# Programmieren und Problemlösen

## Graphen und Graph-Algorithmen

Manuela Fischer und Dennis Komm

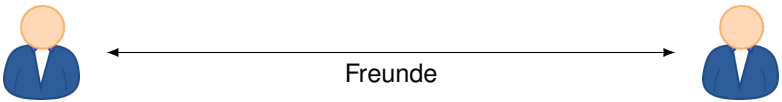
# Graphen

## Suche in Netzwerken

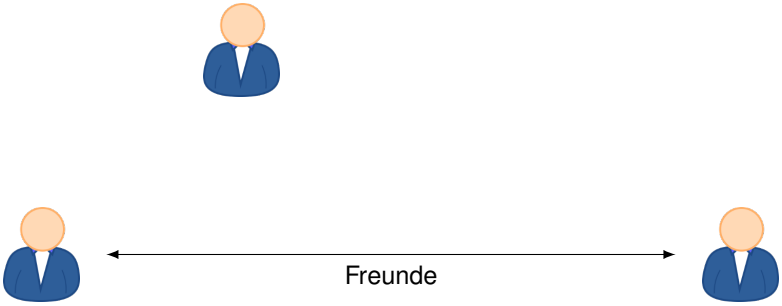
# Soziales Netzwerk

# Soziales Netzwerk

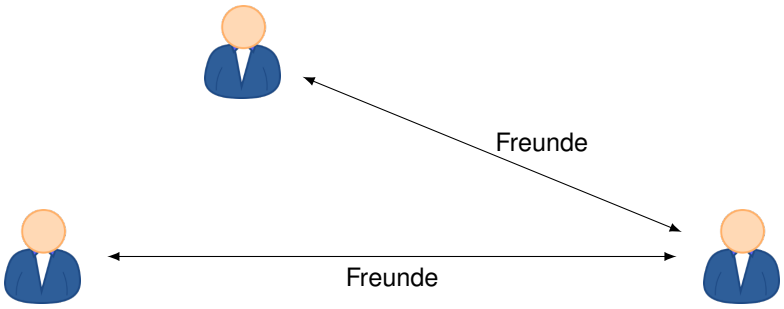
# Soziales Netzwerk



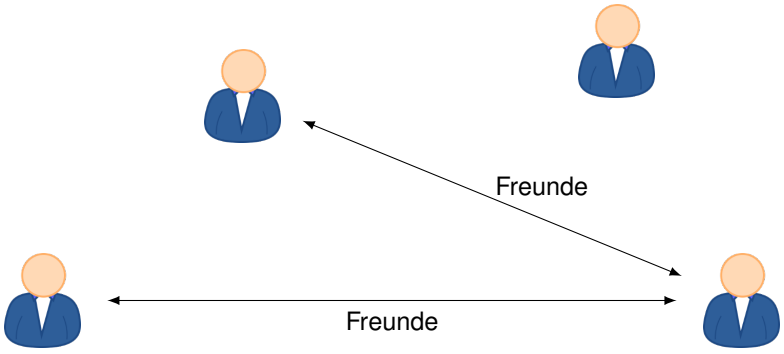
# Soziales Netzwerk



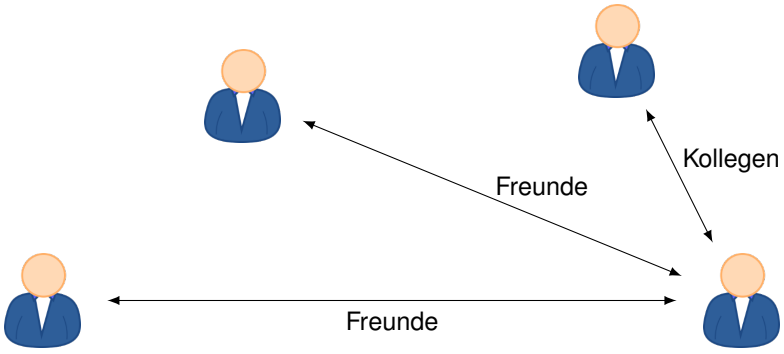
# Soziales Netzwerk



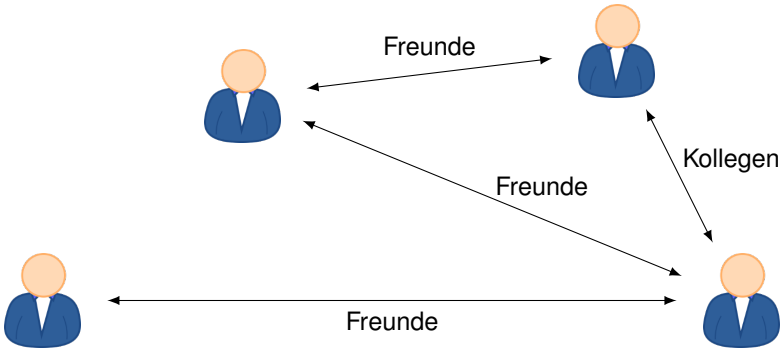
# Soziales Netzwerk



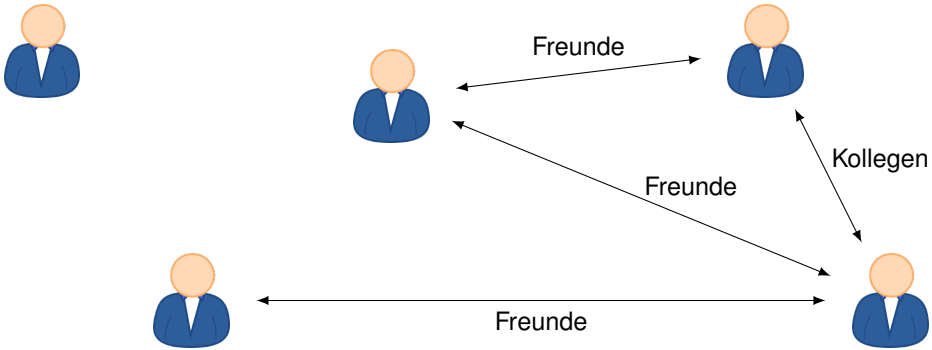
# Soziales Netzwerk



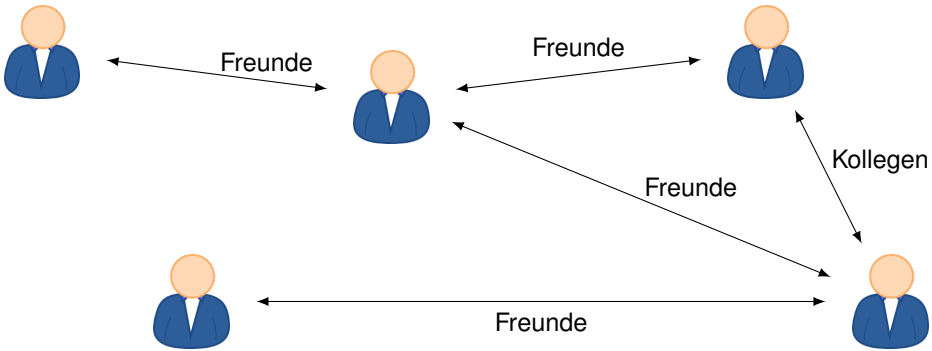
# Soziales Netzwerk



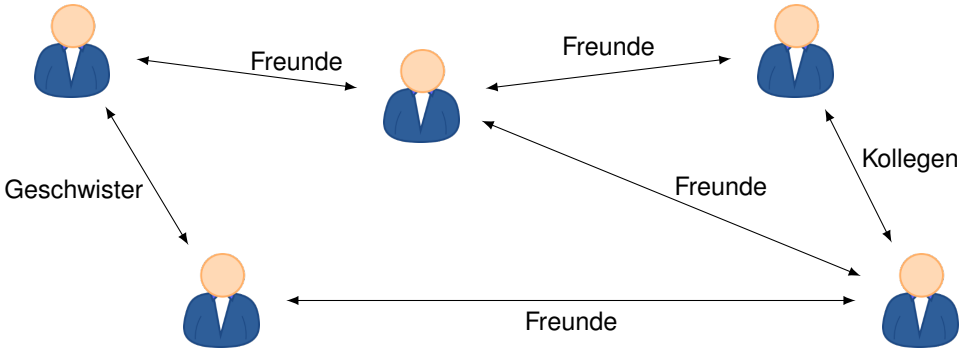
# Soziales Netzwerk



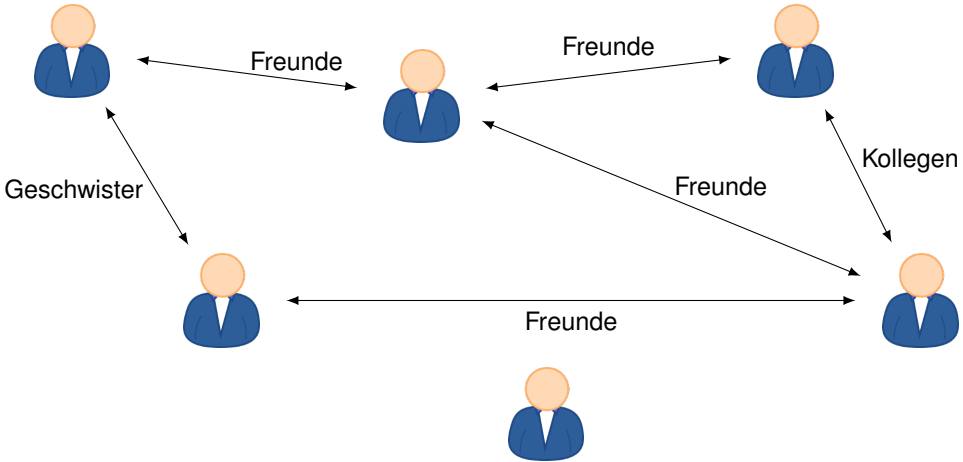
# Soziales Netzwerk



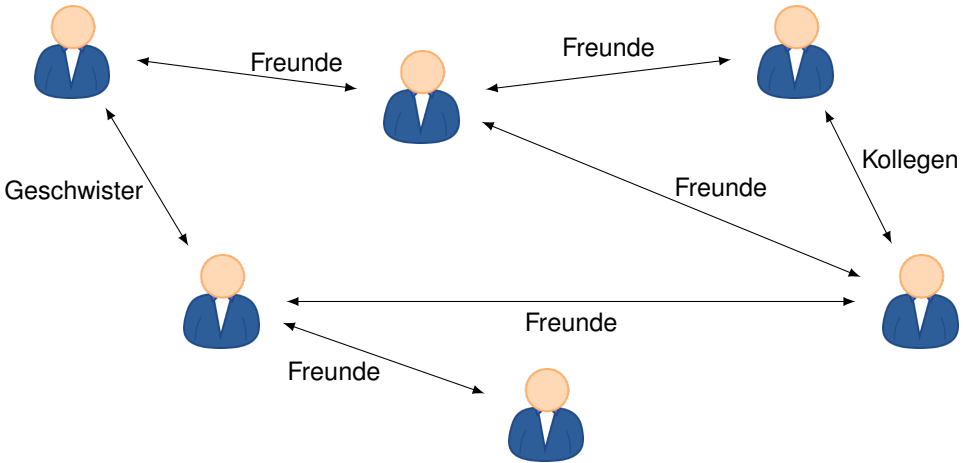
# Soziales Netzwerk



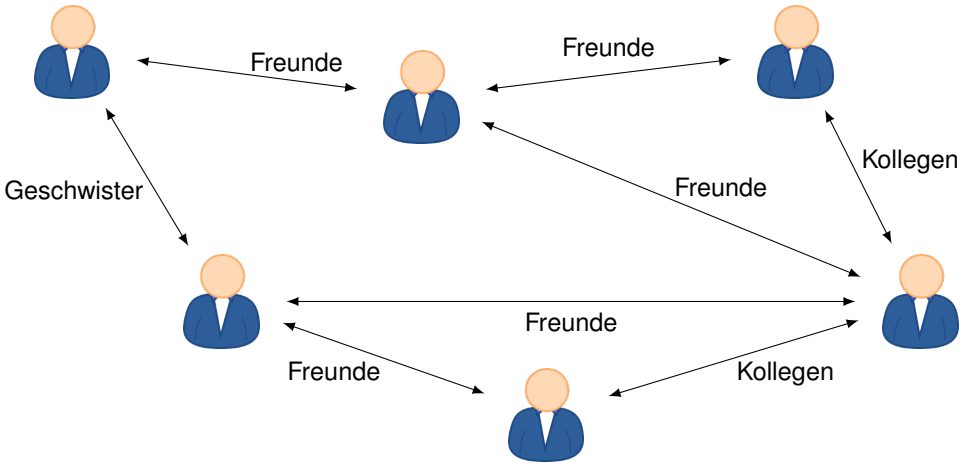
# Soziales Netzwerk



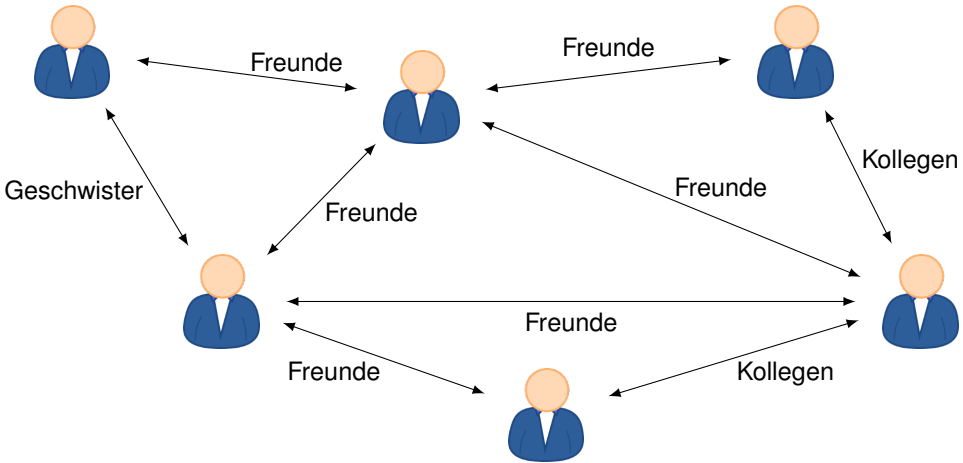
# Soziales Netzwerk



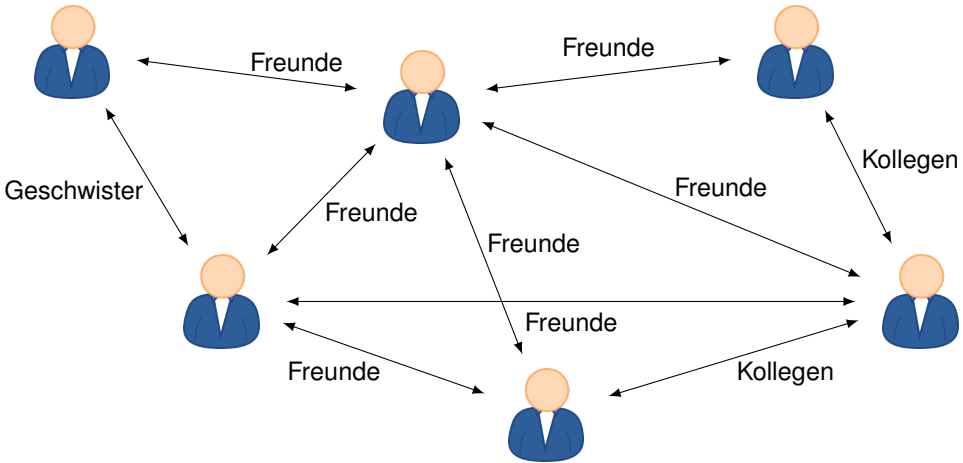
# Soziales Netzwerk



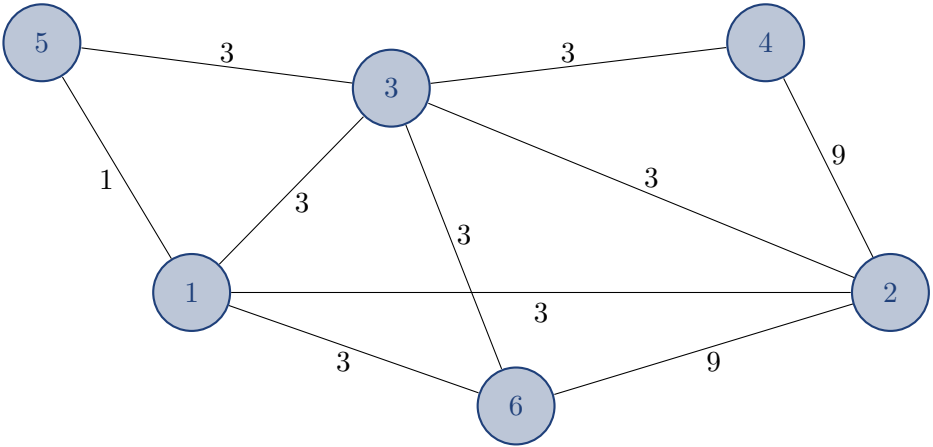
# Soziales Netzwerk



# Soziales Netzwerk



# Abstrakte Modellierung



# Abstrakte Modellierung

Ein **Graph** $G = (V, E, w)$ besteht aus

1. einer Menge $V$ von Knoten
2. einer Menge $E$ von Kanten zwischen einigen der Knoten
3. (einer Gewichtsfunktion)

# Abstrakte Modellierung

Ein **Graph** $G = (V, E, w)$ besteht aus

1. einer Menge $V$ von Knoten
2. einer Menge $E$ von Kanten zwischen einigen der Knoten
3. (einer Gewichtsfunktion)

- Knoten werden bezeichnet mit $v_0, v_1, v_2, \ldots$

# Abstrakte Modellierung

Ein **Graph** $G = (V, E, w)$ besteht aus

1. einer Menge $V$ von Knoten
2. einer Menge $E$ von Kanten zwischen einigen der Knoten
3. (einer Gewichtsfunktion)

- Knoten werden bezeichnet mit $v_0, v_1, v_2, \ldots$
- Graphen sind entweder **gewichtet** oder **ungewichtet**

# Abstrakte Modellierung

Ein **Graph** $G = (V, E, w)$ besteht aus

1. einer Menge $V$ von Knoten
2. einer Menge $E$ von Kanten zwischen einigen der Knoten
3. (einer Gewichtsfunktion)

- Knoten werden bezeichnet mit $v_0, v_1, v_2, \ldots$
- Graphen sind entweder **gewichtet** oder **ungewichtet**
- Graphen sind entweder **gerichtet** oder **ungerichtet**

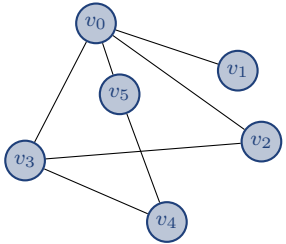
# Abstrakte Modellierung

Ein **Graph** $G = (V, E, w)$ besteht aus

1. einer Menge $V$ von Knoten
