Karlsruher Institut für Technologie

Graphen I

Gustavo Crivelli, Beini Ma, Matthias Schimek, Matthias Schmitt

ICPC-Praktikum 2015 - Graphen I

Gliederung

Einleitung

Grundbegriffe der Graphentheorie

Graph

- Ein Graph G ist ein geordnetes Paar $G = (V, E)$
- V Menge von Knoten/Vertices
- E Menge von Kanten/Edges
  - ungerichteter Graph (ohne Mehrfachkanten)
    $E \subseteq \{M \in \mathcal{P}(V) \mid |M| = 2\}$
  - gerichteter Graph (ohne Mehrfachkanten) $E \subseteq V \times V$

Einleitung

Grundbegriffe der Graphentheorie

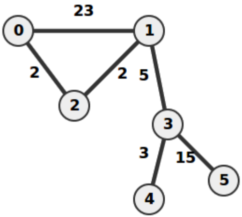
einige bereits bekannte Begriffe

Un/Gerichtet	Un/Gewichtet	Schlinge
(Ein/Ausgangs) Grad	Weg/Pfad	Zyklus/Kreis
Knoten ereichbar/isoliert	Teil/Untergraph	Baum

- einfacher Graph := ungerichteter Graph, ohne Mehrfachkanten und ohne Schleifen
- DAG (directed acyclic graph) := ein gerichteter Graph ohne Zyklus
- vollständiger Graph := maximale Kantenanzahl $|E| = \frac{|V|(|V|-1)}{2}$
- Dichte $dn(G) = \frac{2|E|}{|V|(|V|-1)}$
  - Verhältnis der Kantenanzahl zur Kantenanzahl eines vollständigen Graphen auf gleichvielen Knoten

Einleitung

Implementierungen (gewichtet)



	Adj.matrix	Adj.liste	Edgelist
Code	int[][]	vec<vec<pair<int,int>>>	vec<tuple<int,int,int>
Platz	$\mathcal{O}(\lvert V\rvert^2)$	$\mathcal{O}(\lvert V\rvert + \lvert E\rvert)$	$\mathcal{O}(\lvert E\rvert)$

Adjacency matrix

	0	**1**	**2**	**3**	**4**	**5**
0	0	23	2	0	0	0
1	23	0	2	5	0	0
2	2	2	0	0	0	0
3	0	5	0	0	3	15
4	0	0	0	3	0	0
5	0	0	0	15	0	0

Adjacency list

0 :	(23,1)	(2,2)	
1 :	(23,0)	(2,2)	(5,3)
2 :	(2,0)	(2,1)	
3 :	(5,1)	(3,4)	(15,5)
4 :	(3,3)		
5 :	(15,3)		

Edge list

0 :	23	0	1
1 :	2	0	2
2 :	2	2	1
3 :	5	1	3
4 :	3	3	4
5 :	15	3	5

Fakultät für Informatik
Institut für Theoretische Informatik

Einleitung

Implementierungen Unterschiede

- Adjazenzmatrix
  - hoher Speicherverbrauh
  - Anwenden bei kleinen dichten Graphen
  - über Nachbarn iterieren $\mathcal{O}(|V|)$
  - ICPC Tipp: wenn $V \leq 1000$
- Adjazenzliste
  - kompakter und effizienter, da oft $|E| \ll \frac{|V|(|V|-1)}{2} \in \mathcal{O}(|V|^2)$
  - über alle k Nachbarn iterieren $\mathcal{O}(k)$
- Edgelist
  - sortiert (bspw. nach Gewicht) wichtige Representation für bestimmte Algorithmen
  - verkomplizier viele Algorithmen

Breitensuche
breadth-first search, BFS

Breitensuche

Breitensuche

Traversieren der Knoten der Breite/der Entfernung zum Startknoten nach.

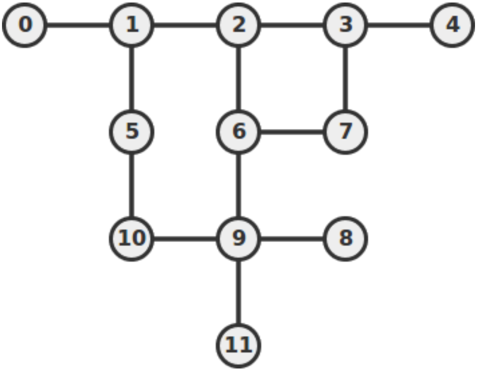
Idee: Besuche den Startknoten, dann dessen Nachbarn, dann deren Nachbarn, usw...

Implementierung: Queue und besuchte Knoten markieren

1. Startknoten markieren und in die Queue einreihen
2. den ersten Knoten **u** aus der Queue nehmen
3. alle nicht markierten Nachbarn von **u** markieren und einreihen
4. gehe zu *2.* wenn Queue nicht leer sonst fertig

