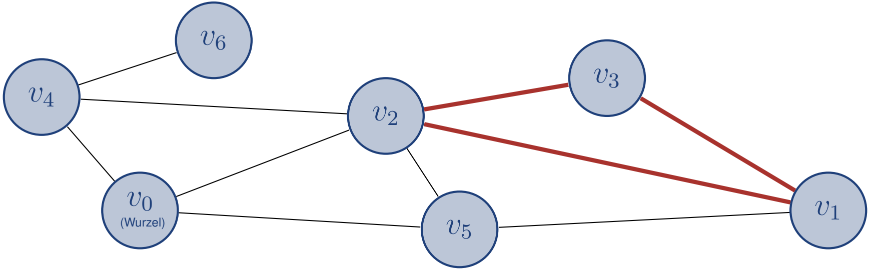
- DFS computes this
- Traverse graph as before
- Is there an edge to a vertex we already visited?

Finding Cycles



- DFS computes this
- Traverse graph as before
- Is there an edge to a vertex we already visited?
- Back-Edge

Finding Cycles



- DFS computes this
- Traverse graph as before
- Is there an edge to a vertex we already visited?
- Back-Edge
- **Attention:** Single edge is not a cycle

Finding Cycles

Compute whether graph contains a cycle

Finding Cycles

Compute whether graph contains a cycle

Extend DFS such that parent is considered

```
def find_cycle(G, current, parent):
    visited[current] = 1
    print(current, end=" ")
    for i in range(len(G)):
        if G[current][i] == 1 and visited[i] == 0:
            find_cycle(G, i, current)
        elif G[current][i] == 1 and visited[i] == 1 and i != parent:
            print("Found cycle.")
            return
```

Thanks for your attention