2. einer Menge $E$ von Kanten zwischen einigen der Knoten
3. (einer Gewichtsfunktion)

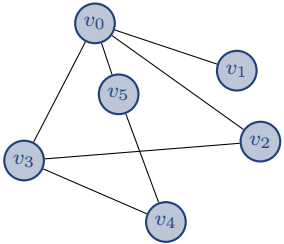
- Knoten werden bezeichnet mit $v_0, v_1, v_2, \ldots$
- Graphen sind entweder **gewichtet** oder **ungewichtet**
- Graphen sind entweder **gerichtet** oder **ungerichtet**
- Graphen sind entweder **verbunden** oder **nicht verbunden**

# Abstrakte Modellierung

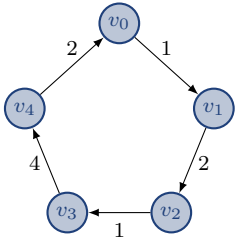



Ungerichteter ungewichteter Graph

# Abstrakte Modellierung

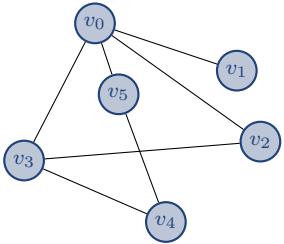



Ungerichteter ungewichteter Graph

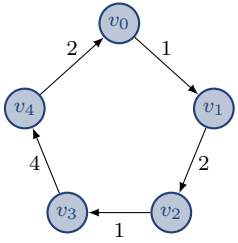



Gerichteter gewichteter Graph

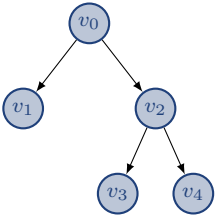
# Abstrakte Modellierung




Ungerichteter ungewichteter Graph

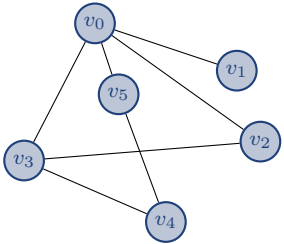



Gerichteter gewichteter Graph

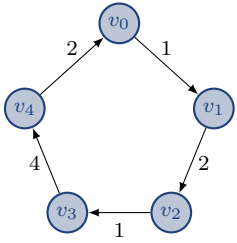



Gerichteter ungewichteter Graph

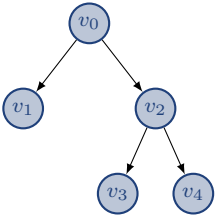
# Abstrakte Modellierung




Ungerichteter ungewichteter Graph



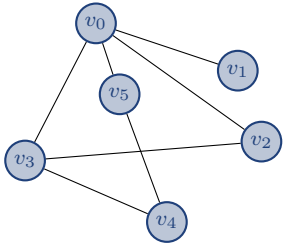

Gerichteter gewichteter Graph



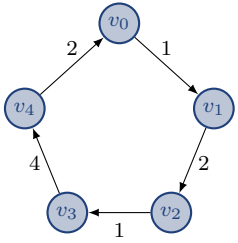

Gerichteter ungewichteter Graph

Welcher Typ von Graph verwendet wird, hängt vom Modell ab

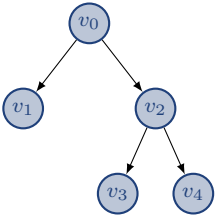
# Abstrakte Modellierung




Ungerichteter ungewichteter Graph




Gerichteter gewichteter Graph




Gerichteter ungewichteter Graph

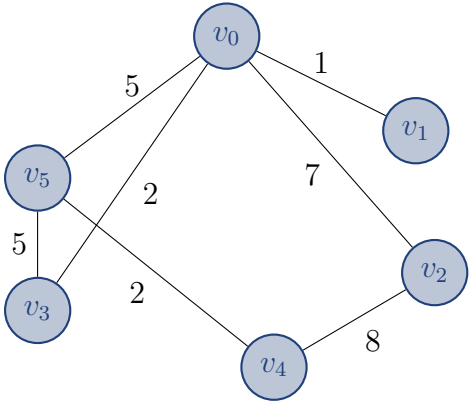
Welcher Typ von Graph verwendet wird, hängt vom Modell ab

Wir betrachten meist ungerichtete, ungewichtete und verbundene Graphen

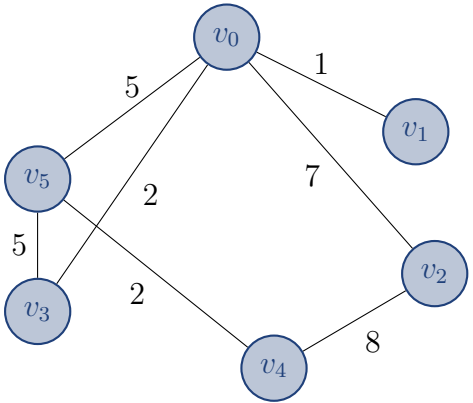
# Graphen

## Auf dem Computer

# Adjazenzmatrizen – Ungerichtete gewichtete Graphen

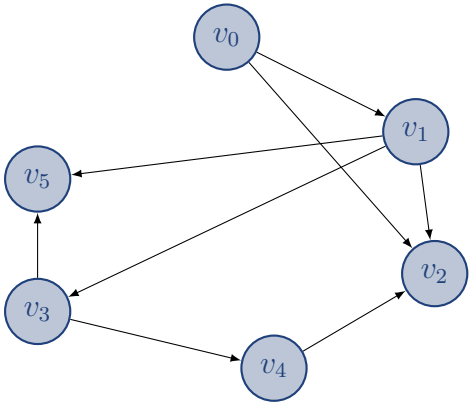


# Adjazenzmatrizen – Ungerichtete gewichtete Graphen

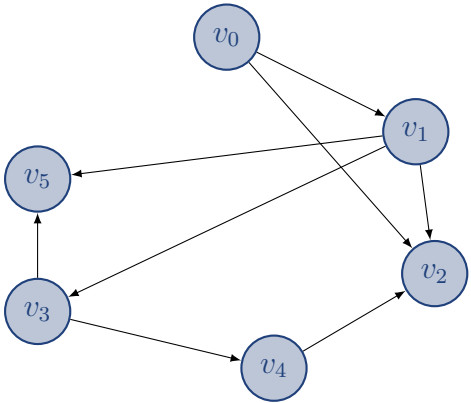



$$\begin{pmatrix} 0 & 1 & 7 & 2 & 0 & 5 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 8 & 0 \\ 2 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 8 & 0 & 0 & 2 \\ 5 & 0 & 0 & 5 & 2 & 0 \end{pmatrix}$$

# Adjazenzmatrizen – Gerichtete ungewichtete Graphen



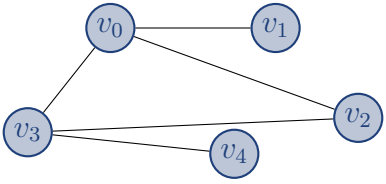
# Adjazenzmatrizen – Gerichtete ungewichtete Graphen




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

# Adjazenzmatrizen – Gerichtete / ungerichtete Graphen

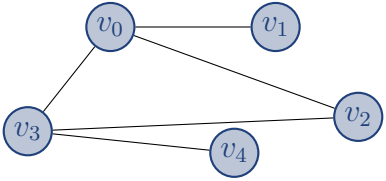
Matrizen ungerichteter Graphen sind symmetrisch