Breitensuche

Beispiel

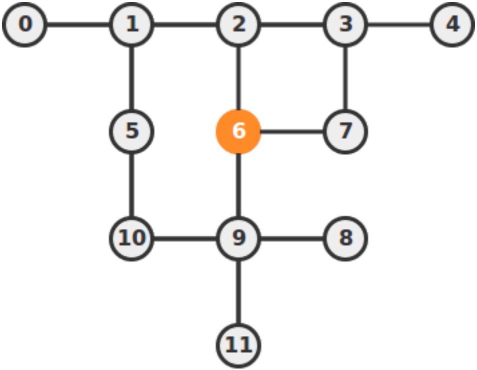


Startknoten s = 6 Queue q = {6} Markierte Knotenmege d = {6}

Fakultät für Informatik
Institut für Theoretische Informatik

Breitensuche

Beispiel

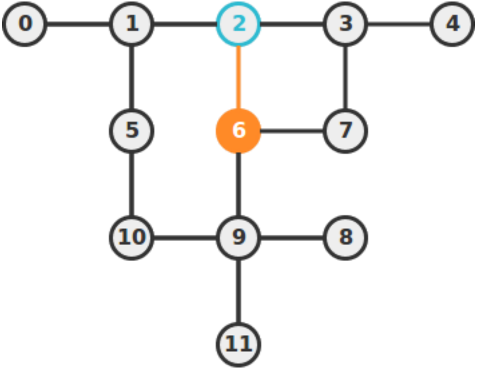


$u = 6$ $\quad q = \{\}$

$d = \{6\}$

Breitensuche

Beispiel

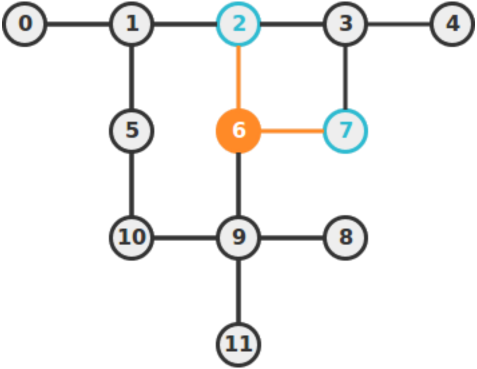


$u = 6$ $q = \{2\}$

$d = \{6, 2\}$

Breitensuche

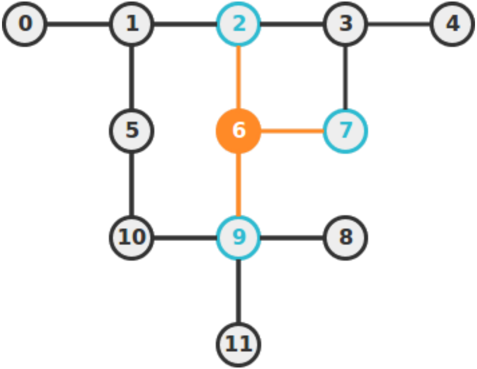
Beispiel



u = 6 q = {2, 7}
d = {6, 2, 7}

Breitensuche

Beispiel

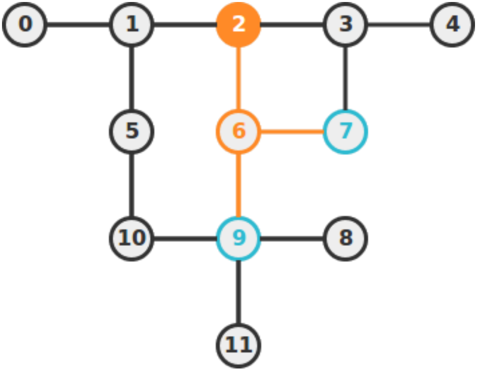


$u = 6$ $q = \{2, 7, 9\}$

$d = \{6, 2, 7, 9\}$

Breitensuche

Beispiel

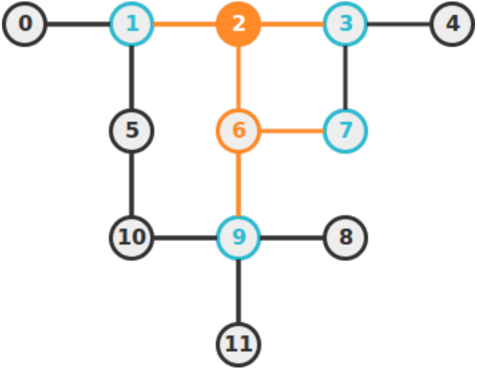


$u = 2$ $\quad\quad$ $q = \{7, 9\}$

$d = \{6, 2, 7, 9\}$

Breitensuche

Beispiel

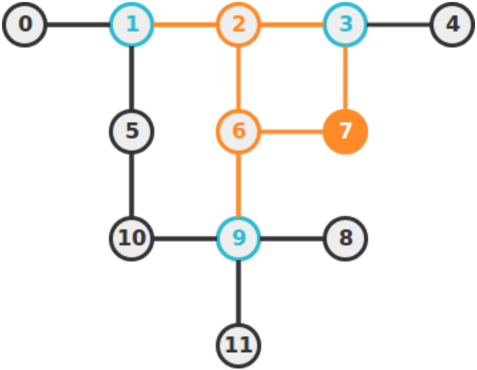


u = 2 q = {7, 9, 1, 3}

d = {6, 2, 7, 9, 1, 3}

Breitensuche

Beispiel

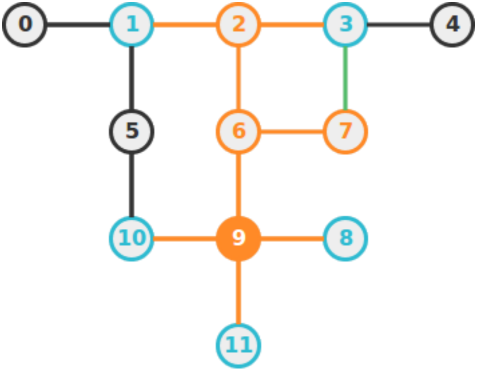


u = 7 q = {9, 1, 3}

d = {6, 2, 7, 9, 1, 3}

Breitensuche

Beispiel

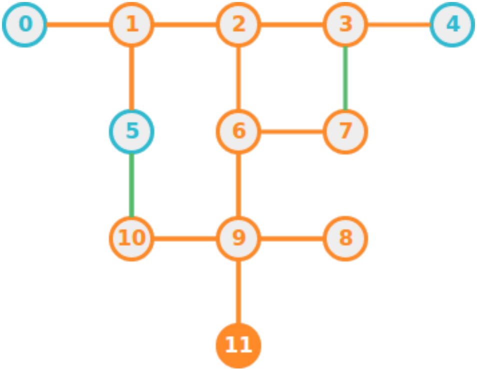


$u = 9$ $q = \{1, 3, 8, 10, 11\}$

$d = \{6, 2, 7, 9, 1, 3, 8, 10, 11\}$

Breitensuche

Beispiel

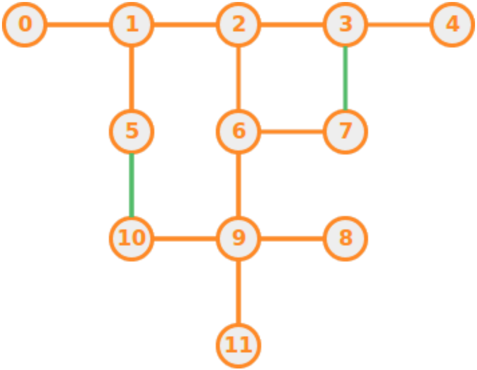


u = 11 q = {0, 5, 4}

d = {6, 2, 7, 9, 1, 3, 8, 10, 11, 0, 5, 4}

Breitensuche

Beispiel



q = {}
d = {6, 2, 7, 9, 1, 3, 8, 10, 11, 0, 5, 4}

Breitensuche

weiteres

- **Laufzeit**: $\mathcal{O}(|V| + |E|)$ bzw. $\mathcal{O}(|V|^2)$
- **Speicher**: $\mathcal{O}(|V|)$ da alle endekten Knoten gespeichert werden
- statt zu markieren, speichere das "Level" (Level Elternknoten + 1) von **u** und erhälte die Entfernung zu **s**
  $\iff$ Länge des kürzesten Pfades von **s** nach **u**

Breitensuche

Code

```
vector<int> d(V, -1);
d[s] = 0;
queue<int> q;
q.push(s);
while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        int v = AdjList[u][j];
        if (d[v] == -1) {
            d[v] = d[u] + 1;
            q.push(v);
        }
    }
}
```

Tiefensuche

Einführung

Indiana Jones and the Fate of Atlantis

Indiana Jones braucht unsere Hilfe! Er ist auf der Suche nach einer mysteriösen Statue muss er ein Labyrinth überwinden. Alles was er als Hilfsmittel besitzt ist eine (unendlich lange)rote Schnur. Wie geht Indy vor, um möglichst wenig Zeit zu verschwenden?

Anforderungen

- findet stets die Lösung, wenn sie existiert
- vermeidet doppelte Wege

Tiefensuche

Einführung

Indiana Jones and the Fate of Atlantis

Indiana Jones braucht unsere Hilfe! Er ist auf der Suche nach einer mysteriösen Statue muss er ein Labyrinth überwinden. Alles was er als Hilfsmittel besitzt ist eine (unendlich lange)rote Schnur. Wie geht Indy vor, um möglichst wenig Zeit zu verschwenden?

Anforderungen

- findet stets die Lösung, wenn sie existiert
- vermeidet doppelte Wege

Tiefensuche

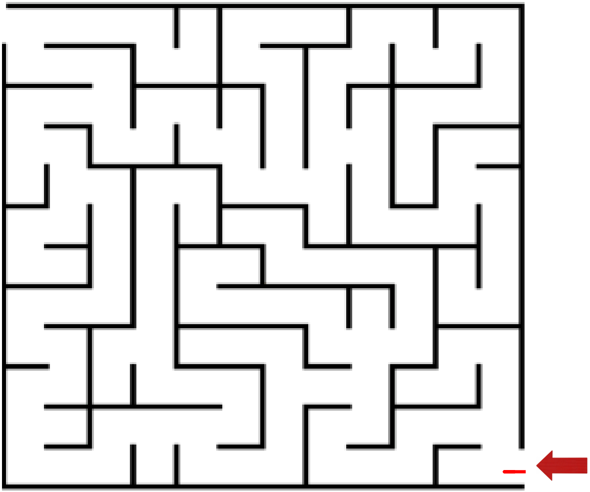
Einführung

Strategie

- Die ganze Zeit lang spannen wir unsere rote Schnur und markieren damit bereits gesehene Wege.
- Wenn wir auf eine Gabelung stoßen, gehen wir immer einen Weg, den wir noch nicht gesehen haben.
- Wenn wir auf eine Sackgasse stoßen, gehen wir zurück zu der letzten Gabelung, in der sich noch ein ungesehener Weg befindet.

