

# Graphen I

Gustavo Crivelli, Beini Ma, Matthias Schimek, Matthias Schmitt

ICPC-Praktikum 2015 - Graphen I

Einleitung

BFS

DFS

Iterative Tiefensuche

Starke Zusammenhangskomponenten

Bipartite Graphen

Brücke

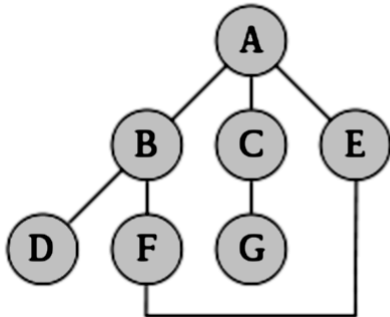
## Idee des Algorithmus

- DFS Algorithmus mit begrenzt Suchtiefe, repetiert mit steigenden Werte.
- Besucht neue Knoten in BFS Ordnung, hat aber geringer Speicherverbrauch als BFS

- BFS Speicherverbrauch:  $\mathcal{O}(b^d)$
- Iterative Tiefensuche Speicherverbrauch:  $\mathcal{O}(bd)$

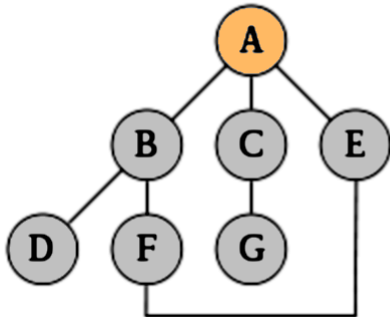
Wo  $b$  (*branching factor*) die durchschnittliche Zahl Nachbarn eines Knotens ist,  
und  $d$  die Tiefe des flachsten Ziels.

- Laufzeit:  $\mathcal{O}(b^d)$



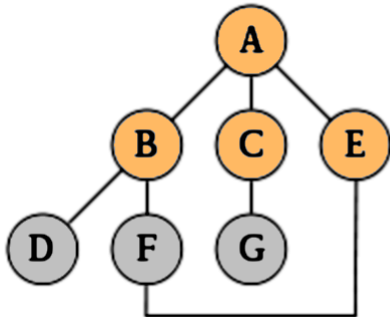
Anfang: A

# Iterative Tiefensuche



Anfang: A

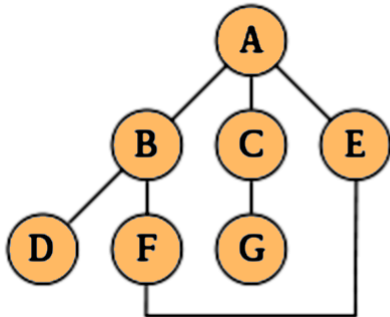
$d = 0$ : A



Anfang: A

$d = 0$ : A

$d = 1$ : ABCE

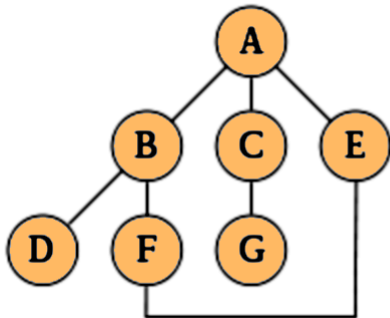


Anfang: A

d = 0: A

d = 1: ABCE

d = 2: ABDFCGEF



Anfang: A

d = 0: A

d = 1: ABCE

d = 2: ABDFCGEF

d = 3: ABDFECGEFB



# Iterative Tiefensuche

## Rekursive begrenzt DFS Algorithmus

```
vector< vector<int> > adjList;  
const int LIMITBREAK = -1;  
  
int iterativeDFS(int origin)  
{  
    int depth = 0, found = LIMITBREAK;  
    while(found == LIMITBREAK)  
    {  
        found = limitedDFS(origin, depth);  
        depth++;  
    }  
    return found;  
}
```

# Iterative Tiefensuche

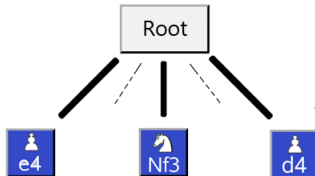
## Rekursive begrenzt DFS Algorithmus

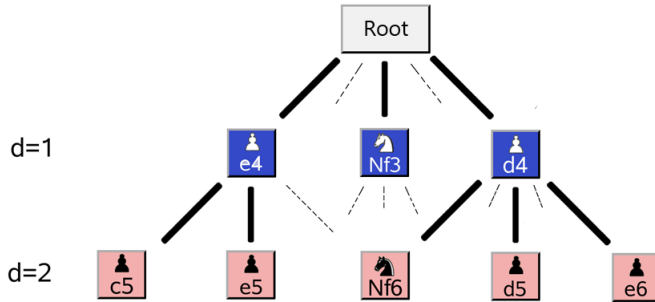
```
int limitedDFS(int v, int depth)
{
    if(depth == 0 && v == goal)
        return v;
    else if(depth > 0)
    {
        for(int i = 0; i < adjList[v].size(); i++)
        {
            int found = limitedDFS(adjList[v][i], depth-1);
            if(found != LIMITBREAK)
                return found;
        }
    }
    return LIMITBREAK;
}
```