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

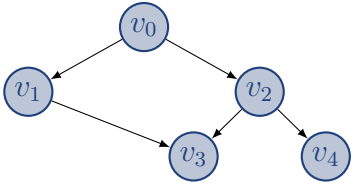
# Adjazenzmatrizen – Gerichtete / ungerichtete Graphen

Matrizen ungerichteter Graphen sind symmetrisch



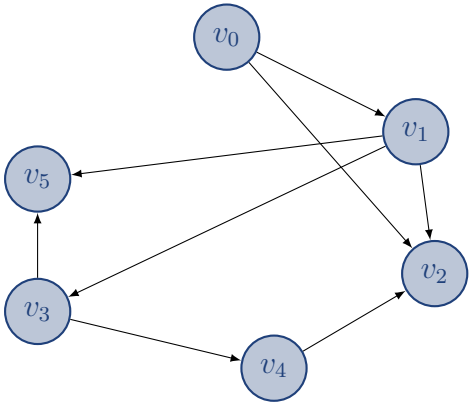

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Matrizen gerichteter Graphen sind nicht (immer) symmetrisch

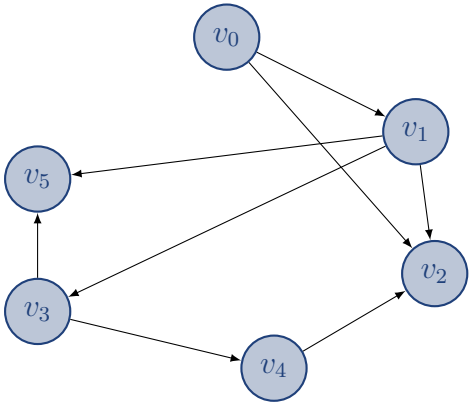



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

# Adjazenzlisten – Gerichtete ungewichtete Graphen



# Adjazenzlisten – Gerichtete ungewichtete Graphen




$$((1,2),\ (2,3,5),\ (),\ (4,5),\ (2),\ ())$$

# Adjazenzmatrizen und -Listen in Python

## Verwende 2-dimensionale Listen

# Adjazenzmatrizen und -Listen in Python

## Verwende 2-dimensionale Listen

Matrix: Gewichtet

```
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 ] ]
```

# Adjazenzmatrizen und -Listen in Python

## Verwende 2-dimensionale Listen

Matrix: Gewichtet

```
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: Ungewichtet

```
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 ] ]
```

# Adjazenzmatrizen und -Listen in Python

## Verwende 2-dimensionale Listen

Matrix: Gewichtet

```
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: Ungewichtet

```
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 ] ]
```

Liste: Ungewichtet

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

# Graph-Algorithmen

## Breitensuche und Tiefensuche

# Breitensuche (BFS) und Tiefensuche (DFS)

**Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen**

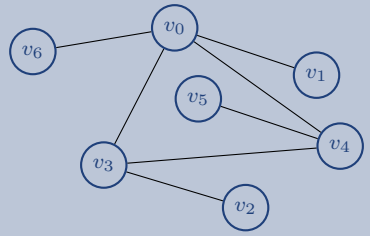
- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

# Breitensuche (BFS) und Tiefensuche (DFS)

## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index

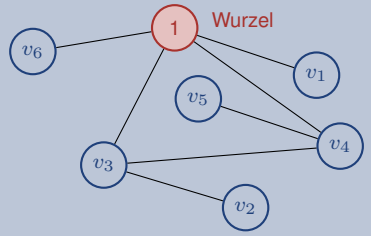


# Breitensuche (BFS) und Tiefensuche (DFS)

## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index

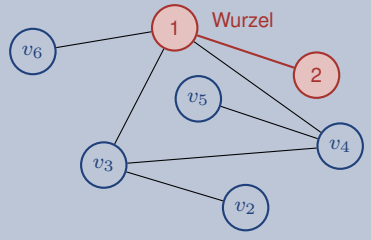


# Breitensuche (BFS) und Tiefensuche (DFS)

## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index

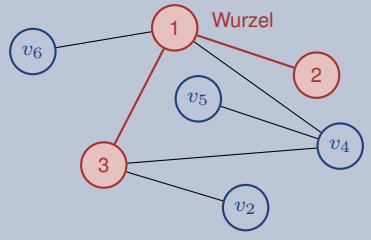


# Breitensuche (BFS) und Tiefensuche (DFS)

## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index

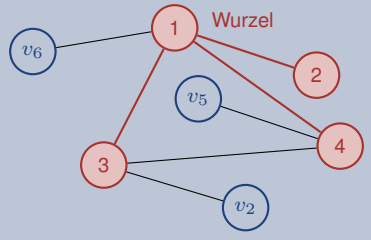


# Breitensuche (BFS) und Tiefensuche (DFS)

## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index

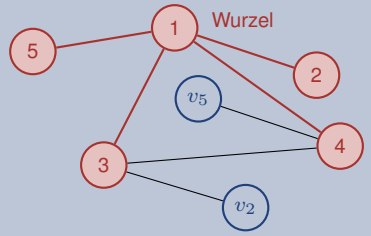


# Breitensuche (BFS) und Tiefensuche (DFS)

## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index

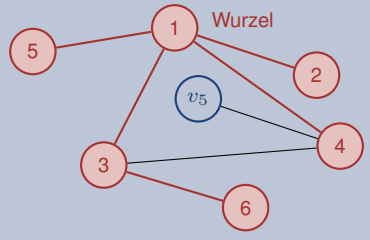


# Breitensuche (BFS) und Tiefensuche (DFS)

## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index

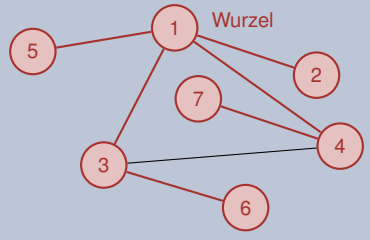


# Breitensuche (BFS) und Tiefensuche (DFS)

## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index

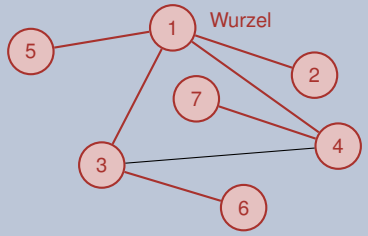


# Breitensuche (BFS) und Tiefensuche (DFS)

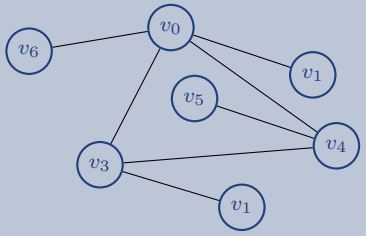
## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index




DFS: Steige so tief wie möglich in den Graphen ab, dann in die Breite; bei Auswahl, nimm kleineren Index

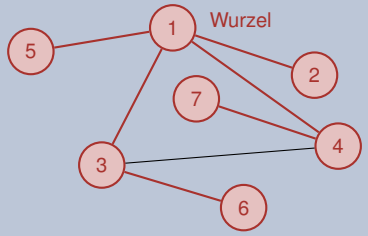


# Breitensuche (BFS) und Tiefensuche (DFS)

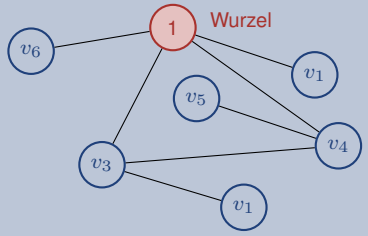
## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index




DFS: Steige so tief wie möglich in den Graphen ab, dann in die Breite; bei Auswahl, nimm kleineren Index

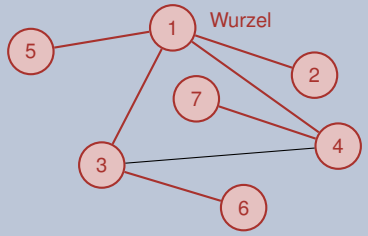


# Breitensuche (BFS) und Tiefensuche (DFS)

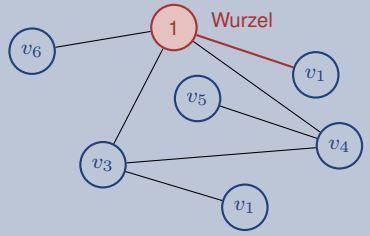
## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index




DFS: Steige so tief wie möglich in den Graphen ab, dann in die Breite; bei Auswahl, nimm kleineren Index

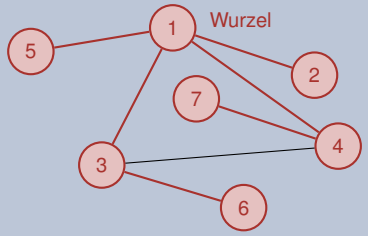


# Breitensuche (BFS) und Tiefensuche (DFS)

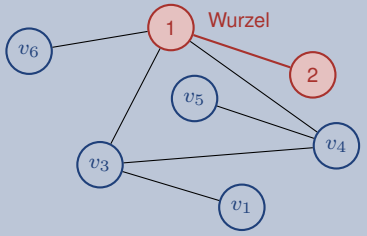
## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index




DFS: Steige so tief wie möglich in den Graphen ab, dann in die Breite; bei Auswahl, nimm kleineren Index

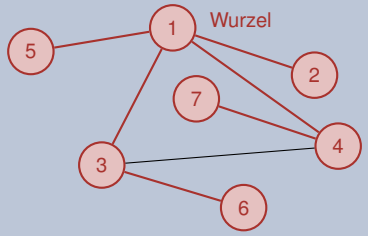


# Breitensuche (BFS) und Tiefensuche (DFS)

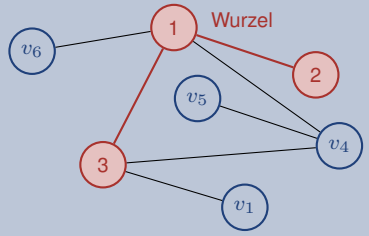
## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index




DFS: Steige so tief wie möglich in den Graphen ab, dann in die Breite; bei Auswahl, nimm kleineren Index

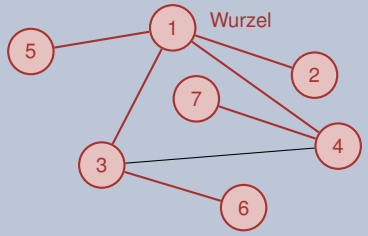


# Breitensuche (BFS) und Tiefensuche (DFS)

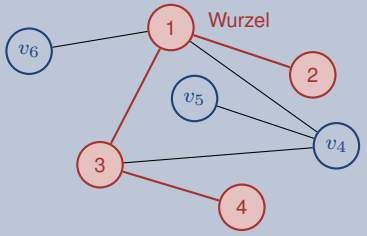
## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index




DFS: Steige so tief wie möglich in den Graphen ab, dann in die Breite; bei Auswahl, nimm kleineren Index

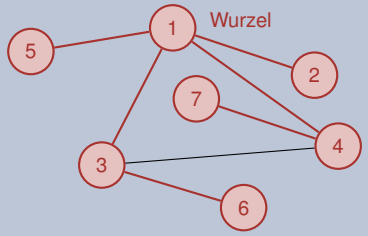


# Breitensuche (BFS) und Tiefensuche (DFS)

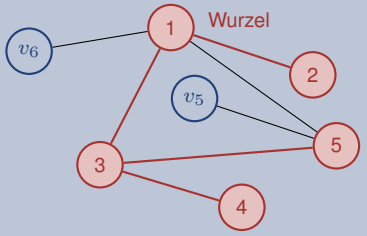
## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index




DFS: Steige so tief wie möglich in den Graphen ab, dann in die Breite; bei Auswahl, nimm kleineren Index

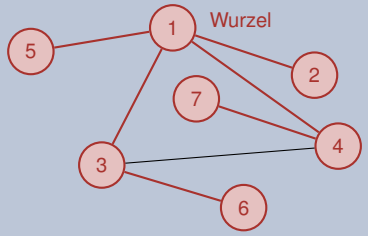


# Breitensuche (BFS) und Tiefensuche (DFS)

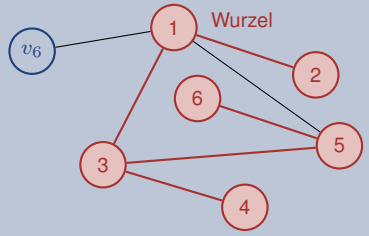
## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index




DFS: Steige so tief wie möglich in den Graphen ab, dann in die Breite; bei Auswahl, nimm kleineren Index

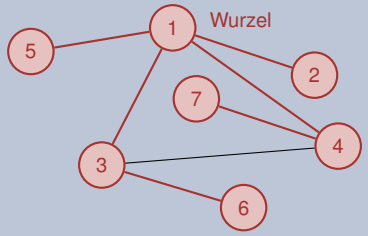


# Breitensuche (BFS) und Tiefensuche (DFS)

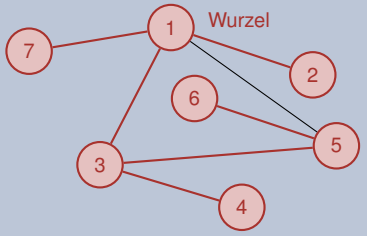
## Viele Aufgaben erfordern systematisches Durchsuchen eines Graphen