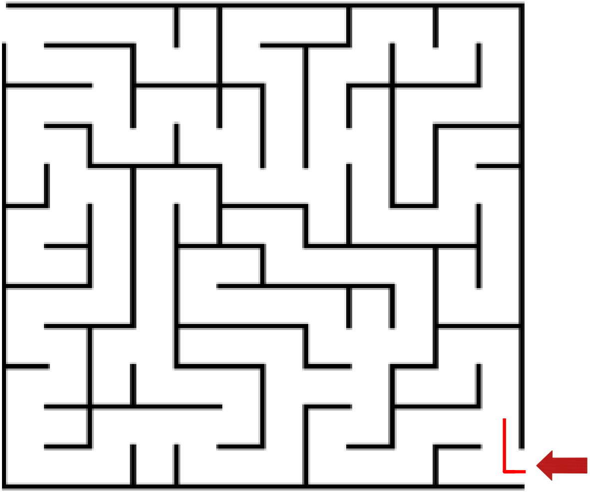
Tiefensuche

Einführung



Tiefensuche

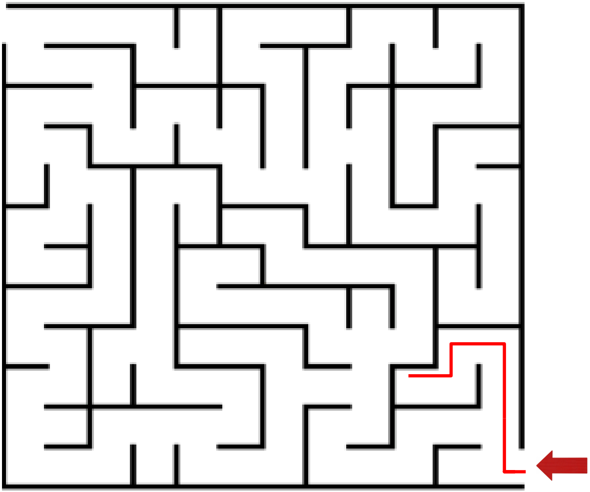
Einführung



Fakultät für Informatik
Institut für Theoretische Informatik

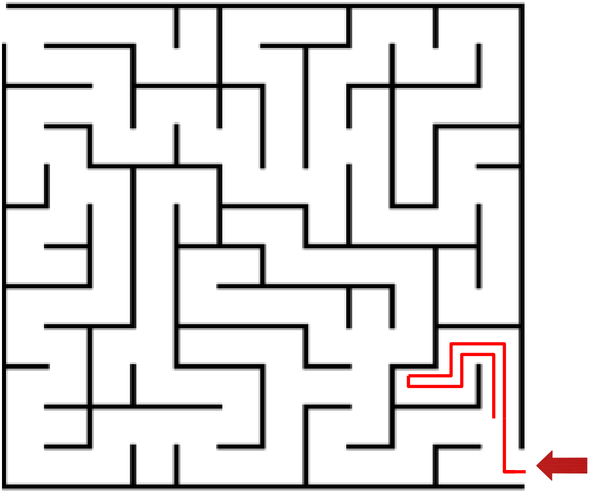
Tiefensuche

Einführung



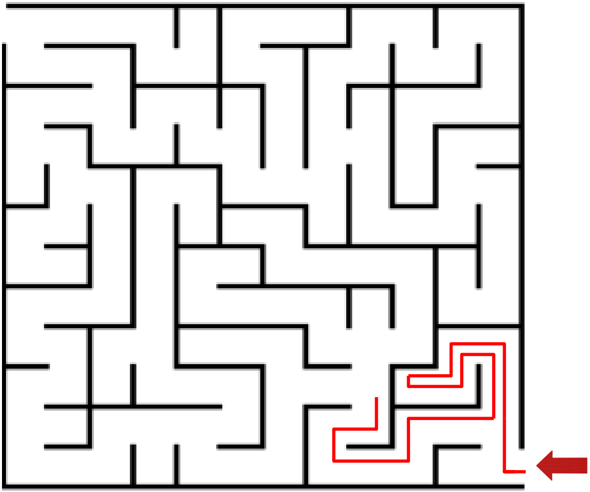
Tiefensuche

Einführung



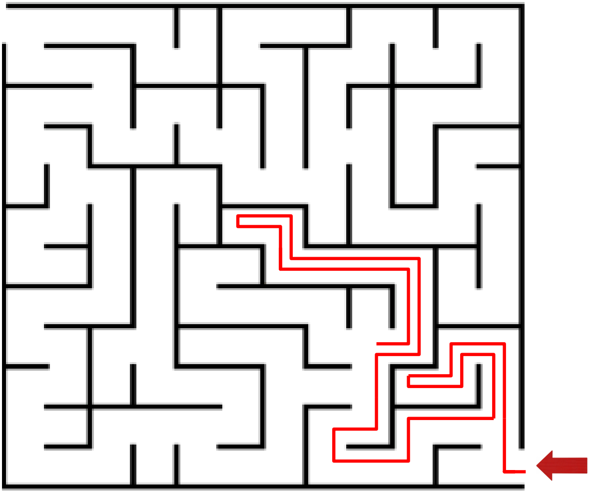
Tiefensuche

Einführung



Tiefensuche

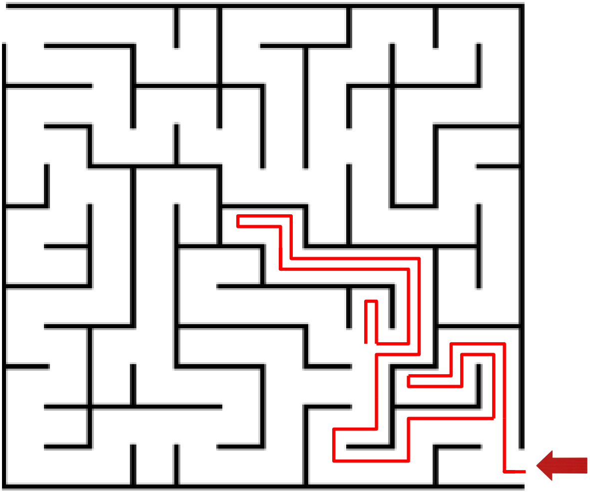
Einführung



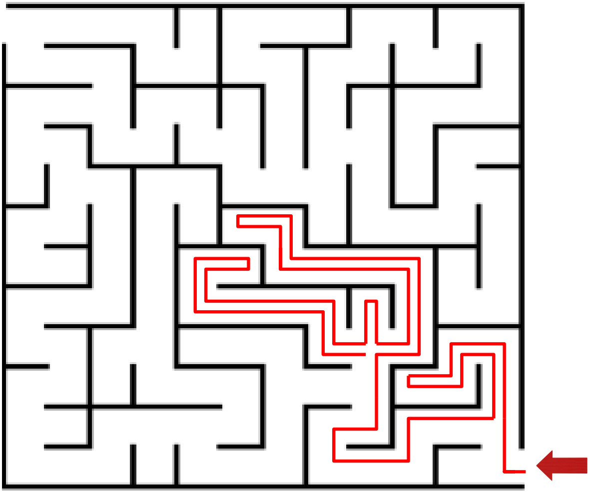
Fakultät für Informatik
Institut für Theoretische Informatik