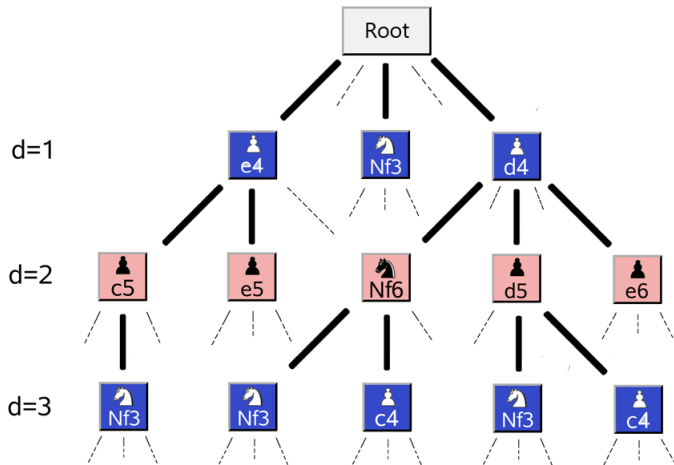
## Schach

- Besonders sinnvoll für Probleme, die hoch branching factor haben, z.B.: **Schach**
- Im Durchschnitt hat ein Schachspieler für jeder Position 35~38 mögliche Bewegungen
- Iterative Tiefensuche findet die beste Bewegung bis zum eine Tiefe  $T$ , innerhalb ein Zeitlimit

d=1







# SCCs finden mittels DFS

## Beispielaufgabe

### UVa 11383 Come and Go

In einer Stadt gibt es  $N$  Kreuzungen, die durch Straßen verbunden sind. Da man in der Stadt von einem Punkt (Kreuzung) zu jedem anderen kommen möchte, sollte es eine Verbindung zwischen zwei beliebigen Kreuzungen geben.

Für eine gegebene Stadt mit  $N$  Kreuzungen und  $M$  Straßen soll entschieden werden, ob dies möglich ist.

### Definition Strongly Connected Components SCC

In einem gerichteten Graph  $G = (V, E)$ , wird  $V' \subseteq V$  starke Zusammenhangskomponente (SCC) genannt, wenn zwischen je zwei Knoten in  $V'$  ein Pfad existiert.

# SCCs finden mittels DFS

## Beispielaufgabe

### UVa 11383 Come and Go

In einer Stadt gibt es  $N$  Kreuzungen, die durch Straßen verbunden sind. Da man in der Stadt von einem Punkt (Kreuzung) zu jedem anderen kommen möchte, sollte es eine Verbindung zwischen zwei beliebigen Kreuzungen geben.

Für eine gegebene Stadt mit  $N$  Kreuzungen und  $M$  Straßen soll entschieden werden, ob dies möglich ist.

### Definition Strongly Connected Components SCC

In einem gerichteten Graph  $G = (V, E)$ , wird  $V' \subseteq V$  starke Zusammenhangskomponente (SCC) genannt, wenn zwischen je zwei Knoten in  $V'$  ein Pfad existiert.



- Zum Lösen der Aufgabe untersuchen, ob das Straßennetz der Stadt aus einer oder mehreren SCCs besteht.
- $\Rightarrow$  Benötigen effizienten Algorithmus zum Finden von SCCs

## Algorithmus von Tarjan für SCCs

- Wurde von Robert Tarjan gefunden
- Basiert auf dem Konzept der DFS
- Laufzeit:  $\mathcal{O}(|V| + |E|)$

- Zum Lösen der Aufgabe untersuchen, ob das Straßennetz der Stadt aus einer oder mehreren SCCs besteht.
- $\Rightarrow$  Benötigen effizienten Algorithmus zum Finden von SCCs

## Algorithmus von Tarjan für SCCs

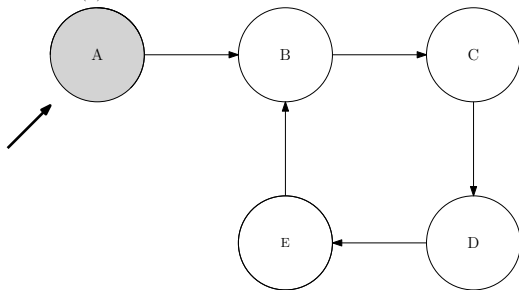
- Wurde von Robert Tarjan gefunden
- Basiert auf dem Konzept der DFS
- Laufzeit:  $\mathcal{O}(|V| + |E|)$

# SCCs finden mittels DFS

- 
- $\text{dfs num}(A) = 0$
  - $\text{dfs low}(A) = 0$

STACK:

- A



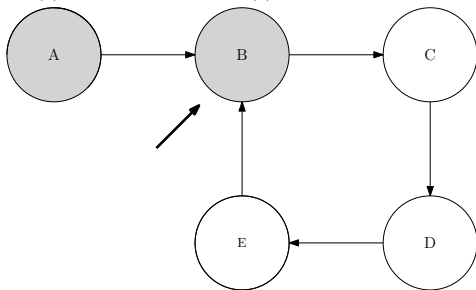
# SCCs finden mittels DFS

- 
- $\text{dfs num}(A) = 0$
  - $\text{dfs low}(A) = 0$

- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = 1$

STACK:

- A
- B



# SCCs finden mittels DFS

•  $\text{dfs num}(A) = 0$

•  $\text{dfs low}(A) = 0$

•  $\text{dfs num}(B) = 1$

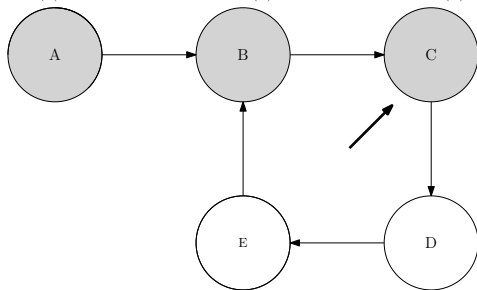
•  $\text{dfs low}(B) = 1$

•  $\text{dfs num}(C) = 2$

•  $\text{dfs low}(C) = 2$

STACK:

- A
- B
- C



# SCCs finden mittels DFS

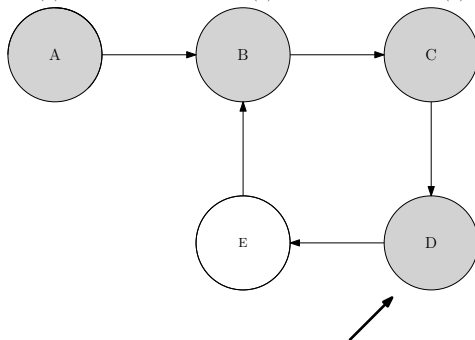
- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$

- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = 1$

- $\text{dfs num}(C) = 2$
- $\text{dfs low}(C) = 2$

STACK:

- A
- B
- C
- D



- $\text{dfs num}(D) = 3$
- $\text{dfs low}(D) = 3$

# SCCs finden mittels DFS

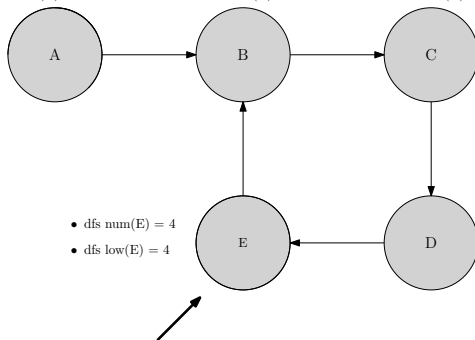
- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$

- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = 1$

- $\text{dfs num}(C) = 2$
- $\text{dfs low}(C) = 2$

STACK:

- A
- B
- C
- D
- E



- $\text{dfs num}(E) = 4$
- $\text{dfs low}(E) = 4$

- $\text{dfs num}(D) = 3$
- $\text{dfs low}(D) = 3$

# SCCs finden mittels DFS

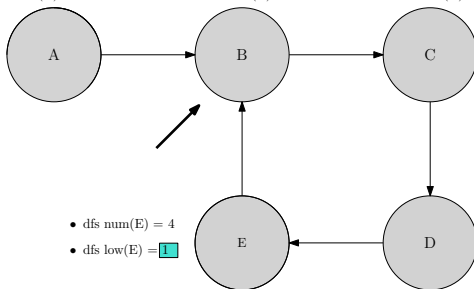
- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$

- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = 1$

- $\text{dfs num}(C) = 2$
- $\text{dfs low}(C) = 2$

STACK:

- A
- B
- C
- D
- E



- $\text{dfs num}(E) = 4$
- $\text{dfs low}(E) = 1$

- $\text{dfs num}(D) = 3$
- $\text{dfs low}(D) = 3$



# SCCs finden mittels DFS

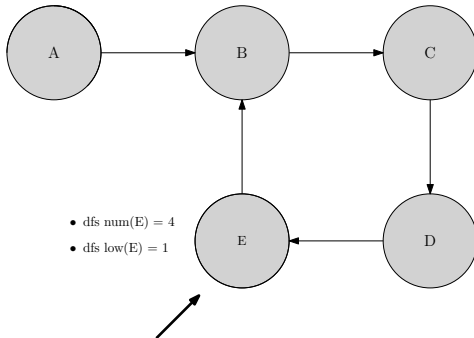
- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$

- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = 1$

- $\text{dfs num}(C) = 2$
- $\text{dfs low}(C) = 2$

STACK:

- A
- B
- C
- D
- E



- $\text{dfs num}(E) = 4$
- $\text{dfs low}(E) = 1$

- $\text{dfs num}(D) = 3$
- $\text{dfs low}(D) = 1$

# SCCs finden mittels DFS

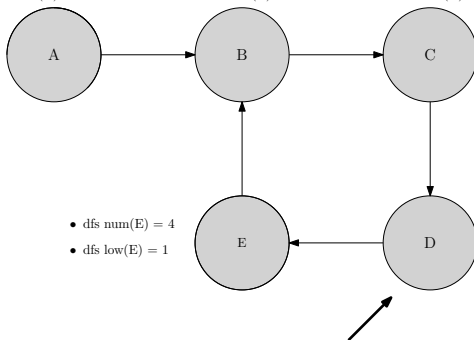
- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$

- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = 1$

- $\text{dfs num}(C) = 2$
- $\text{dfs low}(C) = 1$

STACK:

- A
- B
- C
- D
- E



- $\text{dfs num}(E) = 4$
- $\text{dfs low}(E) = 1$

- $\text{dfs num}(D) = 3$
- $\text{dfs low}(D) = 1$

# SCCs finden mittels DFS

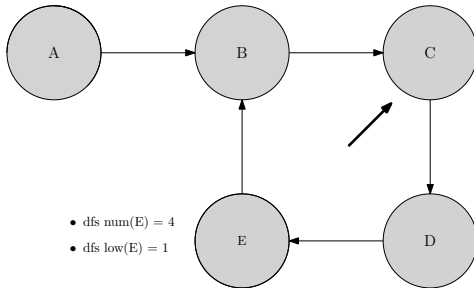
- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$

- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = 1$

- $\text{dfs num}(C) = 2$
- $\text{dfs low}(C) = 1$

STACK:

- A
- B
- C
- D
- E



- $\text{dfs num}(E) = 4$
- $\text{dfs low}(E) = 1$

- $\text{dfs num}(D) = 3$
- $\text{dfs low}(D) = 1$

# SCCs finden mittels DFS

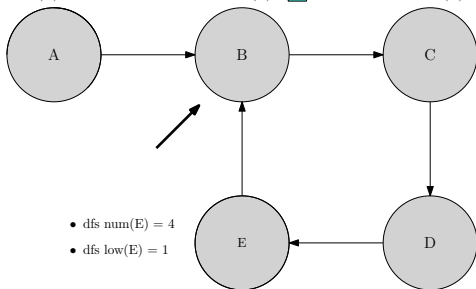
- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$

- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = 1$

- $\text{dfs num}(C) = 2$
- $\text{dfs low}(C) = 1$

STACK:

- A
- B
- C
- D



- $\text{dfs num}(E) = 4$
- $\text{dfs low}(E) = 1$

- $\text{dfs num}(D) = 3$
- $\text{dfs low}(D) = 1$

# SCCs finden mittels DFS

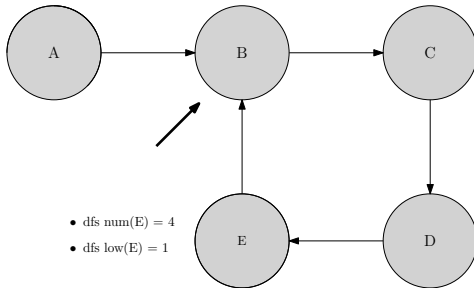
- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$

- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = 1$

- $\text{dfs num}(C) = 2$
- $\text{dfs low}(C) = 1$

STACK:

- A
- B
- C



- $\text{dfs num}(E) = 4$
- $\text{dfs low}(E) = 1$

- $\text{dfs num}(D) = 3$
- $\text{dfs low}(D) = 1$

# SCCs finden mittels DFS

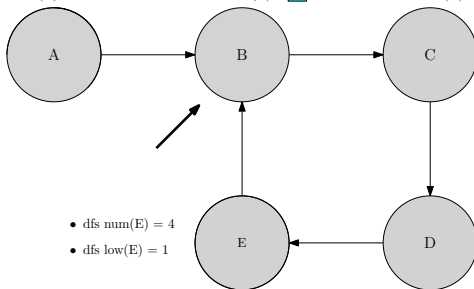
- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$

- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = 1$

- $\text{dfs num}(C) = 2$
- $\text{dfs low}(C) = 1$

STACK:

- A
- B

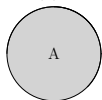


- $\text{dfs num}(E) = 4$
- $\text{dfs low}(E) = 1$

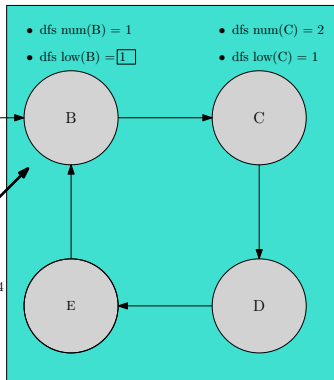
- $\text{dfs num}(D) = 3$
- $\text{dfs low}(D) = 1$

# SCCs finden mittels DFS

- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$



- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = \boxed{1}$
- $\text{dfs num}(C) = 2$
- $\text{dfs low}(C) = 1$



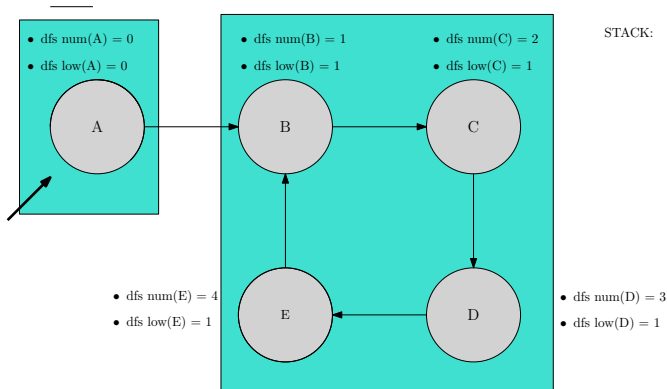
- $\text{dfs num}(E) = 4$
- $\text{dfs low}(E) = 1$

STACK:

- A

- $\text{dfs num}(D) = 3$
- $\text{dfs low}(D) = 1$

# SCCs finden mittels DFS





# SCCs finden mittels DFS

## Sourcecode

```
void findSCC(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // initialize
    S.push_back(u); visited[u] = 1;

    for (int j = 0; j < AdjacencyList[u].size(); j++) {
        int v = AdjacencyList[u][j];
        if (dfs_num[v.first] == UNVISITED) // not yet visited by DFS
            findSCC(v.first);
        if (visited[v.first]) // belongs to current SCC
            dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
    }
    if (dfs_low[u] == dfs_num[u]) { // root of current SCC
        cout << "SCC_" << ++numSCC; // print vertices in SCC
        while(true) {
            int v = S.back(); S.pop_back(); visited[v] = 0;
            cout << "_" << v;
            if (u == v) break;
        }
        cout << endl;
    }
}
```

- **findSCC(int u)** findet alle SCCs, die von Knoten **u** aus erreichbar sind.
- Für vollständige Liste an SCCs **findSCC(int u)** für alle Knoten eines Graphen laufen lassen.