- Starte an einem beliebigen Knoten
- Laufe an Kanten entlang durch den Graphen
- Speichere Knoten in der entsprechenden Reihenfolge

BFS: Gehe erst in die Breite und dann in die Tiefe, wie beim Heap; bei Auswahl, nimm kleineren Index



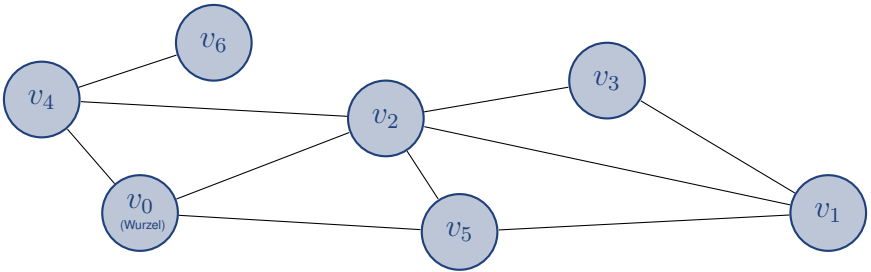

DFS: Steige so tief wie möglich in den Graphen ab, dann in die Breite; bei Auswahl, nimm kleineren Index



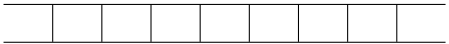
# Breitensuche

## Iterativ mit einer Queue

# Breitensuche mit einer Queue

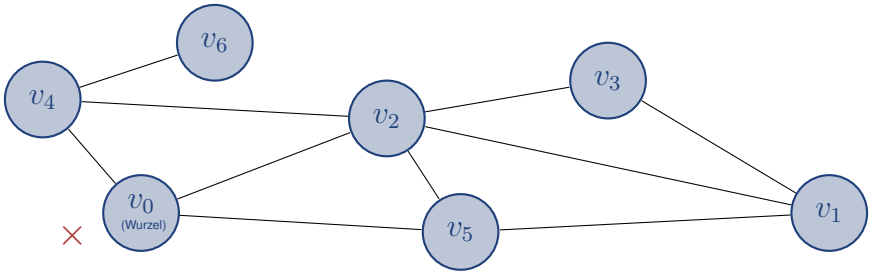



Queue:

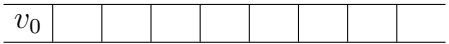


Ausgabe:

# Breitensuche mit einer Queue

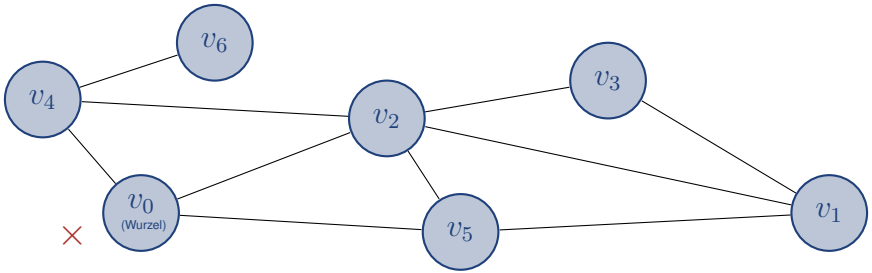



Queue:

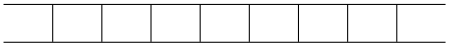



Ausgabe:

# Breitensuche mit einer Queue

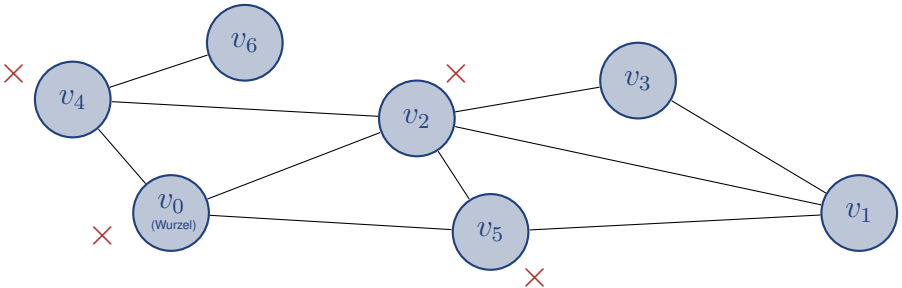



Queue:



Ausgabe: $v_0$

# Breitensuche mit einer Queue

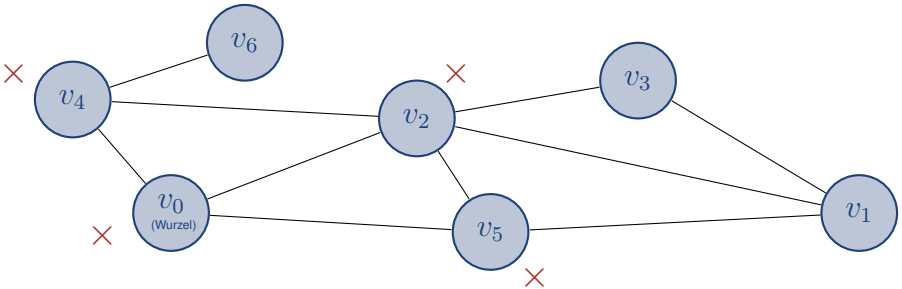



Queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_2$ | $v_4$ | $v_5$ | | | | | | |

Ausgabe: $v_0$

# Breitensuche mit einer Queue

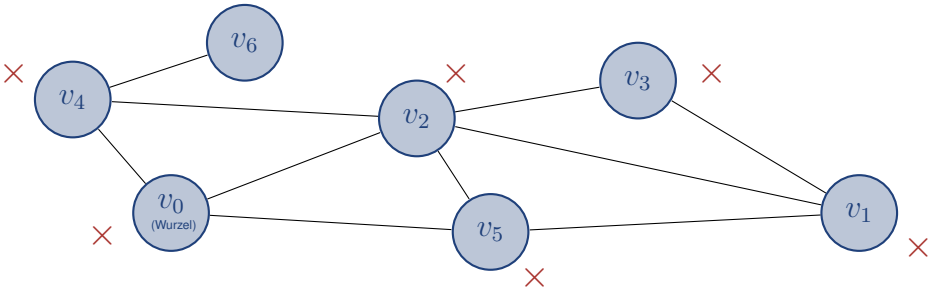



Queue:

| $v_4$ | $v_5$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|

Ausgabe: $v_0$ $v_2$

# Breitensuche mit einer Queue

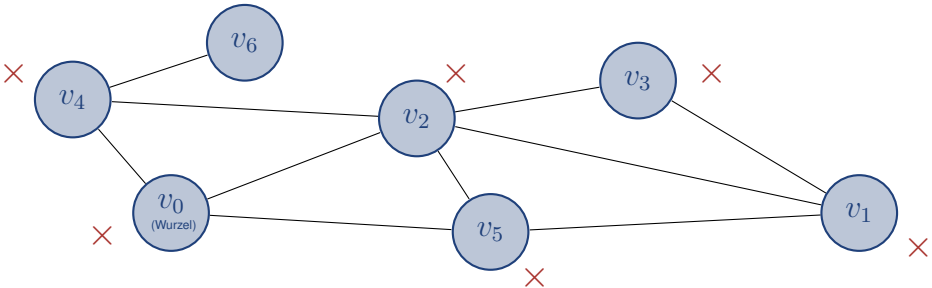



Queue:

| $v_4$ | $v_5$ | $v_1$ | $v_3$ | | | | | |
|---|---|---|---|---|---|---|---|---|

Ausgabe: $v_0$ $v_2$

# Breitensuche mit einer Queue

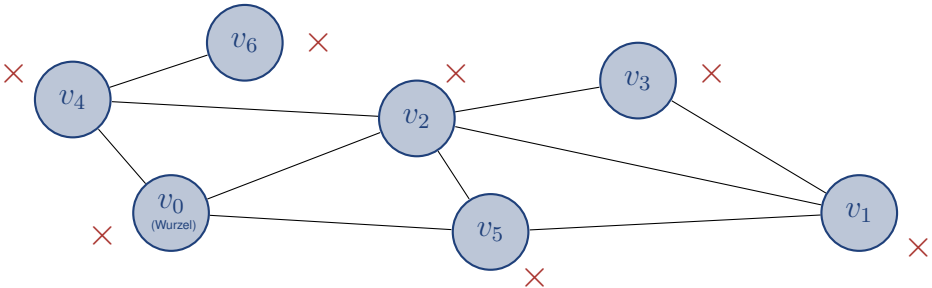



Queue:

| $v_5$ | $v_1$ | $v_3$ | | | | | | |
|---|---|---|---|---|---|---|---|---|

Ausgabe: $v_0 \quad v_2 \quad v_4$

# Breitensuche mit einer Queue

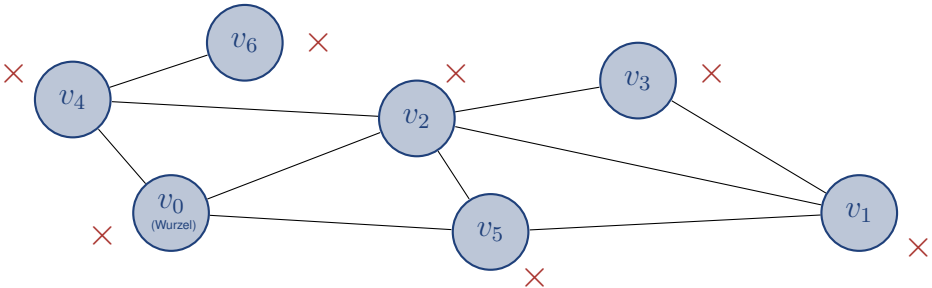



Queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_5$ | $v_1$ | $v_3$ | $v_6$ | | | | | |

Ausgabe: $v_0$ $v_2$ $v_4$

# Breitensuche mit einer Queue

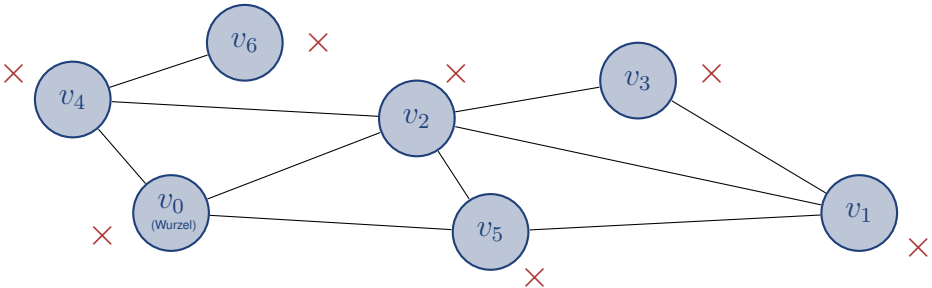



Queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_1$ | $v_3$ | $v_6$ | | | | | | |

Ausgabe: $v_0$ $v_2$ $v_4$ $v_5$

# Breitensuche mit einer Queue

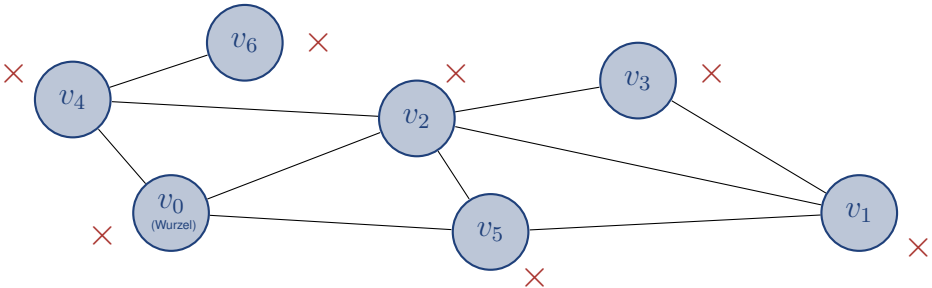



Queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_3$ | $v_6$ | | | | | | | |

Ausgabe: $v_0$ $v_2$ $v_4$ $v_5$ $v_1$

# Breitensuche mit einer Queue

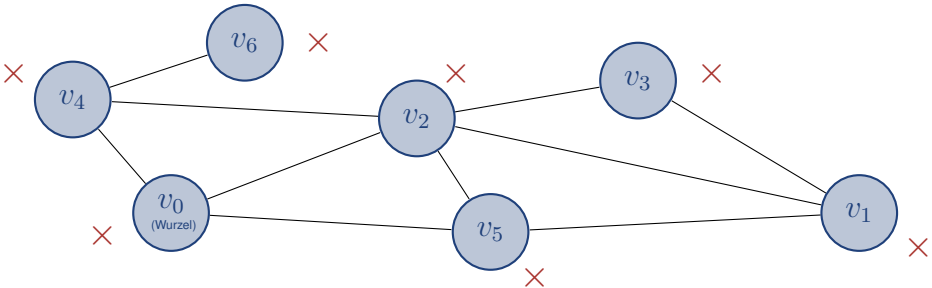



Queue:

| $v_6$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|

Ausgabe: $v_0$ $v_2$ $v_4$ $v_5$ $v_1$ $v_3$

# Breitensuche mit einer Queue




Queue:

Ausgabe: $v_0$ $v_2$ $v_4$ $v_5$ $v_1$ $v_3$ $v_6$

# Breitensuche mit Queue und Adjazenzmatrix

```
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)) ]
```

# Breitensuche mit Queue und Adjazenzmatrix

```
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)) ]
```

- Betrachte ersten Knoten in Queue und gib ihn aus

# Breitensuche mit Queue und Adjazenzmatrix

```
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)) ]
```

- Betrachte ersten Knoten in Queue und gib ihn aus
- Füge noch nicht besuchte Nachbarn in Queue ein

# Breitensuche mit Queue und Adjazenzmatrix

```
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)) ]
```

- Betrachte ersten Knoten in Queue und gib ihn aus
- Füge noch nicht besuchte Nachbarn in Queue ein
- `visited` speichert, welche schon besucht wurden

# Breitensuche mit Queue und Adjazenzmatrix

```
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)) ]
```

- Betrachte ersten Knoten in Queue und gib ihn aus
- Füge noch nicht besuchte Nachbarn in Queue ein
- `visited` speichert, welche schon besucht wurden
- Wiederhole, solange Queue nicht leer ist

# Aufgabe – Breitensuche mit Queue und Adjazenzmatrix

**Implementieren Sie die Breitensuche**

- als Python-Funktion
- mit einer 2-dimensionalen Liste als Parameter
- unter Verwendung einer Queue
- und einer Adjazenzmatrix

```
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] ]
```

# Breitensuche mit Queue und Adjazenzmatrix

```
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] ])
```

# Aufgabe – Breitensuche mit Queue und Adjazenzliste

**Implementieren Sie die Breitensuche**

- als Python-Funktion
- mit einer 2-dimensionalen Liste als Parameter
- unter Verwendung einer Queue
- und einer Adjazenzliste

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

# Breitensuche mit Queue und Adjazenzliste

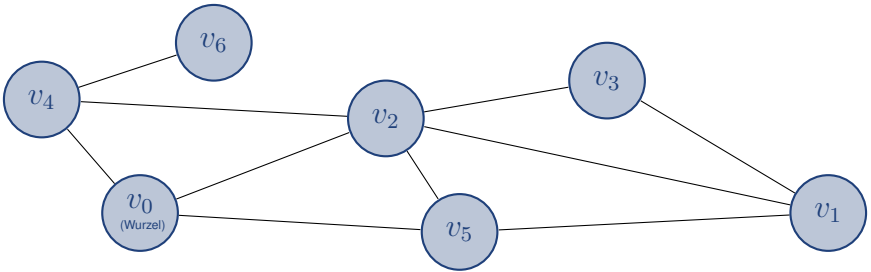
```
def BFS(G):
    queue = []
    visited = [ 0 for i in range(len(G)) ]
    queue.append(0)
    visited[0] = 1
    while len(queue) > 0:
        current = queue.pop(0)
        print(current, end=" ")
        for j in G[current]:
            if visited[j] == 0:
                visited[j] = 1
      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

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

# Tiefensuche

## Iterativ mit einem Stack

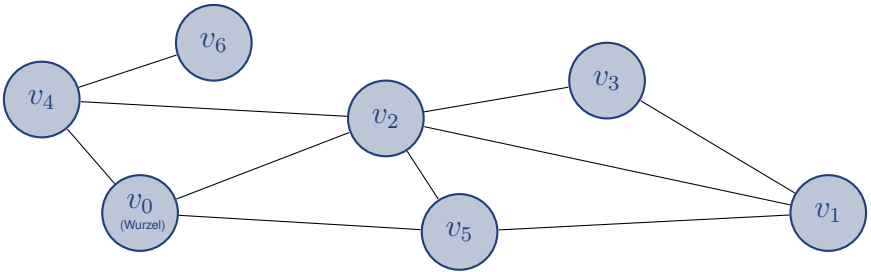
# Tiefensuche mit einem Stack



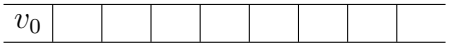

Stack:

Ausgabe:

# Tiefensuche mit einem Stack

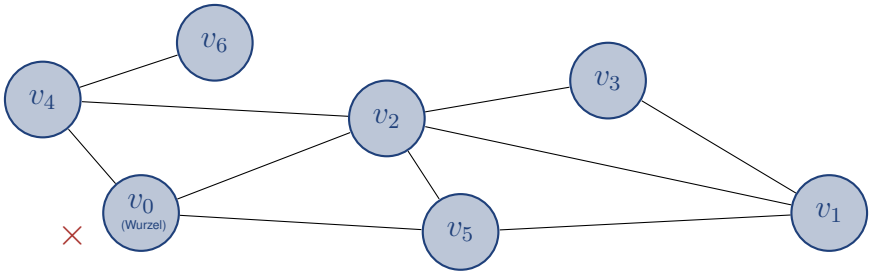



Stack:

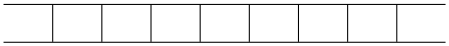



Ausgabe:

# Tiefensuche mit einem Stack

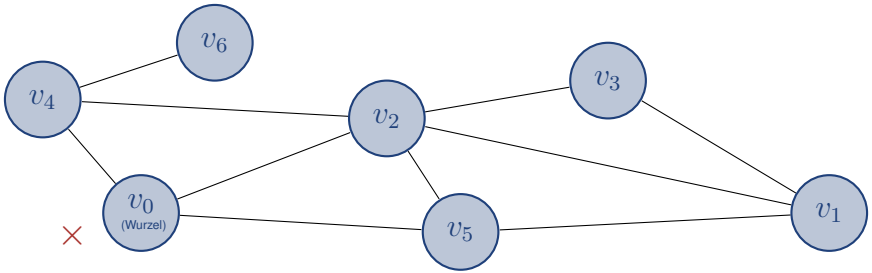



Stack:



Ausgabe: $v_0$

# Tiefensuche mit einem Stack

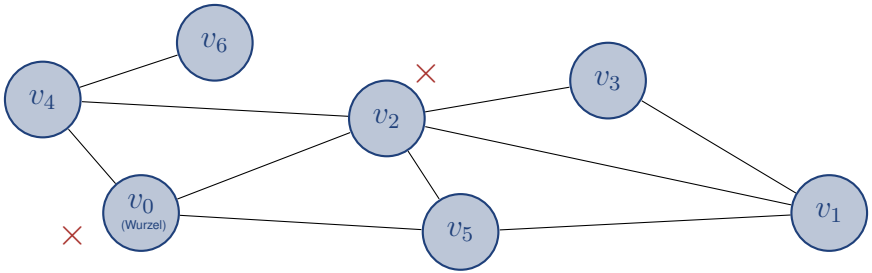



Stack:

| $v_5$ | $v_4$ | $v_2$ | | | | | | |
|---|---|---|---|---|---|---|---|---|

Ausgabe: $v_0$

# Tiefensuche mit einem Stack

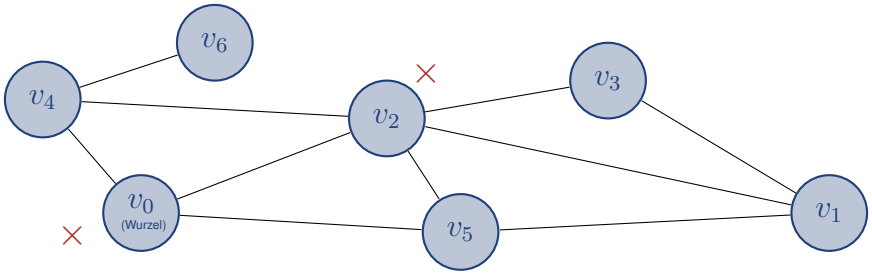



Stack:

| $v_5$ | $v_4$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|

Ausgabe: $v_0$ $v_2$

# Tiefensuche mit einem Stack

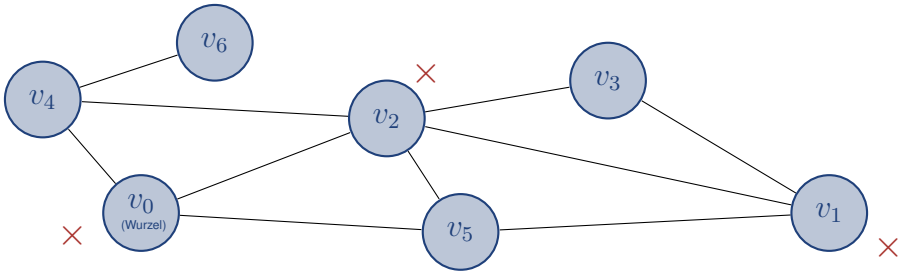



Stack:

| $v_5$ | $v_4$ | $v_5$ | $v_4$ | $v_3$ | $v_1$ | | | |
|---|---|---|---|---|---|---|---|---|

Ausgabe: $v_0 \quad v_2$

# Tiefensuche mit einem Stack

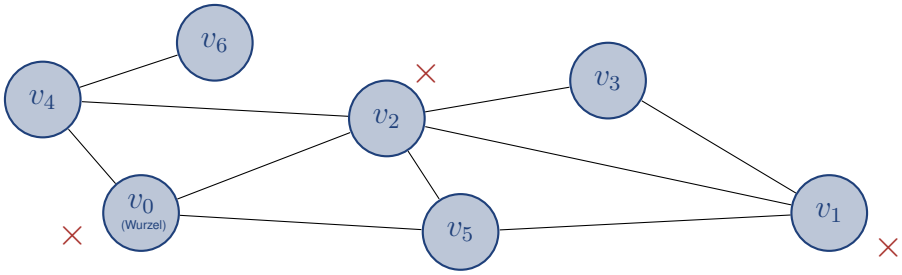



Stack:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_5$ | $v_4$ | $v_5$ | $v_4$ | $v_3$ | | | | |

Ausgabe: $v_0$ $v_2$ $v_1$

# Tiefensuche mit einem Stack

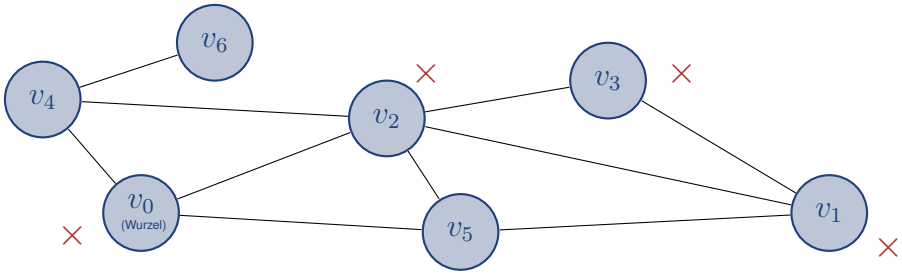



Stack:

| $v_5$ | $v_4$ | $v_5$ | $v_4$ | $v_3$ | $v_5$ | $v_3$ | | |
|---|---|---|---|---|---|---|---|---|

Ausgabe: $v_0$ $v_2$ $v_1$

# Tiefensuche mit einem Stack

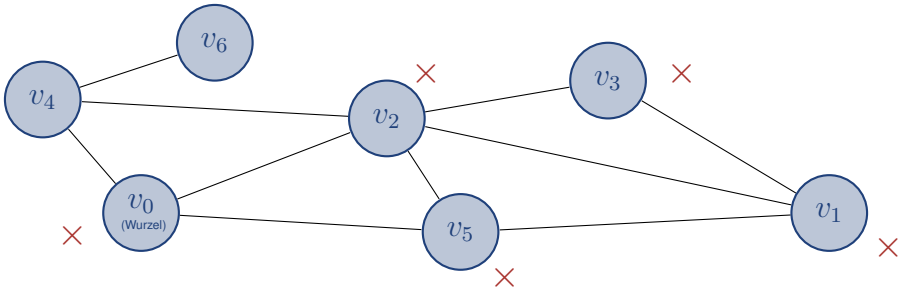



Stack:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_5$ | $v_4$ | $v_5$ | $v_4$ | $v_3$ | $v_5$ | | | |

Ausgabe: $v_0$ $v_2$ $v_1$ $v_3$

# Tiefensuche mit einem Stack

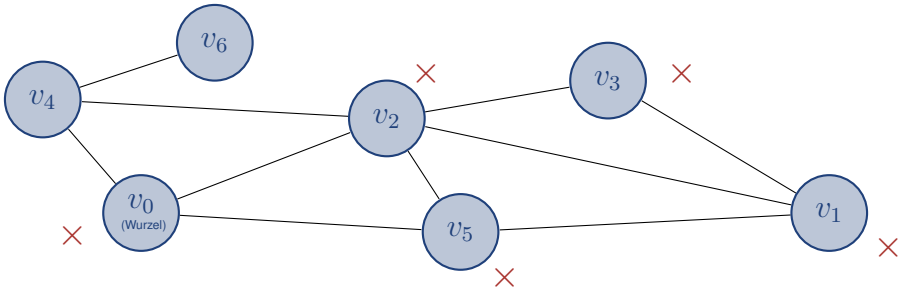



Stack:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_5$ | $v_4$ | $v_5$ | $v_4$ | $v_3$ | | | | |

Ausgabe: $v_0$ $v_2$ $v_1$ $v_3$ $v_5$

# Tiefensuche mit einem Stack

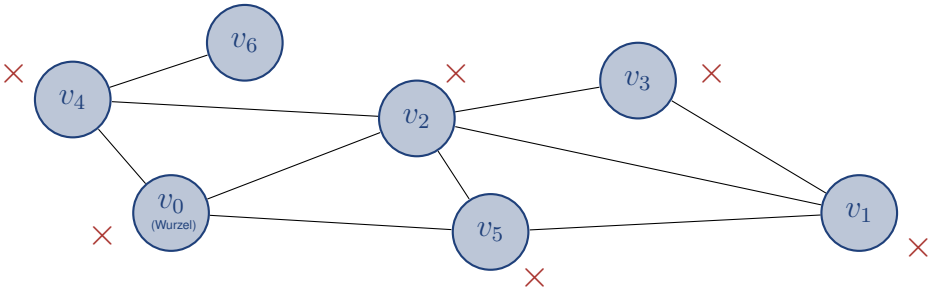



Stack:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_5$ | $v_4$ | $v_5$ | $v_4$ | | | | | |

Ausgabe: $v_0$ $v_2$ $v_1$ $v_3$ $v_5$

# Tiefensuche mit einem Stack

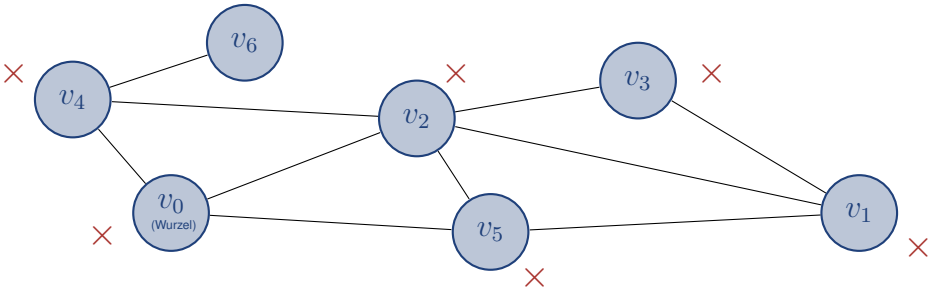



Stack:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_5$ | $v_4$ | $v_5$ | | | | | | |

Ausgabe: $v_0$ $v_2$ $v_1$ $v_3$ $v_5$ $v_4$

# Tiefensuche mit einem Stack

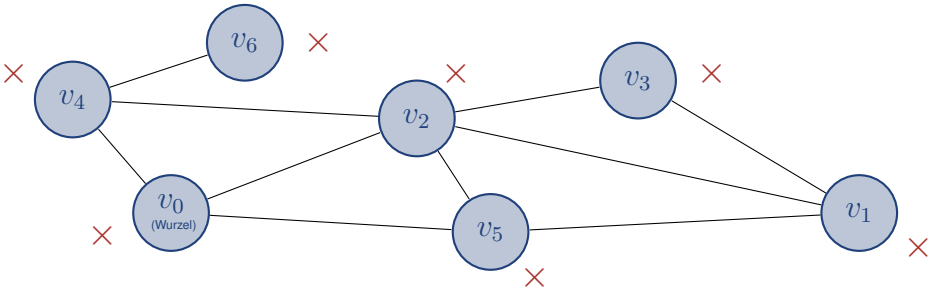



Stack:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_5$ | $v_4$ | $v_5$ | $v_6$ | | | | | |

Ausgabe: $v_0$ $v_2$ $v_1$ $v_3$ $v_5$ $v_4$

# Tiefensuche mit einem Stack

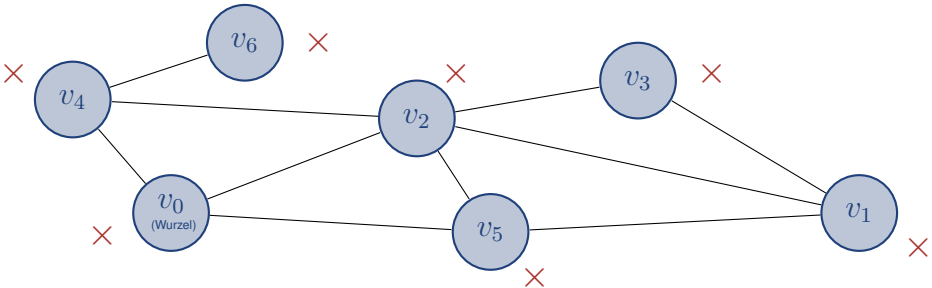



Stack:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_5$ | $v_4$ | $v_5$ | | | | | | |

Ausgabe: $v_0$ $v_2$ $v_1$ $v_3$ $v_5$ $v_4$ $v_6$

# Tiefensuche mit einem Stack

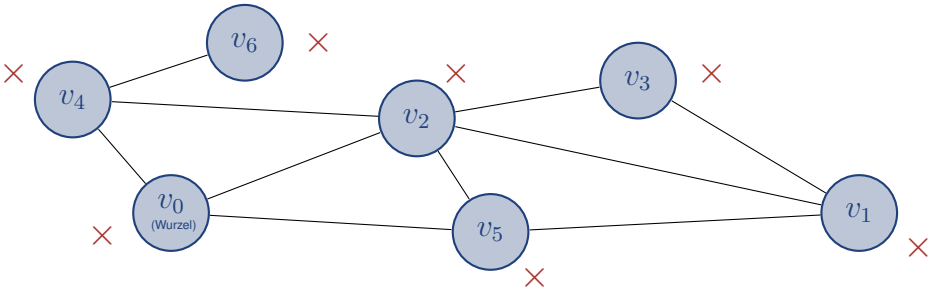



Stack:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $v_5$ | $v_4$ | | | | | | | |

Ausgabe: $v_0$ $v_2$ $v_1$ $v_3$ $v_5$ $v_4$ $v_6$

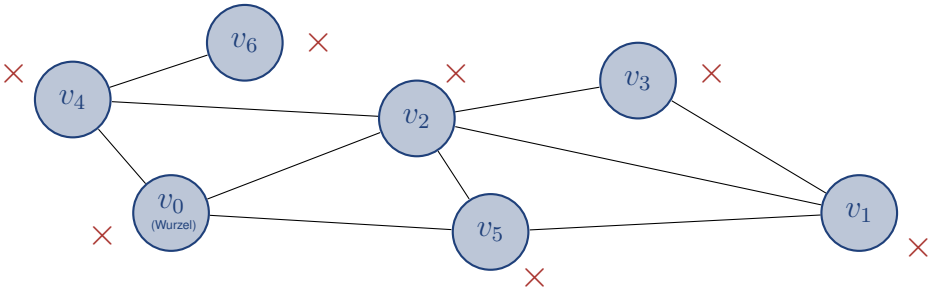
# Tiefensuche mit einem Stack




Stack:


Ausgabe: $v_0$ $v_2$ $v_1$ $v_3$ $v_5$ $v_4$ $v_6$

# Tiefensuche mit einem Stack




Stack:

Ausgabe: $v_0$ $v_2$ $v_1$ $v_3$ $v_5$ $v_4$ $v_6$

# Tiefensuche mit Stack und Adjazenzmatrix