Tiefensuche

Einführung

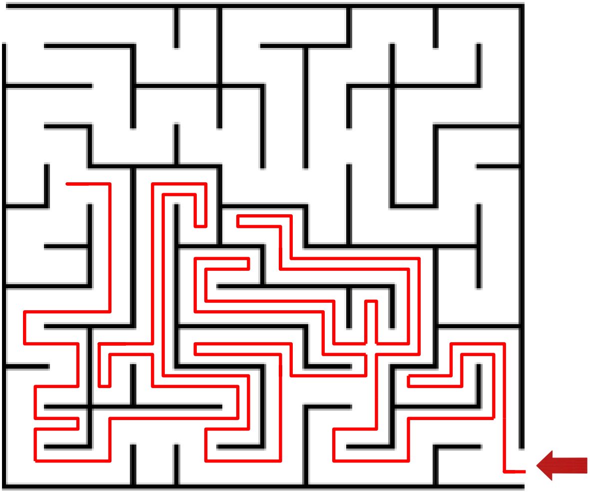


Tiefensuche
Einführung



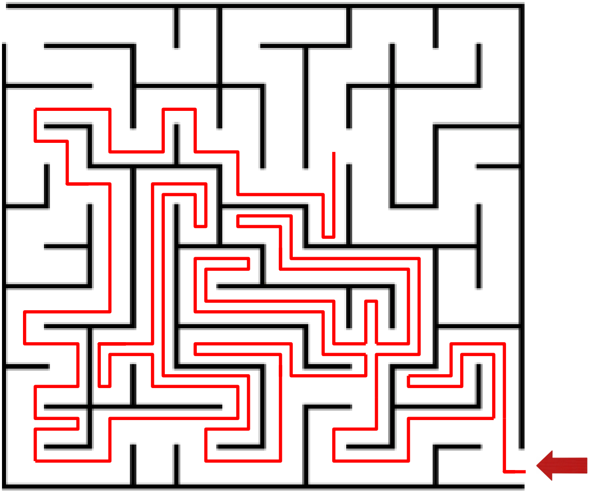
Tiefensuche

Einführung



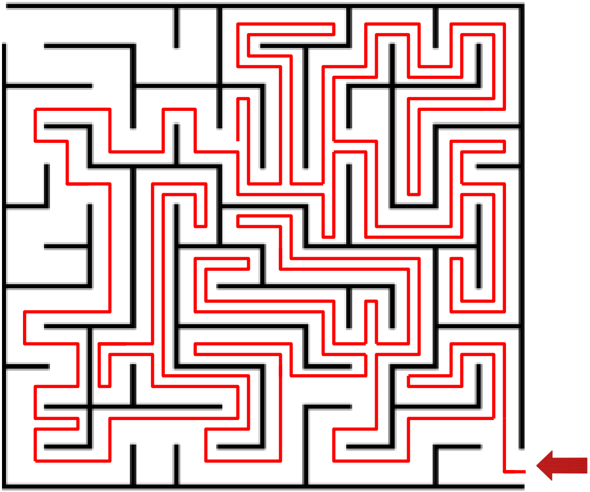
Tiefensuche

Einführung



Tiefensuche

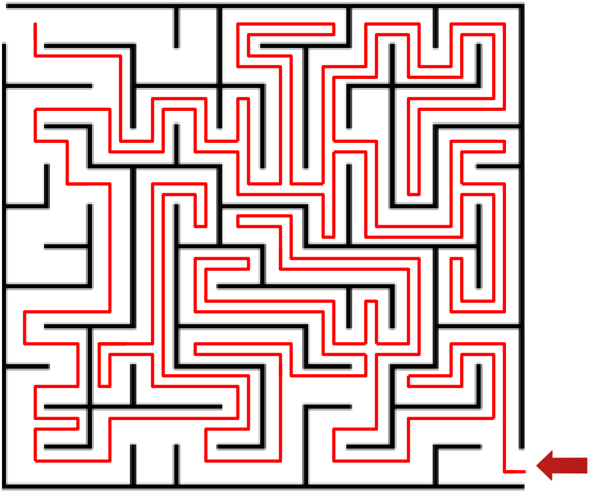
Einführung



Fakultät für Informatik
Institut für Theoretische Informatik

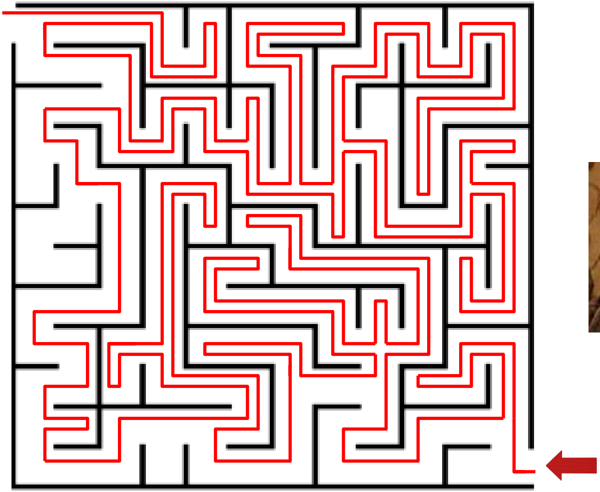
Tiefensuche

Einführung



Tiefensuche

Einführung



Tiefensuche

rekursive Implementierung

```
bool dfs_visited[AdjMatrix.size()];

void dfs(int u) {
        dfs_visited[u] = true;
        // possibly do something
        for (int i = 0; i < AdjMatrix[u].size(); i++) {
                if (AdjMatrix[u][i] == 1
                        && dfs_visited[i] == false) {
                        dfs(i);
                }
        }
}

for (int i = 0; i < AdjMatrix.size(); i++) {
        if (dfs_visited[i] == false) {
                dfs(i);
        }
}
```

Laufzeit: $\mathcal{O}(|V|^2)$, wenn der Graph als Adjazenzmatrix gespeichert ist, sonst $\mathcal{O}(|V| + |E|)$

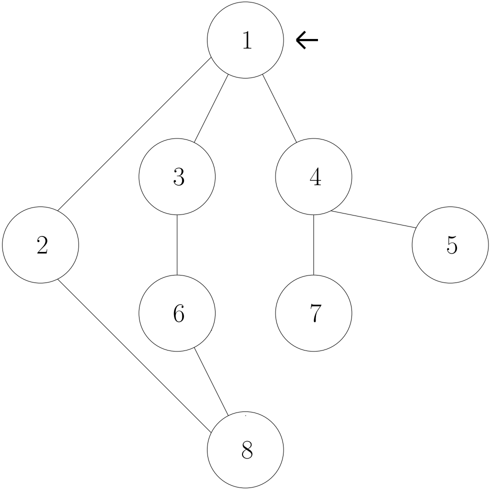
Tiefensuche

rekursive Implementierung

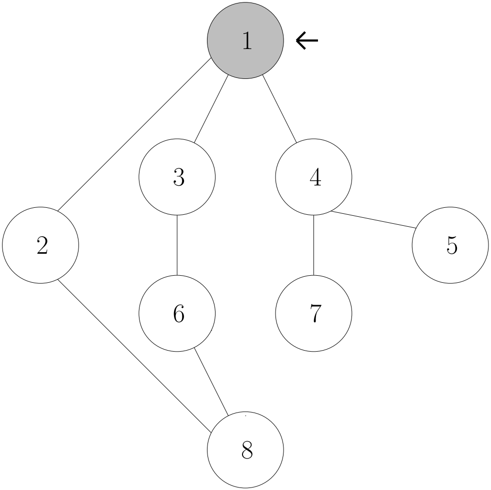
```
bool dfs_visited[AdjMatrix.size()];

void dfs(int u) {
        dfs_visited[u] = true;
        // possibly do something
        for (int i = 0; i < AdjMatrix[u].size(); i++) {
                if (AdjMatrix[u][i] == 1
                        && dfs_visited[i] == false) {
                        dfs(i);
                }
        }
}

for (int i = 0; i < AdjMatrix.size(); i++) {
        if (dfs_visited[i] == false) {
                dfs(i);
        }
}
```

Laufzeit: $\mathcal{O}(|V|^2)$, wenn der Graph als Adjazenzmatrix gespeichert ist, sonst $\mathcal{O}(|V| + |E|)$