### Separatoren und Brücken in ungerichteten Graphen

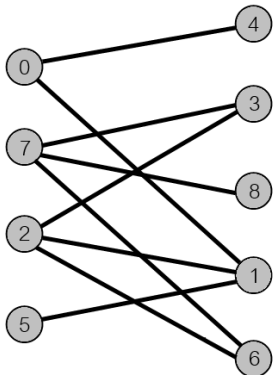
Sei  $G = (V, E)$  ein ungerichteter Graph.

- $G$  heißt **Bipartite** falls sich seine Knoten in zwei disjunkte Teilmengen  $A, B$  aufteilen lassen, so dass es zwischen den Knoten innerhalb einer Teilmenge keine Kanten gibt.
- Das heißt, für jeder Kante  $\{u, v\} \in E$  gilt entweder  $u \in A$  und  $v \in B$  oder  $u \in B$  und  $v \in A$ .

# Bipartite Graphen

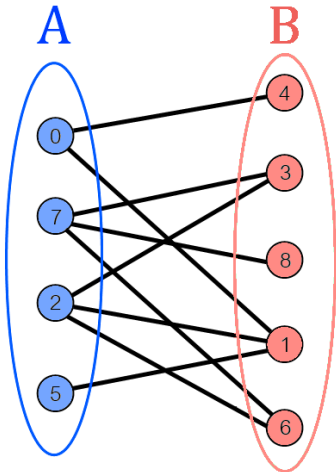
## Beispiel

G



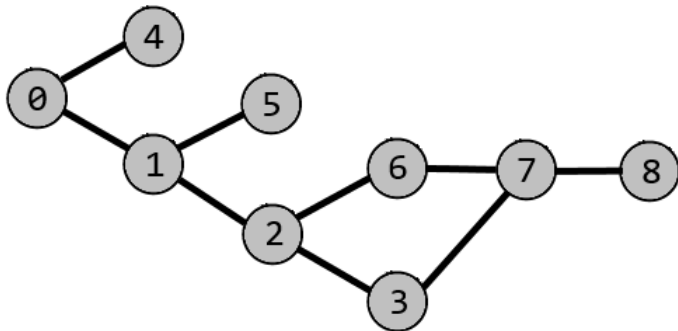
# Bipartite Graphen

## Beispiel



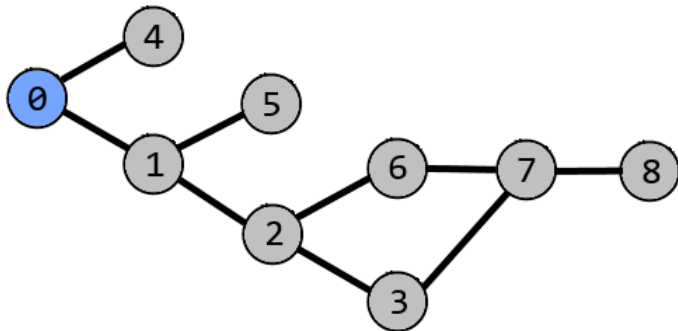
# Bipartite Graph Check

## DFS Algorithmus



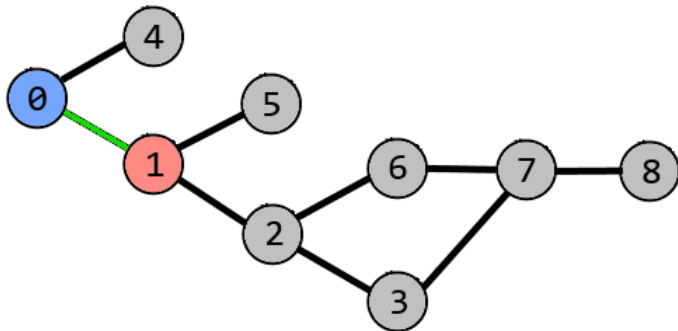
# Bipartite Graph Check

## DFS Algorithmus



# Bipartite Graph Check

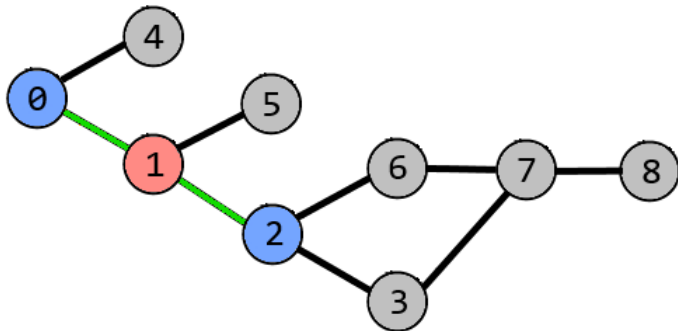
## DFS Algorithmus





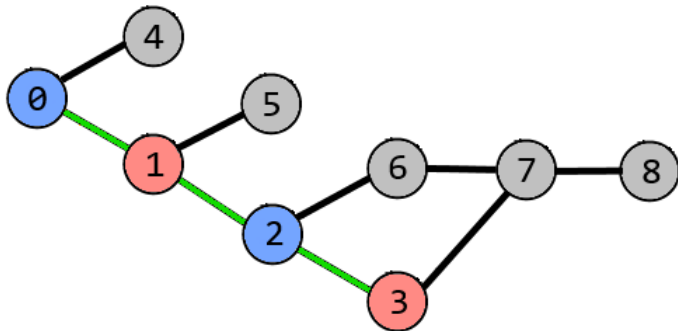
# Bipartite Graph Check

## DFS Algorithmus



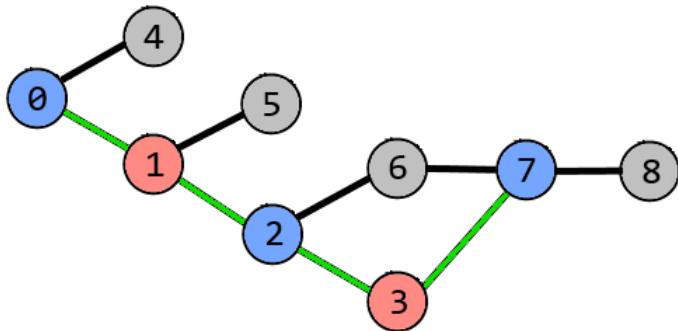
# Bipartite Graph Check

## DFS Algorithmus



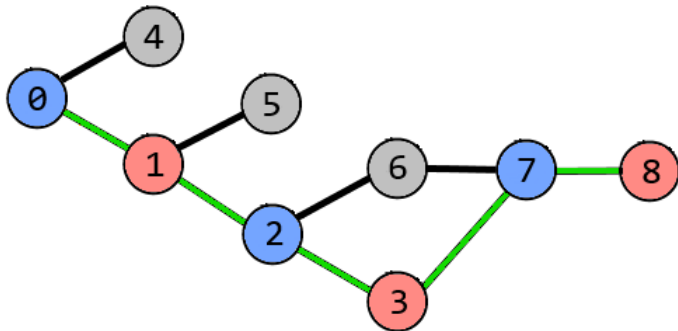
# Bipartite Graph Check

## DFS Algorithmus



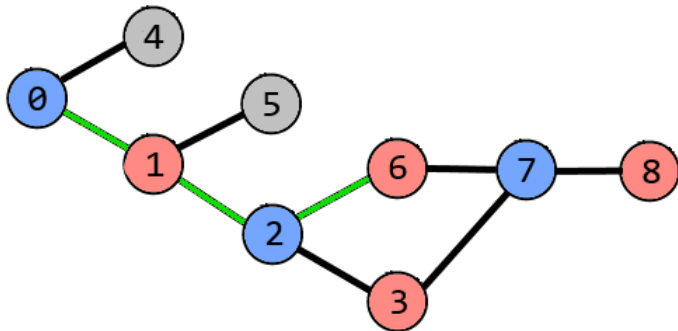
# Bipartite Graph Check

## DFS Algorithmus



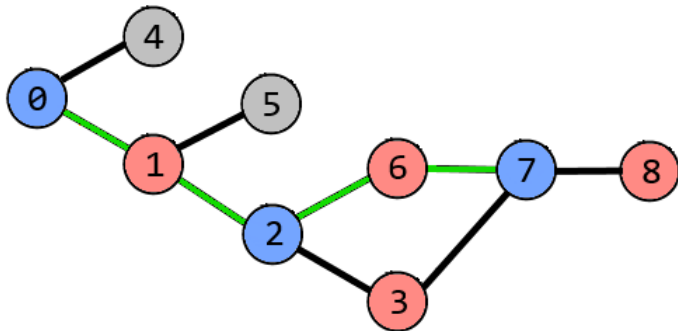
# Bipartite Graph Check

## DFS Algorithmus



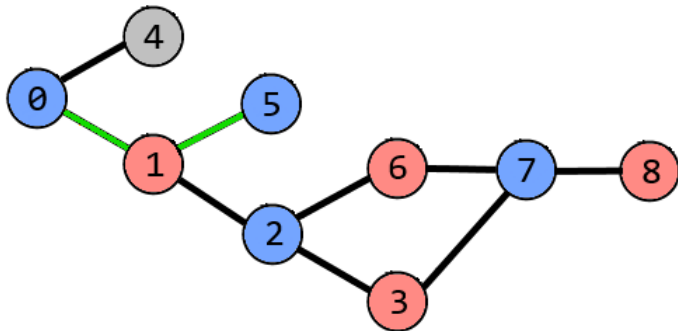
# Bipartite Graph Check

## DFS Algorithmus



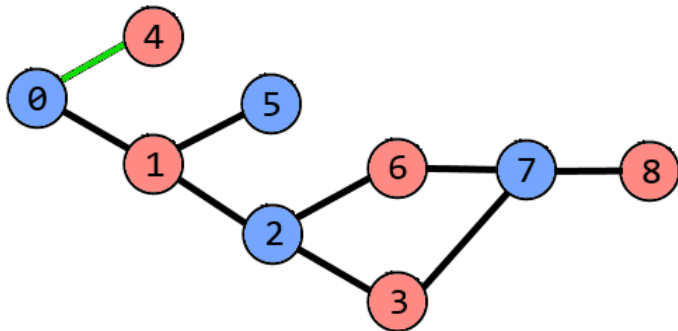