```
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)) ]
```

# Tiefensuche mit Stack und Adjazenzmatrix

```
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)) ]
```

- Betrachte ersten Knoten im Stack und gib ihn aus

# Tiefensuche mit Stack und Adjazenzmatrix

```
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)) ]
```

- Betrachte ersten Knoten im Stack und gib ihn aus
- Füge noch nicht besuchte Nachbarn in Stack ein

# Tiefensuche mit Stack und Adjazenzmatrix

```
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)) ]
```

- Betrachte ersten Knoten im Stack und gib ihn aus
- Füge noch nicht besuchte Nachbarn in Stack ein
- `visited` speichert, welche schon besucht wurden

# Tiefensuche mit Stack und Adjazenzmatrix

```
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)) ]
```

- Betrachte ersten Knoten im Stack und gib ihn aus
- Füge noch nicht besuchte Nachbarn in Stack ein
- `visited` speichert, welche schon besucht wurden
- Wiederhole, solange Stack nicht leer ist

# Aufgabe – Tiefensuche mit Stack und Adjazenzmatrix

**Implementieren Sie die Tiefensuche**

- als Python-Funktion
- mit einer 2-dimensionalen Liste als Parameter
- unter Verwendung eines Stacks
- und einer Adjazenzmatrix

```
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] ]
```

# Tiefensuche mit Stack und Adjazenzmatrix

```
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] ] )
```

# Aufgabe – Tiefensuche mit Stack und Adjazenzliste

**Implementieren Sie die Tiefensuche**

- als Python-Funktion
- mit einer 2-dimensionalen Liste als Parameter
- unter Verwendung eines Stacks
- und einer Adjazenzmatrix

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

# Tiefensuche mit Stack und Adjazenzliste

```
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] ])
```

# Tiefensuche

## Rekursiv

# Rekursive Tiefensuche

- Globale Liste `visited`

# Rekursive Tiefensuche

- Globale Liste `visited`
- Funktion `DFS`, die rekursiv aufgerufen wird

# Rekursive Tiefensuche

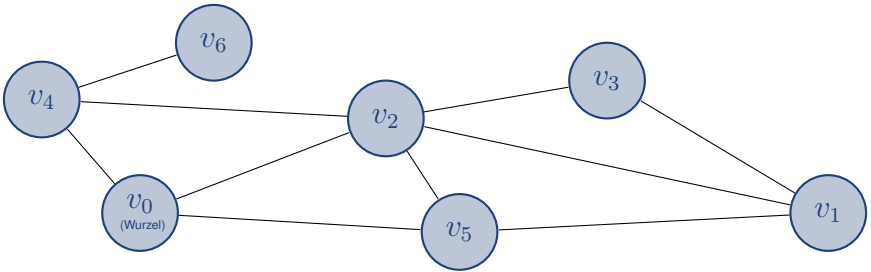
- Globale Liste `visited`
- Funktion `DFS`, die rekursiv aufgerufen wird
- Zwei Parameter
  1. Graph `G`
  2. Start-Knoten `current`

# Rekursive Tiefensuche

- Globale Liste `visited`
- Funktion `DFS`, die rekursiv aufgerufen wird
- Zwei Parameter
  1. Graph `G`
  2. Start-Knoten `current`

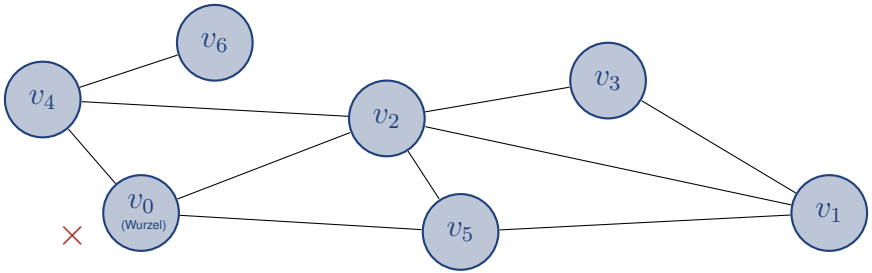
```
visited = [ 0 for i in range(len(G)) ]
def DFS(G, current):
    visited[current] = 1
    print(current, end=" ")
    for i in range(len(G)):
        if G[current][i] == 1 and visited[i] == 0:
            DFS(G, i)
```

# Rekursive Tiefensuche




Ausgabe:

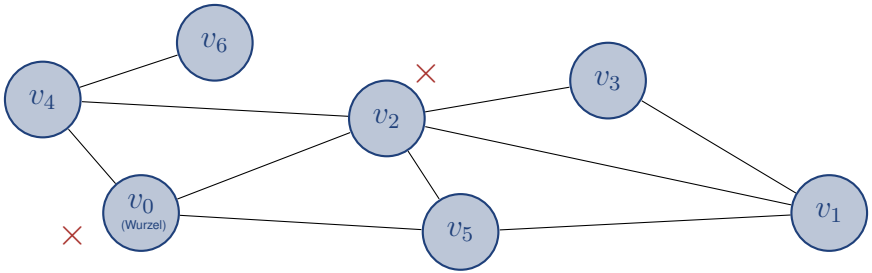
# Rekursive Tiefensuche