Tiefensuche
graphische Darstellung

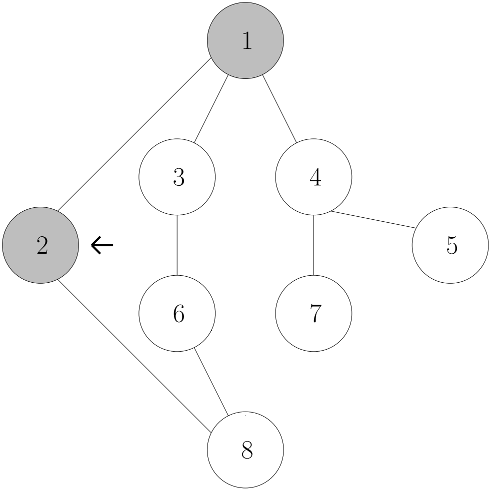


Tiefensuche
graphische Darstellung



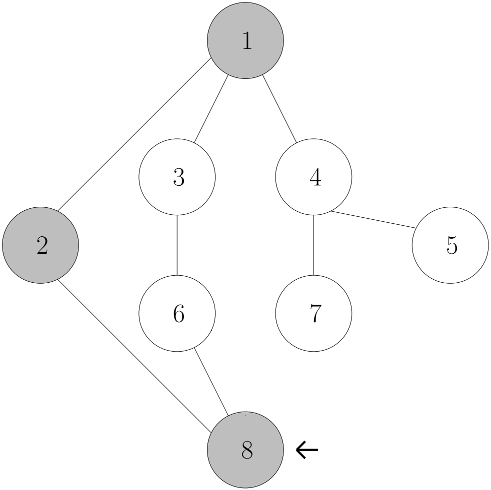
Tiefensuche

graphische Darstellung

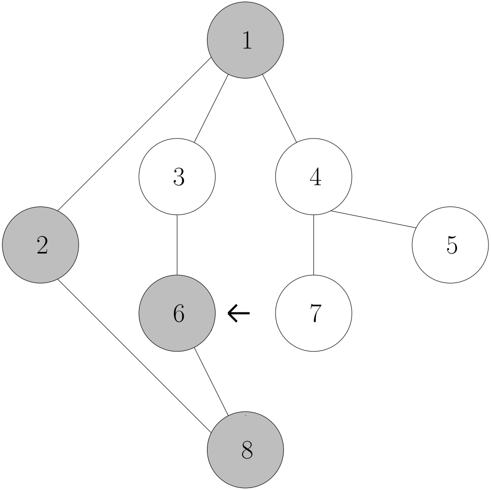


Tiefensuche

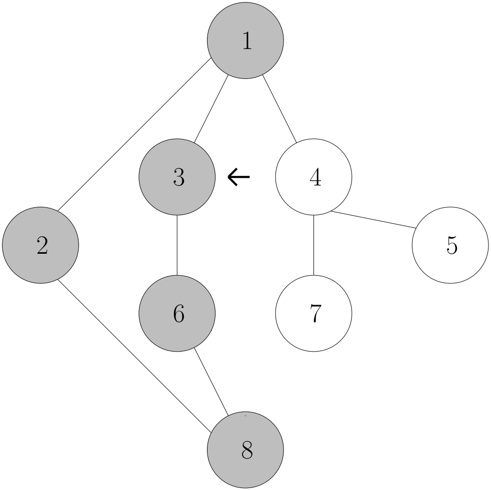
graphische Darstellung



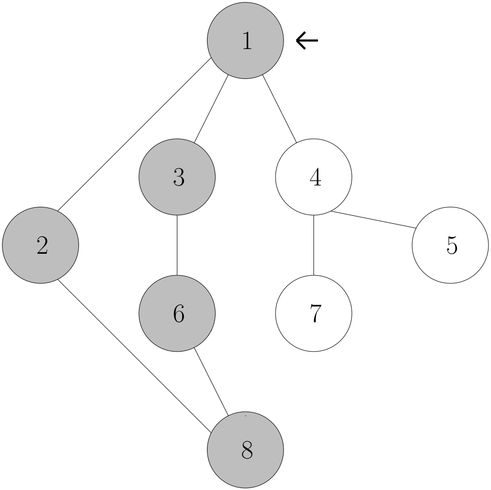
Tiefensuche
graphische Darstellung



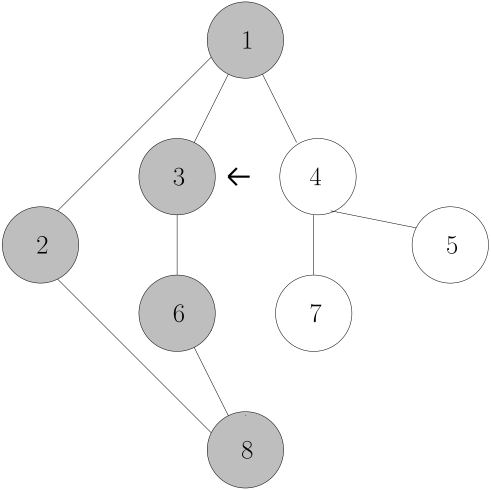
Tiefensuche
graphische Darstellung



Tiefensuche
graphische Darstellung

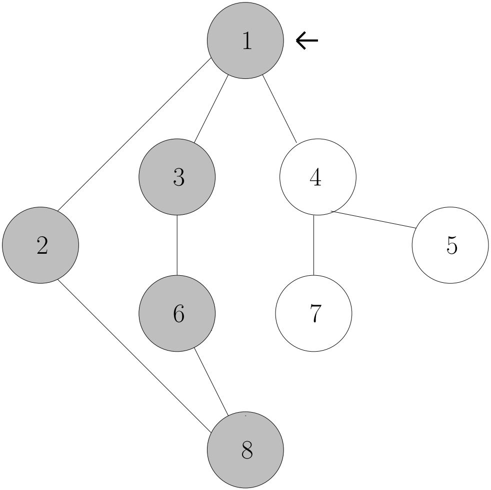


Tiefensuche
graphische Darstellung



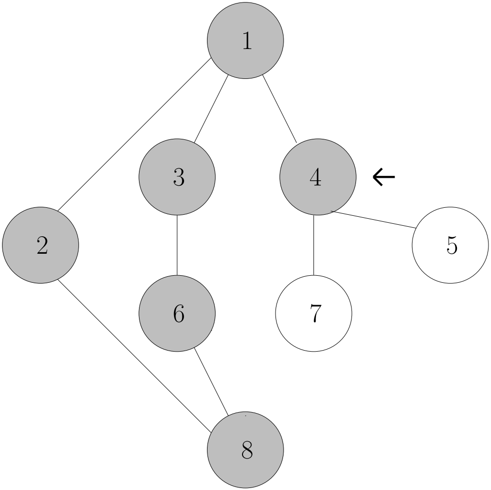
Tiefensuche

graphische Darstellung



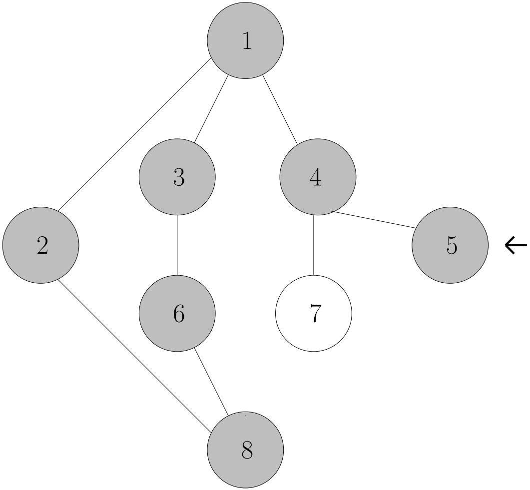
Tiefensuche

graphische Darstellung

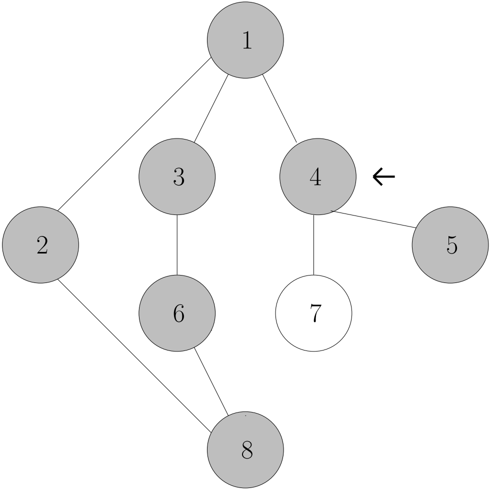


Tiefensuche

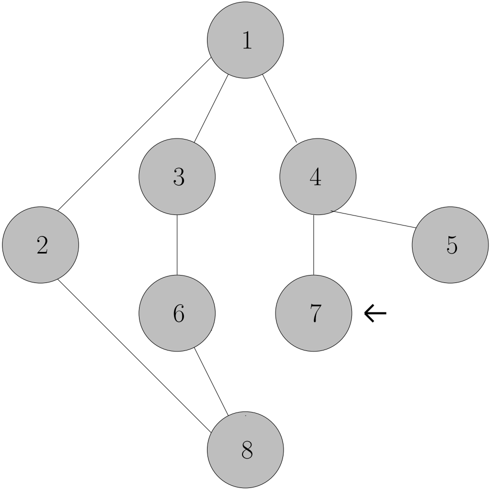
graphische Darstellung



Tiefensuche
graphische Darstellung



Tiefensuche
graphische Darstellung



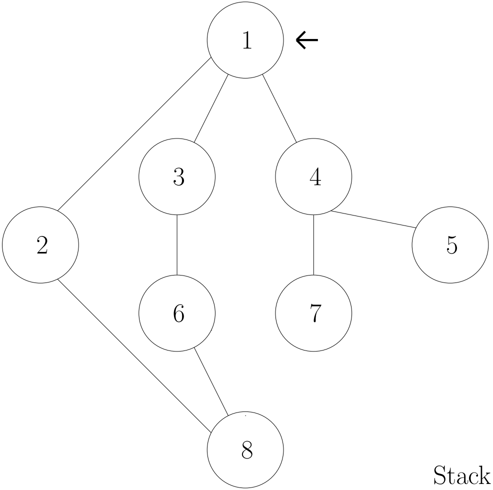
Tiefensuche

rekursive Implementierung

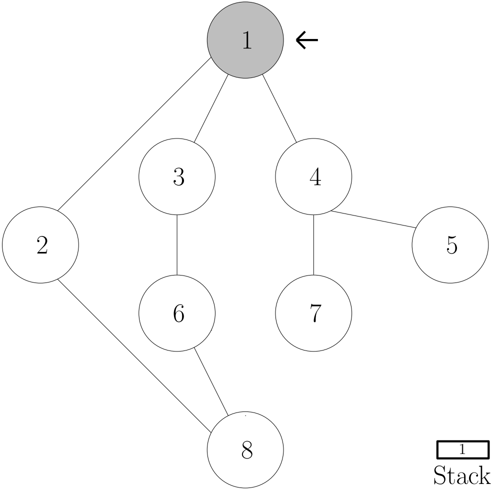
```
bool dfs_visited[AdjMatrix.size()] = {0};

nrdfs(int u) {
        stack myStack;
        myStack.push(u);
        int current;
        while(!myStack.empty()){
                current = myStack.top();
                myStack.pop();
                for(int i = 0; i < AdjMatrix.size(); i++){
                        if(AdjMatrix[current][i] == 1){
                                if(dfs_visited[i] == false){
                                        myStack.push(i);
                                        visit_table[i] = true;
                                        // possibly do something
                                }
                        }
                }
        }
}
```