# Bipartite Graph Check

## DFS Algorithmus



# Bipartite Graph Check

## DFS Algorithmus





# Bipartite Graph Check

## DFS Algorithmus

```
vector< vector<int> > adjList;  
vector<int> colorDFS[200];  
const int UNVISITED = -1, NOT_BIP = 0, BIP = 1;  
int solveDFS(int v, int color) //int main() -> solveDFS(0,0)  
{  
    if(colorDFS[v] == UNVISITED)  
    {  
        colorDFS[v] = color;  
        for(int i = 0; i < adjList[v].size(); i++)  
            if(solveDFS(adjList[v][i], 1 - color) == NOT_BIP)  
                return NOT_BIP;  
    }  
    else if(colorDFS[v] != color)  
        return NOT_BIP;  
    return BIP;  
}
```

# Bipartite Graph Check

## BFS Algorithmus

```
vector< vector<int> > adjList;  
const int UNVISITED = -1, NOT_BIP = 0, BIP = 1;  
  
int checkBFS(int nVertex) {  
    vector<int> color(nVertex, UNVISITED);  
    queue<int> q; q.push(0);  
    while(!q.empty()) {  
        int vertex = q.front(); q.pop();  
        for(int i = 0; i < adjList[vertex].size(); i++) {  
            int next = adjList[vertex][i];  
            if(color[next] == UNVISITED) {  
                color[next] = 1 - color[vertex];  
                q.push(next); }  
            else if(color[next] == color[vertex])  
                return NOT_BIP;  
        }  
    }  
    return BIP;  
}
```

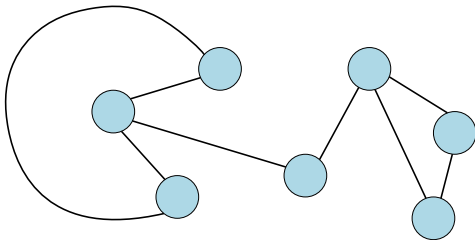
### Separatoren und Brücken in ungerichteten Graphen

Sei  $G = (V, E)$  ein ungerichteter Graph.