```
DFS(G,0)
```

Ausgabe: $v_0$

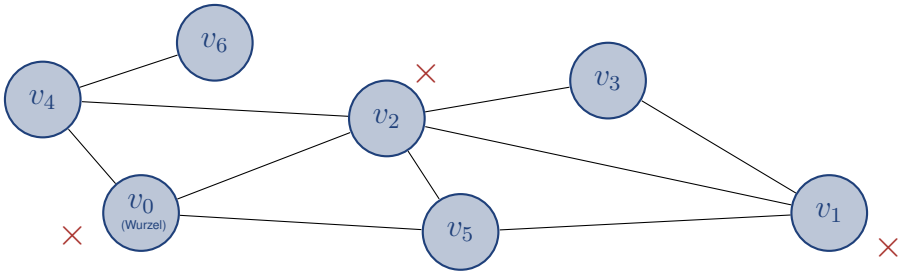
# Rekursive Tiefensuche




`DFS(G,0)` → `DFS(G,2)`

Ausgabe: $v_0 \quad v_2$

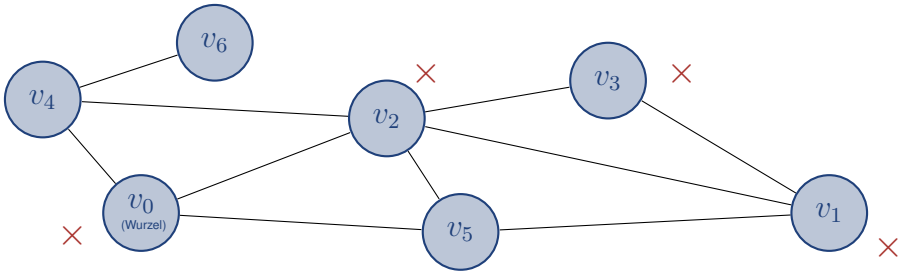
# Rekursive Tiefensuche




`DFS(G,0)` → `DFS(G,2)` → `DFS(G,1)`

Ausgabe: $v_0 \quad v_2 \quad v_1$

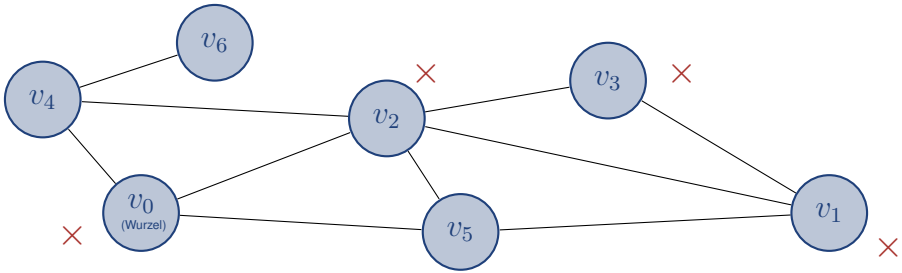
# Rekursive Tiefensuche

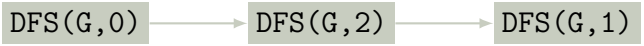



`DFS(G,0)` → `DFS(G,2)` → `DFS(G,1)` → `DFS(G,3)`

Ausgabe: $v_0 \quad v_2 \quad v_1 \quad v_3$

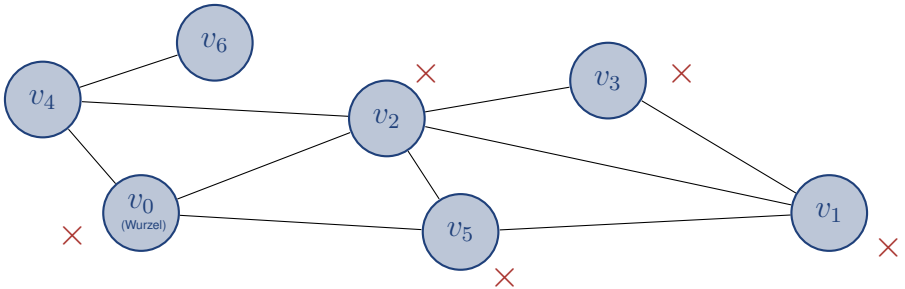
# Rekursive Tiefensuche




Ausgabe: $v_0 \quad v_2 \quad v_1 \quad v_3$

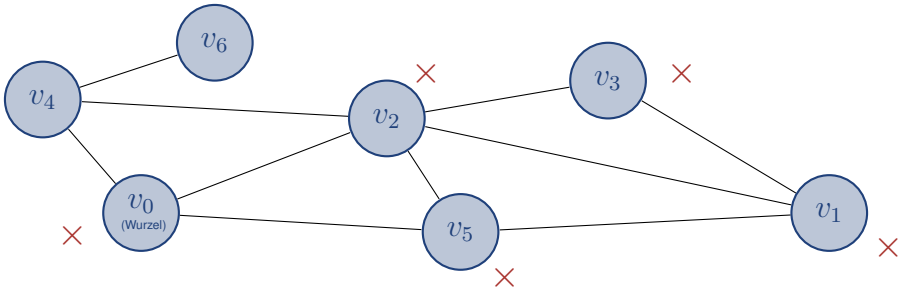
# Rekursive Tiefensuche





Ausgabe: $v_0 \quad v_2 \quad v_1 \quad v_3 \quad v_5$

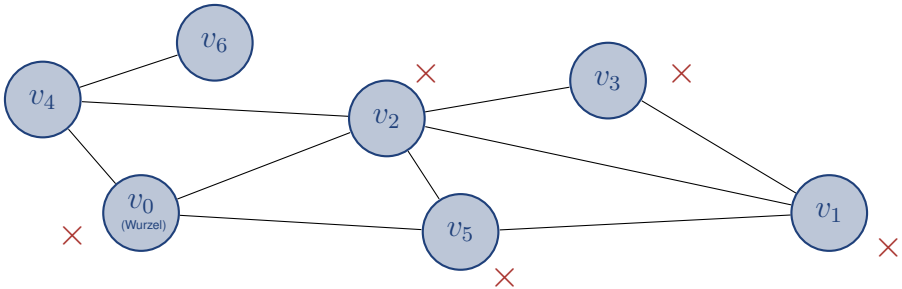
# Rekursive Tiefensuche




`DFS(G,0)` → `DFS(G,2)` → `DFS(G,1)`

Ausgabe: $v_0 \quad v_2 \quad v_1 \quad v_3 \quad v_5$

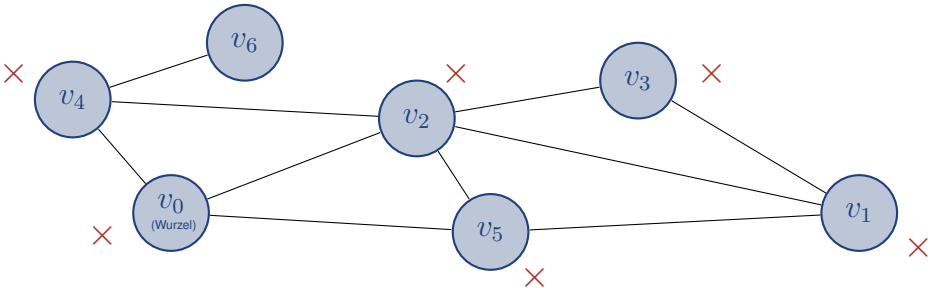
# Rekursive Tiefensuche

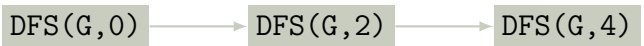



`DFS(G,0)` → `DFS(G,2)`

Ausgabe: $v_0 \quad v_2 \quad v_1 \quad v_3 \quad v_5$

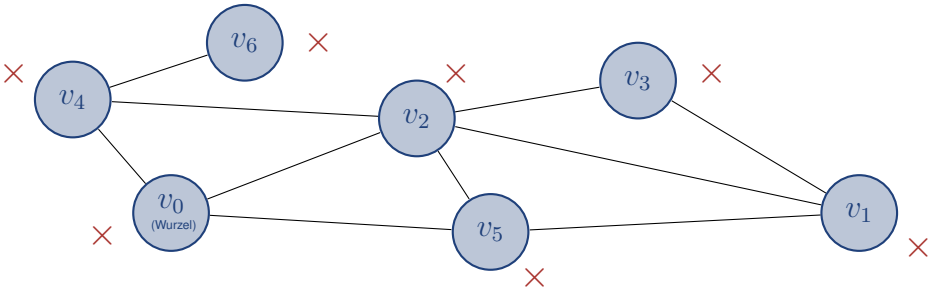
# Rekursive Tiefensuche




Ausgabe: $v_0 \quad v_2 \quad v_1 \quad v_3 \quad v_5 \quad v_4$

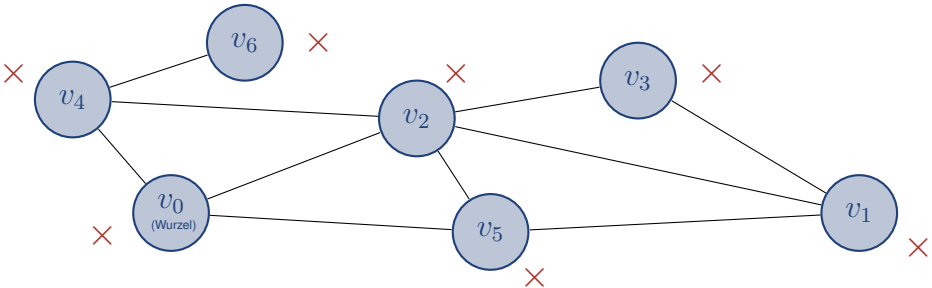
# Rekursive Tiefensuche




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

Ausgabe: $v_0 \quad v_2 \quad v_1 \quad v_3 \quad v_5 \quad v_4 \quad v_6$

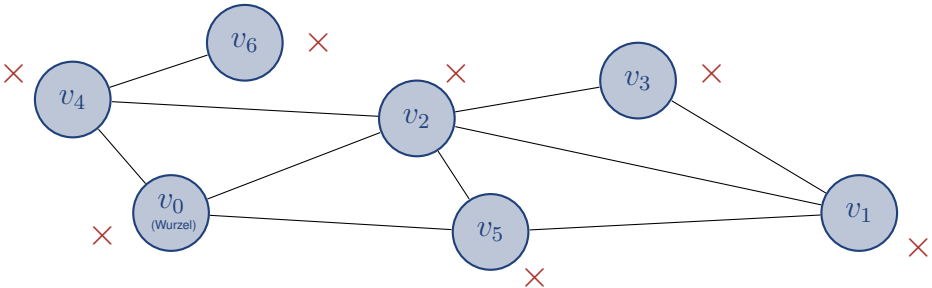
# Rekursive Tiefensuche




`DFS(G,0)` → `DFS(G,2)` → `DFS(G,4)`

Ausgabe: $v_0 \quad v_2 \quad v_1 \quad v_3 \quad v_5 \quad v_4 \quad v_6$

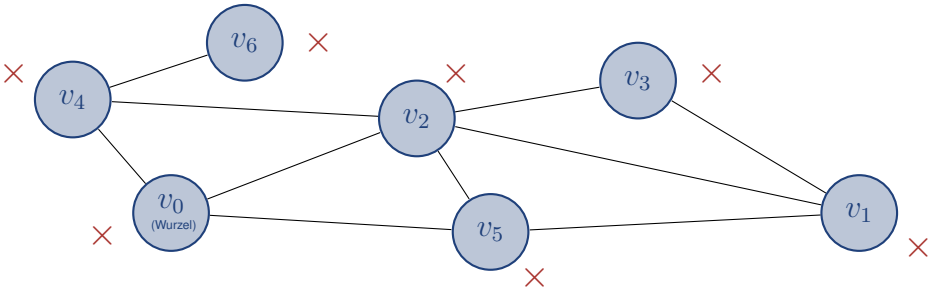
# Rekursive Tiefensuche




```
DFS(G,0)  →  DFS(G,2)
```

Ausgabe: $v_0 \quad v_2 \quad v_1 \quad v_3 \quad v_5 \quad v_4 \quad v_6$

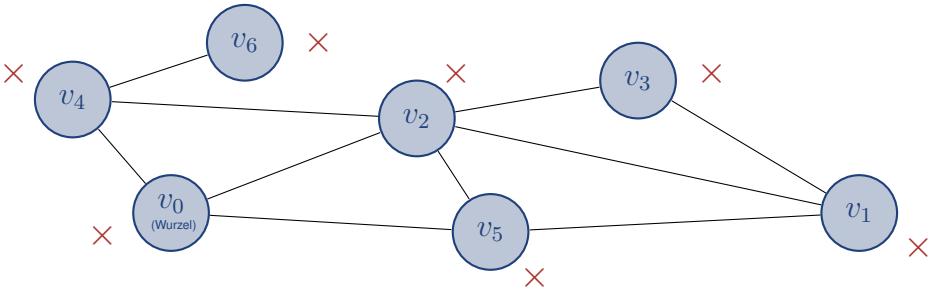
# Rekursive Tiefensuche