Tiefensuche

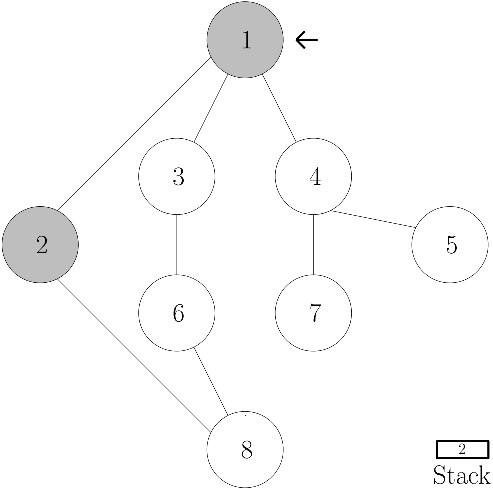
graphische Darstellung



Tiefensuche
graphische Darstellung

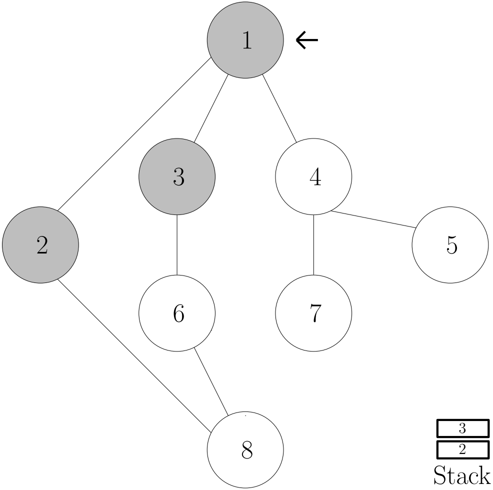


Tiefensuche
graphische Darstellung

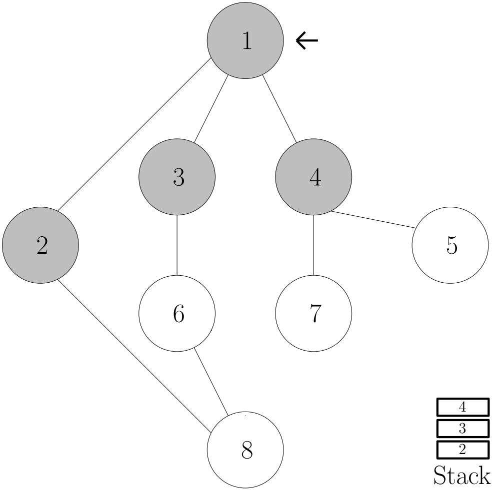


Tiefensuche

graphische Darstellung

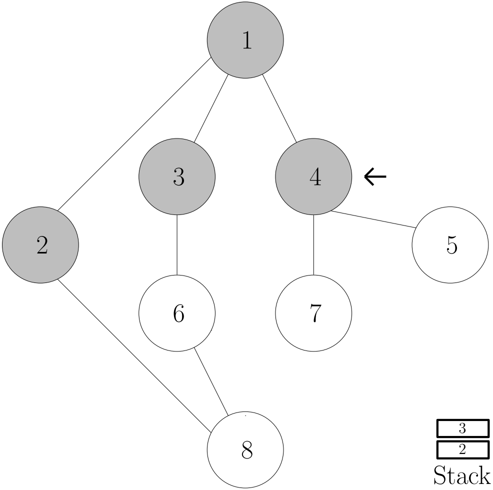


Tiefensuche
graphische Darstellung

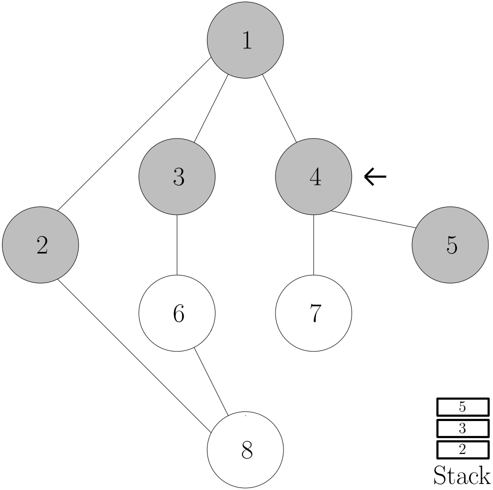


Tiefensuche

graphische Darstellung

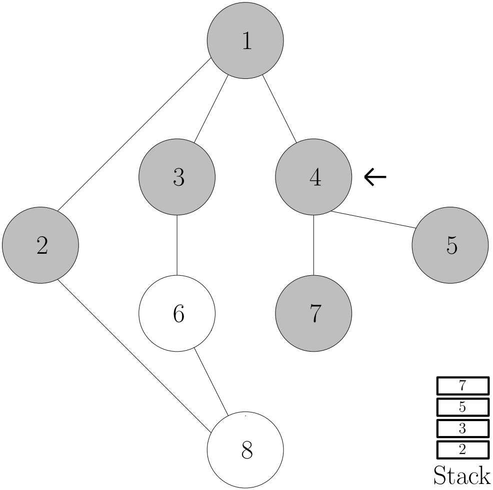


Tiefensuche
graphische Darstellung

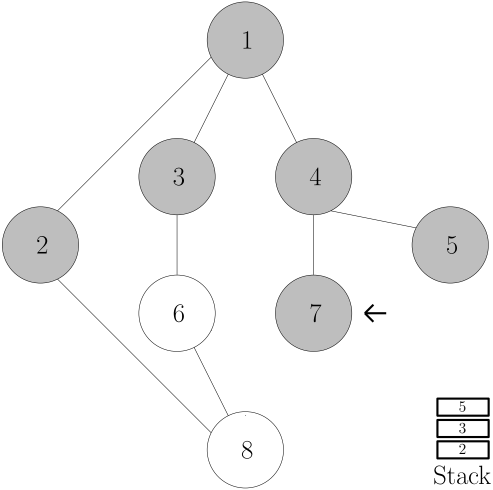


Tiefensuche

graphische Darstellung

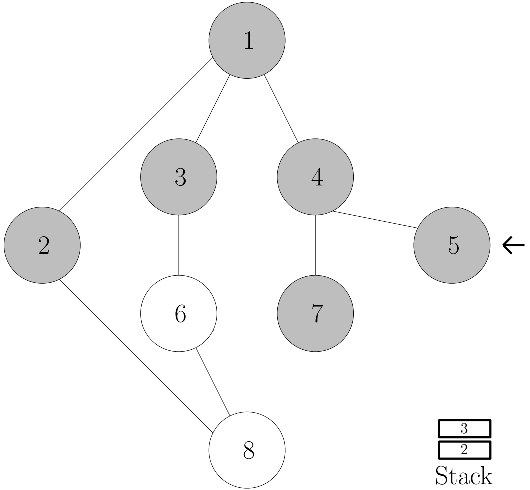


Tiefensuche
graphische Darstellung



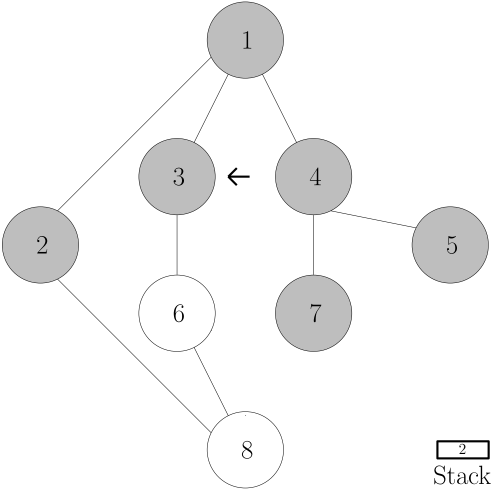
Tiefensuche

graphische Darstellung



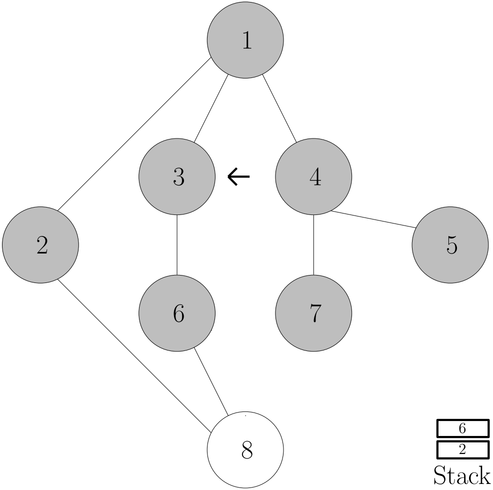
Tiefensuche

graphische Darstellung

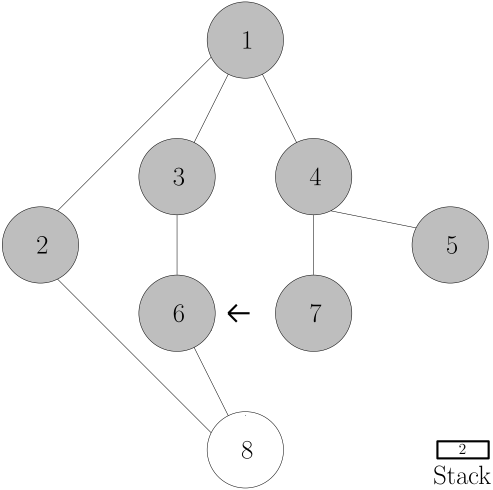


Tiefensuche

graphische Darstellung

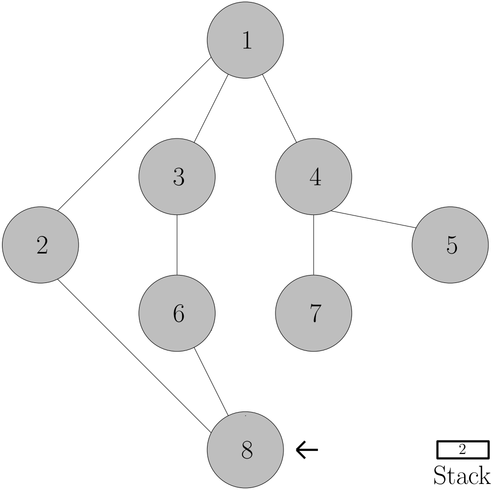


Tiefensuche
graphische Darstellung



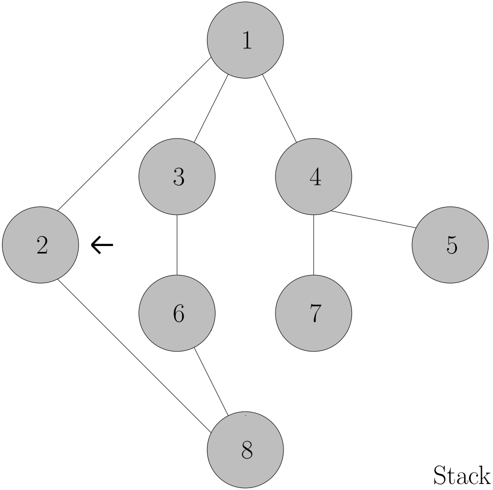
Tiefensuche

graphische Darstellung



Tiefensuche

graphische Darstellung

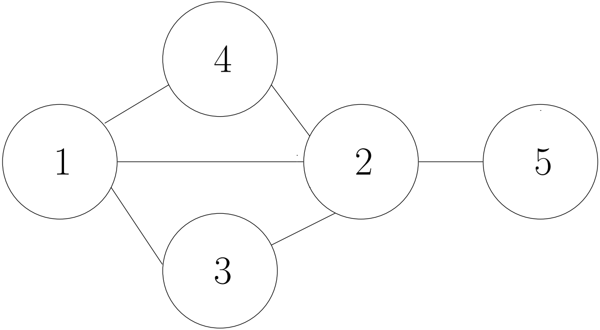


Tiefensuche

Beispielproblem

Dominator

Ein Knoten X eines Graphen dominiert einen anderen Knoten Y, wenn alle Wege von einem gegebenen Startknoten zu Y durch X gehen müssen. Wenn ein Knoten Z nicht vom Startknoten erreicht werden kann, hat Z keinen Dominator.



Tiefensuche

Beispielproblem

Dominator

Gegeben sei ein Graph. Die Aufgabe ist es, für einen gegebenen Graphen für jeden Knoten die Dominator auszurechnen. Dabei ist zu erwähnen, dass der Eingabegraph sehr klein sind, mit weniger als 100 Knoten.

Fakultät für Informatik
Institut für Theoretische Informatik

Tiefensuche

Beispielproblem

Idee

- Lassen zunächst Tiefensuche mit dem Anfangsknoten als Startknoten laufen und speichern uns alle Knoten ein, die erreicht worden sind.

- Um zu prüfen, welche Knoten von einem Knoten X dominiert werden, löschen (oder blenden aus) wir temporär den Knoten X und laufen mit Tiefensuche durch den Graph.

- Alle Knoten, die nun nicht mehr erreicht werden können, werden von X dominiert.

- Laufzeit ist $\mathcal{O}(|V|^3)$ im worst case.