- Ein Knoten  $v \in V$  heißt **Separator** von  $G$ , wenn durch sein Entfernen bestehende Zusammenhangskomponenten aufgetrennt werden.
- Eine Kante  $\{u, v\} \in E$  heißt **Brücke**, wenn durch ihr Entfernen  $u$  und  $v$  in verschiedenen Zusammenhangskomponenten liegen.

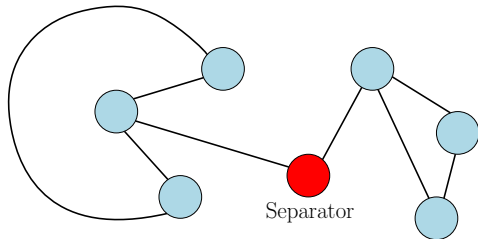
# Brücken und Separatoren

## Beispiel



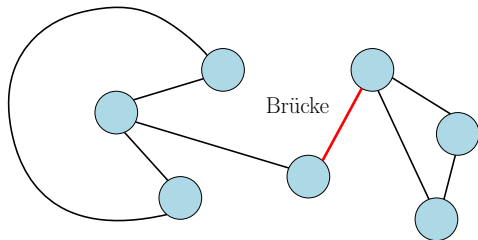
# Brücken und Separatoren

## Beispiel



# Brücken und Separatoren

## Beispiel



- Naive Herangehensweise:
  1. Entferne einen Knoten/Kante
  2. Prüfe mittels DFS/BFS ob sich eine neue Zusammenhangskomponente ergeben hat
  3. Wiederhole Schritt 1 für alle Knoten/Kanten
- Laufzeit:  $\mathcal{O}(|V| \cdot (|V| + |E|))$  bzw.  $\mathcal{O}(|E| \cdot (|V| + |E|))$
- Es existiert Algorithmus in  $\mathcal{O}(|V| + |E|)$
- Basiert auf DFS und ähnelt Algorithmus zum Finden von SCCs

- Führe eine DFS im Graph durch.
- Besuchte Knoten erhalten zwei Nummern:
  1. **dfs\_num(u)**: Speichert Schritt, in dem Knoten  $u$  von DFS besucht wurde.
  2. **dfs\_low(u)**: Niedrigster Wert von **dfs\_low**, der von Knoten  $u$  aus erreicht werden kann.
- Wenn **dfs\_low(v)  $\geq$  dfs\_num(u)**, dann ist  $u$  ein Separator
  - Von  $v$  kann kein Knoten  $w$  "vor"  $u$  erreicht werden.
  - "vor" bedeutet: (**dfs\_num(w)  $>$  dfs\_num(u)**)
  - Um Knoten  $w$  "vor"  $u$  zu erreichen, muss man durch  $u$  laufen.
  - $\Rightarrow u$  teilt Graph in zwei Zusammenhangskomponenten.
  - (Spezialfall: Gilt nicht, wenn  $u$  Wurzel der DFS)