```
DFS(G,0)
```

Ausgabe: $v_0 \quad v_2 \quad v_1 \quad v_3 \quad v_5 \quad v_4 \quad v_6$

# Rekursive Tiefensuche




Ausgabe: $v_0 \quad v_2 \quad v_1 \quad v_3 \quad v_5 \quad v_4 \quad v_6$

# Rekursive Tiefensuche

## Anwendungen

# Anwendungen

## Ist ein Graph verbunden?

# Anwendungen

## Ist ein Graph verbunden?

⇨ Tiefensuche von beliebigem Knoten; wurden am Ende alle Knoten besucht?

# Anwendungen

**Ist ein Graph verbunden?**

⇨ Tiefensuche von beliebigem Knoten; wurden am Ende alle Knoten besucht?

**Ist Knoten $w$ erreichbar von Knoten $v$?**

# Anwendungen

**Ist ein Graph verbunden?**

⇨ Tiefensuche von beliebigem Knoten; wurden am Ende alle Knoten besucht?

**Ist Knoten $w$ erreichbar von Knoten $v$?**

⇨ Tiefensuche von Knoten $v$; wurde $w$ am Ende besucht?

# Anwendungen

**Ist ein Graph verbunden?**

⇨ Tiefensuche von beliebigem Knoten; wurden am Ende alle Knoten besucht?

**Ist Knoten $w$ erreichbar von Knoten $v$?**

⇨ Tiefensuche von Knoten $v$; wurde $w$ am Ende besucht?

**Ist ein Graph 2-färbbar?**

# Anwendungen

**Ist ein Graph verbunden?**

⇨ Tiefensuche von beliebigem Knoten; wurden am Ende alle Knoten besucht?

**Ist Knoten $w$ erreichbar von Knoten $v$?**

⇨ Tiefensuche von Knoten $v$; wurde $w$ am Ende besucht?

**Ist ein Graph 2-färbbar?**

⇨ Tiefensuche von beliebigem Knoten und färbe Level abwechslend

# Anwendungen

**Ist ein Graph verbunden?**

⇨ Tiefensuche von beliebigem Knoten; wurden am Ende alle Knoten besucht?

**Ist Knoten $w$ erreichbar von Knoten $v$?**

⇨ Tiefensuche von Knoten $v$; wurde $w$ am Ende besucht?

**Ist ein Graph 2-färbbar?**

⇨ Tiefensuche von beliebigem Knoten und färbe Level abwechslend

**Enthält ein Graph einen Kreis?**

# Anwendungen

**Ist ein Graph verbunden?**

⇨ Tiefensuche von beliebigem Knoten; wurden am Ende alle Knoten besucht?

**Ist Knoten $w$ erreichbar von Knoten $v$?**

⇨ Tiefensuche von Knoten $v$; wurde $w$ am Ende besucht?

**Ist ein Graph 2-färbbar?**

⇨ Tiefensuche von beliebigem Knoten und färbe Level abwechslend

**Enthält ein Graph einen Kreis?**

⇨ Tiefensuche von beliebigem Knoten; gibt es eine Rück-Kante?

# Rekursive Tiefensuche

## Färben von Graphen

# Färben von Graphen

Betrachte beliebigen Graph

- Kann dieser mit zwei Farben gefärbt werden?
- Verbundene Knoten („Nachbarn“) haben verschiedene Farben

# Färben von Graphen

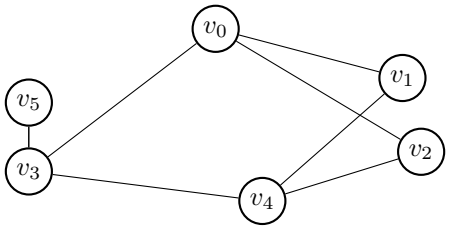
Betrachte beliebigen Graph

- Kann dieser mit zwei Farben gefärbt werden?
- Verbundene Knoten („Nachbarn“) haben verschiedene Farben
- Berechne dies rekursiv

# Färben von Graphen

Betrachte beliebigen Graph

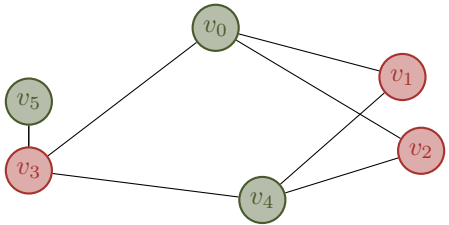
- Kann dieser mit zwei Farben gefärbt werden?
- Verbundene Knoten („Nachbarn“) haben verschiedene Farben
- Berechne dies rekursiv



# Färben von Graphen

Betrachte beliebigen Graph

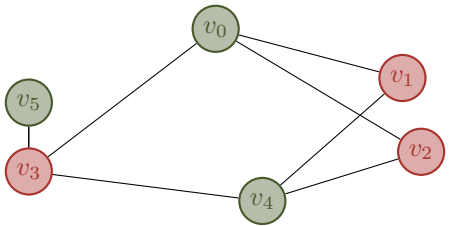
- Kann dieser mit zwei Farben gefärbt werden?
- Verbundene Knoten („Nachbarn“) haben verschiedene Farben
- Berechne dies rekursiv



# Färben von Graphen

Betrachte beliebigen Graph

- Kann dieser mit zwei Farben gefärbt werden?
- Verbundene Knoten („Nachbarn“) haben verschiedene Farben
- Berechne dies rekursiv

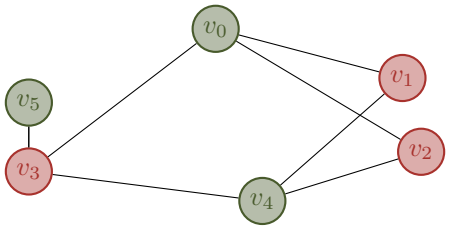



- Liste `color` statt `visited`

# Färben von Graphen

Betrachte beliebigen Graph

- Kann dieser mit zwei Farben gefärbt werden?
- Verbundene Knoten („Nachbarn“) haben verschiedene Farben
- Berechne dies rekursiv

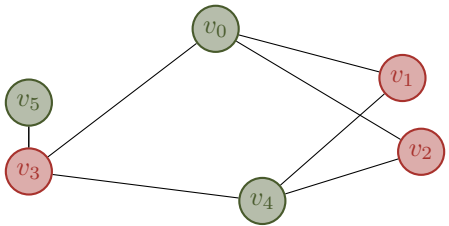



- Liste `color` statt `visited`
- 0: nicht besucht

# Färben von Graphen

Betrachte beliebigen Graph

- Kann dieser mit zwei Farben gefärbt werden?
- Verbundene Knoten („Nachbarn“) haben verschiedene Farben
- Berechne dies rekursiv

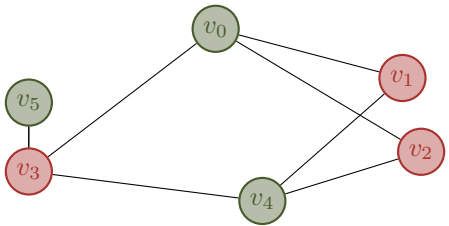



- Liste `color` statt `visited`
- 0: nicht besucht
- 1: grün gefärbt

# Färben von Graphen

Betrachte beliebigen Graph

- Kann dieser mit zwei Farben gefärbt werden?
- Verbundene Knoten („Nachbarn“) haben verschiedene Farben
- Berechne dies rekursiv




- Liste `color` statt `visited`
- 0: nicht besucht
- 1: grün gefärbt
- 2: rot gefärbt

# Färben von Graphen

Wir verwenden rekursive Tiefensuche

- Alle Nachbarn von `current` kriegen die Farbe, die `current` nicht hat

# Färben von Graphen

Wir verwenden rekursive Tiefensuche

- Alle Nachbarn von `current` kriegen die Farbe, die `current` nicht hat
- Hat Nachbar bereits Farbe von `current`, ist Färbung nicht möglich

# Färben von Graphen

Wir verwenden rekursive Tiefensuche

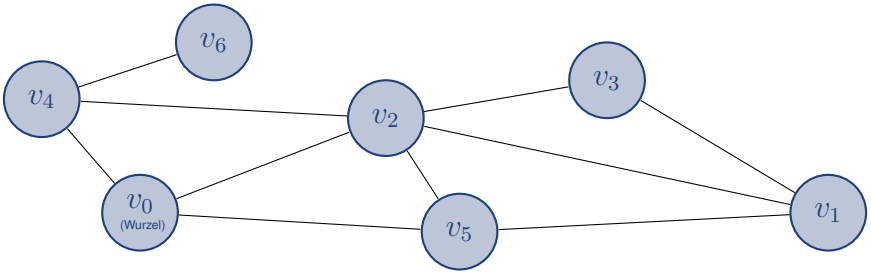
- Alle Nachbarn von `current` kriegen die Farbe, die `current` nicht hat
- Hat Nachbar bereits Farbe von `current`, ist Färbung nicht möglich

```
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("Färbung nicht möglich.")
            return
```

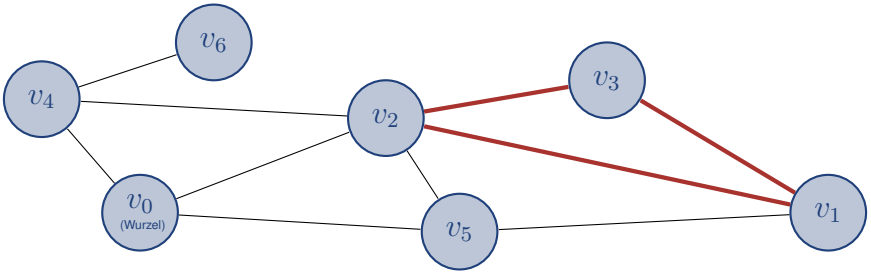
# Rekursive Tiefensuche

## Kreise finden

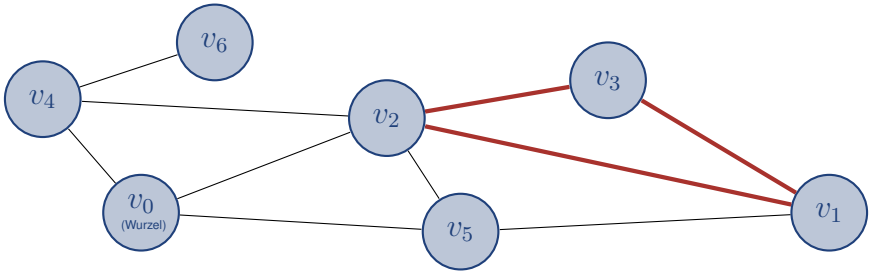
# Kreise finden



# Kreise finden

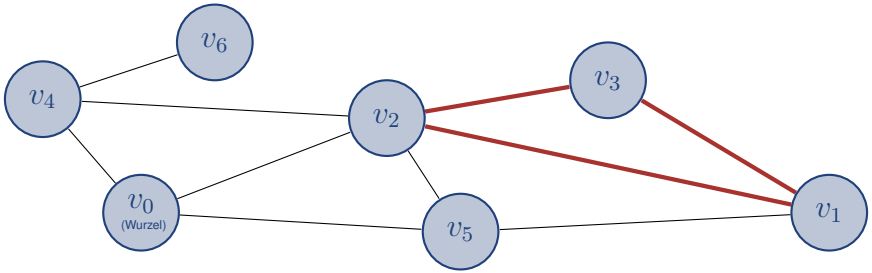


# Kreise finden



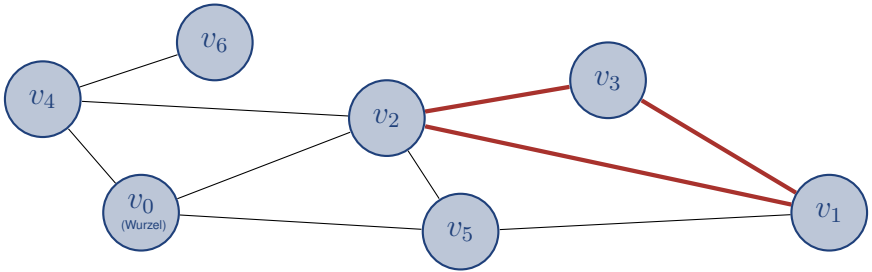

- Tiefensuche liefert dies

# Kreise finden



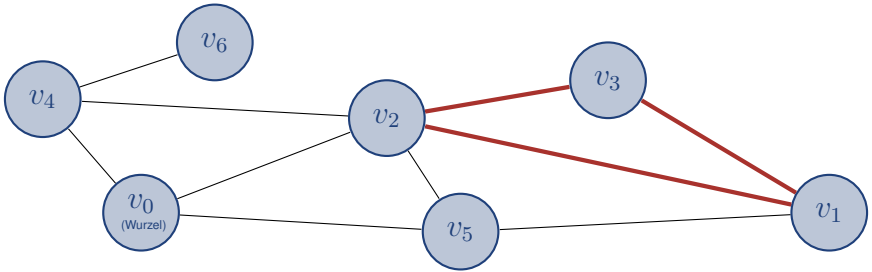

- Tiefensuche liefert dies
- Traversiere Graphen wie davor

# Kreise finden



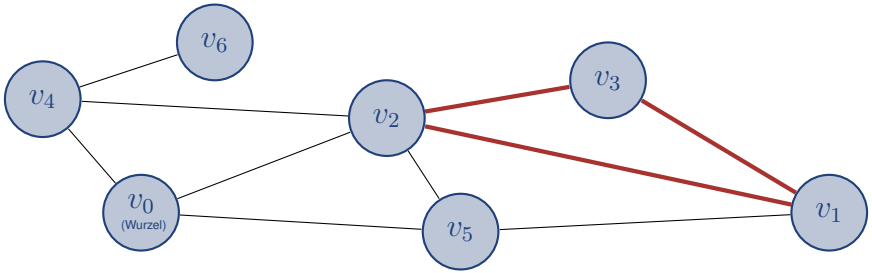

- Tiefensuche liefert dies
- Traversiere Graphen wie davor
- Gibt es eine Kante zu einem Knoten, den wir schon gesehen haben?

# Kreise finden




- Tiefensuche liefert dies
- Traversiere Graphen wie davor
- Gibt es eine Kante zu einem Knoten, den wir schon gesehen haben?
- Rück-Kante

# Kreise finden




- Tiefensuche liefert dies
- Traversiere Graphen wie davor
- Gibt es eine Kante zu einem Knoten, den wir schon gesehen haben?
- Rück-Kante
- **Vorsicht:** Einzelne Kante ist natürlich kein Kreis

# Kreise finden

Berechne, ob Graph einen Kreis enthält

# Kreise finden

Berechne, ob Graph einen Kreis enthält

Erweitere Tiefensuche so, dass der Vorgänger berücksichtigt wird

```
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("Kreis gefunden.")
            return
```

# Danke für die Aufmerksamkeit