Tiefensuche

Beispielproblem

Idee

- Lassen zunächst Tiefensuche mit dem Anfangsknoten als Startknoten laufen und speichern uns alle Knoten ein, die erreicht worden sind.
- Um zu prüfen, welche Knoten von einem Knoten X dominiert werden, löschen (oder blenden aus) wir temporär den Knoten X und laufen mit Tiefensuche durch den Graph.
- Alle Knoten, die nun nicht mehr erreicht werden können, werden von X dominiert.
- Laufzeit ist $\mathcal{O}(|V|^3)$ im worst case.

Tiefensuche

Beispielproblem

Idee

- Lassen zunächst Tiefensuche mit dem Anfangsknoten als Startknoten laufen und speichern uns alle Knoten ein, die erreicht worden sind.
- Um zu prüfen, welche Knoten von einem Knoten X dominiert werden, löschen (oder blenden aus) wir temporär den Knoten X und laufen mit Tiefensuche durch den Graph.
- Alle Knoten, die nun nicht mehr erreicht werden können, werden von X dominiert.
- Laufzeit ist $\mathcal{O}(|V|^3)$ im worst case.

Tiefensuche

Beispielprobleme

Karlsruher Institut für Technologie

Idee

- Lassen zunächst Tiefensuche mit dem Anfangsknoten als Startknoten laufen und speichern uns alle Knoten ein, die erreicht worden sind.
- Um zu prüfen, welche Knoten von einem Knoten X dominiert werden, löschen (oder blenden aus) wir temporär den Knoten X und laufen mit Tiefensuche durch den Graph.
- Alle Knoten, die nun nicht mehr erreicht werden können, werden von X dominiert.
- Laufzeit ist $\mathcal{O}(|V|^3)$ im worst case.

Fakultät für Informatik
Institut für Theoretische Informatik

Tiefensuche

Allgemeines

- Der Speicherplatz für den Algorithmus ist linear, da Informationen darüber gespeichert werden, ob ein Knoten schon besucht wurde.
- Die Laufzeit ist, wie bereits erwähnt $\mathcal{O}(|V|^2)$, wenn der Graph als Adjazenzmatrix gespeichert ist, sonst $\mathcal{O}(|V| + |E|)$.
- Die Tiefensuche ist eher ungeeignet für Graphen mit monoton steigenden Pfadkosten. Dafür eher Breitensuche oder iterative Tiefensuche.

Bipartite Graphen

Bipartite Graphen
Definition

Separatoren und Brücken in ungerichteten Graphen

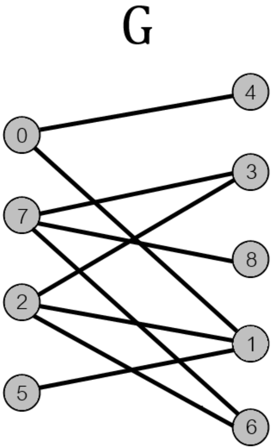
Sei $G = (V, E)$ ein ungerichteter Graph.

- 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 $v \in A$.