- Führe eine DFS im Graph durch.
- Besuchte Knoten erhalten zwei Nummern:
  1. **dfs\_num(u)**: Speichert Schritt, in dem Knoten  $u$  von DFS besucht wurde.
  2. **dfs\_low(u)**: Niedrigster Wert von **dfs\_low**, der von Knoten  $u$  aus erreicht werden kann.
- Wenn **dfs\_low(v)  $\geq$  dfs\_num(u)**, dann ist  $u$  ein Separator
  - Von  $v$  kann kein Knoten  $w$  "vor"  $u$  erreicht werden.
  - "vor" bedeutet: (**dfs\_num(w)  $>$  dfs\_num(u)**)
  - Um Knoten  $w$  "vor"  $u$  zu erreichen, muss man durch  $u$  laufen.
  - $\Rightarrow u$  teilt Graph in zwei Zusammenhangskomponenten.
  - (Spezialfall: Gilt nicht, wenn  $u$  Wurzel der DFS)

- Führe eine DFS im Graph durch.
- Besuchte Knoten erhalten zwei Nummern:
  1. **dfs\_num(u)**: Speichert Schritt, in dem Knoten  $u$  von DFS besucht wurde.
  2. **dfs\_low(u)**: Niedrigster Wert von **dfs\_low**, der von Knoten  $u$  aus erreicht werden kann.
- Wenn **dfs\_low(v)  $\geq$  dfs\_num(u)**, dann ist  $u$  ein Separator
  - Von  $v$  kann kein Knoten  $w$  "vor"  $u$  erreicht werden.
  - "vor" bedeutet: (**dfs\_num(w)  $>$  dfs\_num(u)**)
  - Um Knoten  $w$  "vor"  $u$  zu erreichen, muss man durch  $u$  laufen.
  - $\Rightarrow u$  teilt Graph in zwei Zusammenhangskomponenten.
  - (Spezialfall: Gilt nicht, wenn  $u$  Wurzel der DFS)

- Führe eine DFS im Graph durch.
- Besuchte Knoten erhalten zwei Nummern:
  1. **dfs\_num(u)**: Speichert Schritt, in dem Knoten  $u$  von DFS besucht wurde.
  2. **dfs\_low(u)**: Niedrigster Wert von **dfs\_low**, der von Knoten  $u$  aus erreicht werden kann.
- Wenn **dfs\_low(v)  $\geq$  dfs\_num(u)**, dann ist  $u$  ein Separator
  - Von  $v$  kann kein Knoten  $w$  "vor"  $u$  erreicht werden.
  - "vor" bedeutet: (**dfs\_num(w)  $>$  dfs\_num(u)**)
  - Um Knoten  $w$  "vor"  $u$  zu erreichen, muss man durch  $u$  laufen.
  - $\Rightarrow u$  teilt Graph in zwei Zusammenhangskomponenten.
  - (Spezialfall: Gilt nicht, wenn  $u$  Wurzel der DFS)

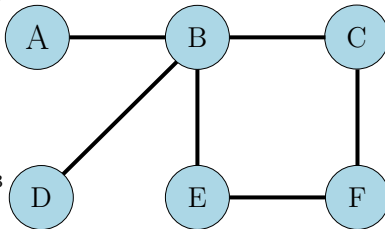
- Besuchte Knoten erhalten zwei Nummern:
  1. **dfs\_num(u)**: Speichert Schritt, in dem Knoten  $u$  von DFS besucht wurde.
  2. **dfs\_low(u)**: Niedrigster Wert von **dfs\_low**, der von Knoten  $u$  aus erreicht werden kann.
- Wenn **dfs\_low(v) > dfs\_num(u)**, dann ist  $\{u, v\}$  eine Brücke
  - Von  $v$  kann Knoten  $v$  nur über die Kante  $u, v$  erreicht werden.
  - Ansonsten **dfs\_low(v) seq dfs\_num(u)**.
  - $\Rightarrow \{u, v\}$  teilt Graph in zwei Zusammenhangskomponenten.

- Besuchte Knoten erhalten zwei Nummern:
  1. **dfs\_num(u)**: Speichert Schritt, in dem Knoten  $u$  von DFS besucht wurde.
  2. **dfs\_low(u)**: Niedrigster Wert von **dfs\_low**, der von Knoten  $u$  aus erreicht werden kann.
- Wenn **dfs\_low(v) > dfs\_num(u)**, dann ist  $\{u, v\}$  eine Brücke
  - Von  $v$  kann Knoten  $v$  nur über die Kante  $u, v$  erreicht werden.
  - Ansonsten **dfs\_low(v) seq dfs\_num(u)**.
  - $\Rightarrow \{u, v\}$  teilt Graph in zwei Zusammenhangskomponenten.

# Brücken und Separatoren

## Illustration des Algorithmus'

- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$
- $\text{dfs num}(B) = 1$
- $\text{dfs low}(B) = 1$



- $\text{dfs num}(D) = 3$
- $\text{dfs low}(D) = 3$

- $\text{dfs num}(E) = 4$
- $\text{dfs low}(E) = 1$

- $\text{dfs num}(C) = 2$
- $\text{dfs low}(C) = 2$

- $\text{dfs num}(F) = 5$
- $\text{dfs low}(F) = 1$