Fakultät für Informatik
Institut für Theoretische Informatik

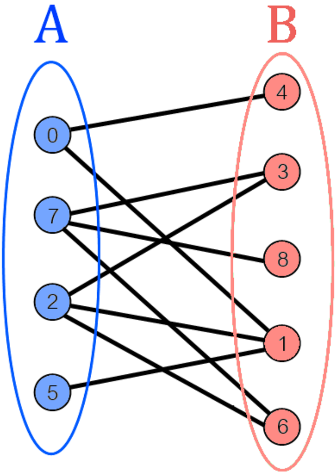
Bipartite Graphen

Beispiel



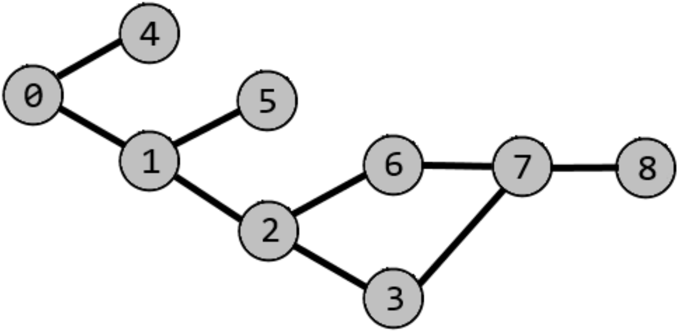
Bipartite Graphen

Beispiel



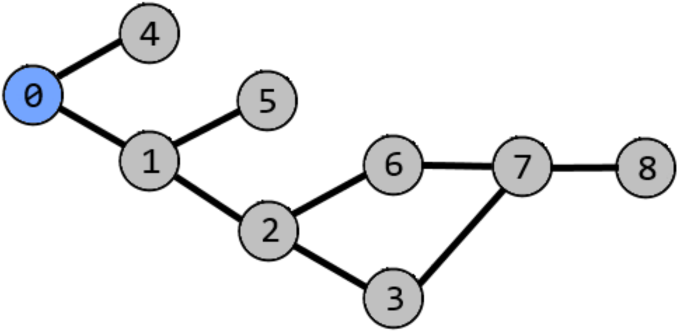
Bipartite Graph Check

DFS Algorithmus



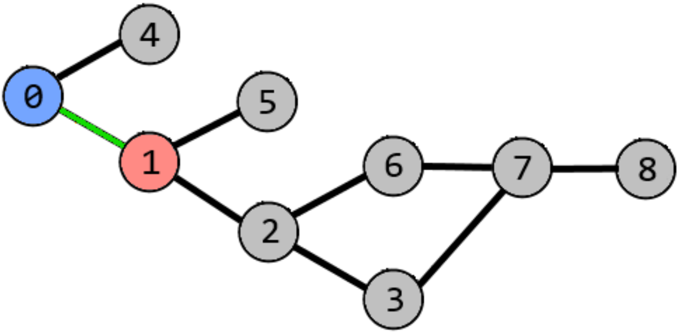
Bipartite Graph Check

DFS Algorithmus



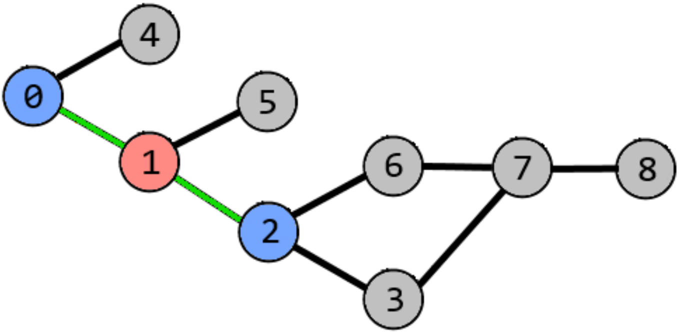
Bipartite Graph Check

DFS Algorithmus



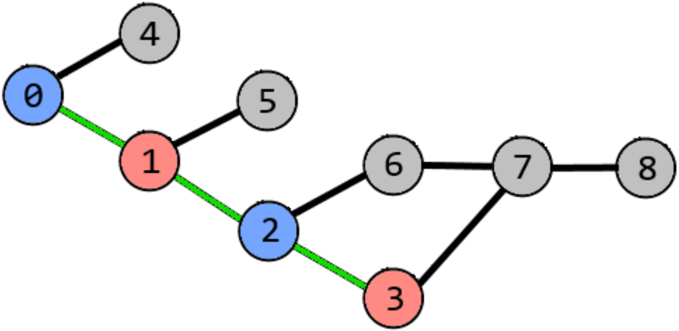
Bipartite Graph Check

DFS Algorithmus



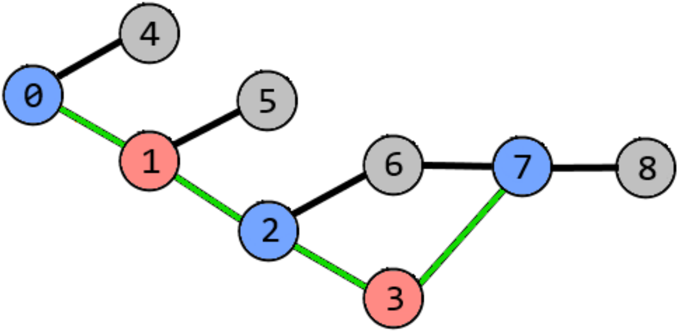
Bipartite Graph Check

DFS Algorithmus



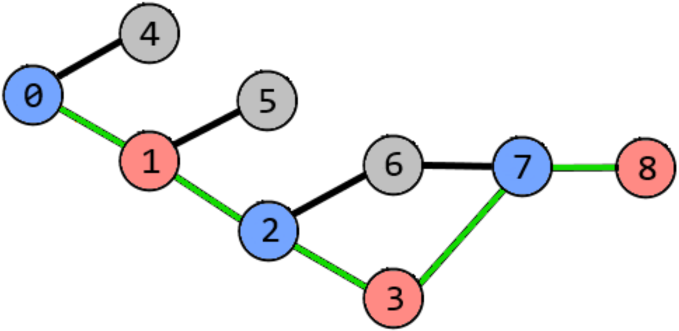
Bipartite Graph Check

DFS Algorithmus



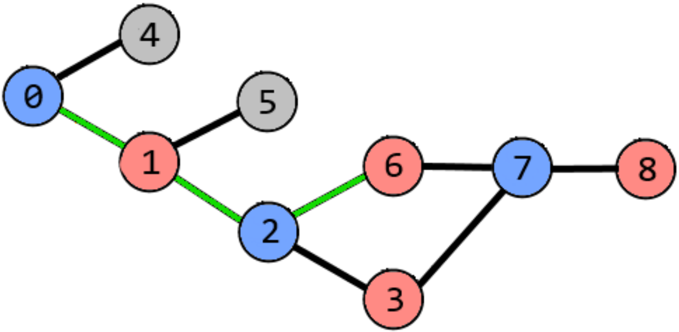
Bipartite Graph Check

DFS Algorithmus



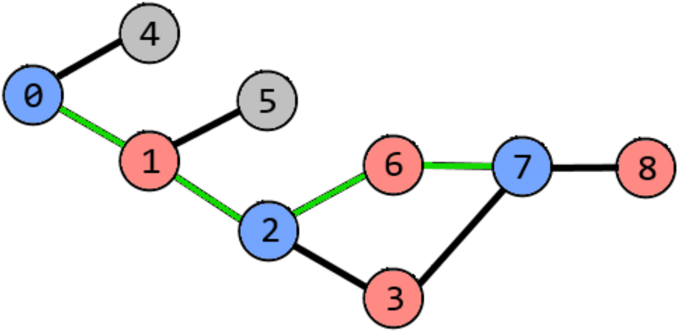
Bipartite Graph Check

DFS Algorithmus



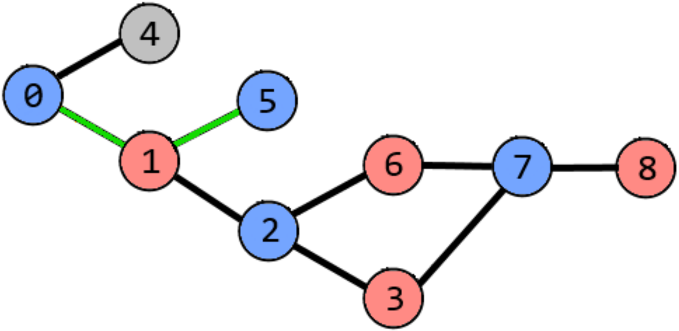
Bipartite Graph Check

DFS Algorithmus



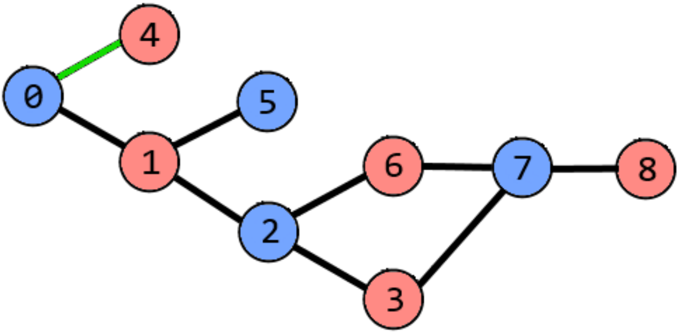
Bipartite Graph Check

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;
}
```

Iterative Tiefensuche

Iterative Tiefensuche

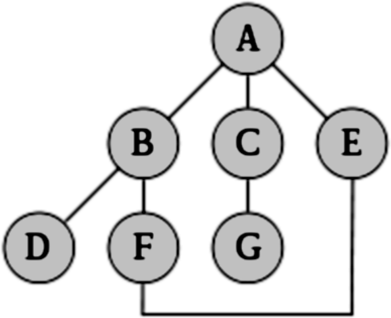
Idee des Algorithmus

- 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,

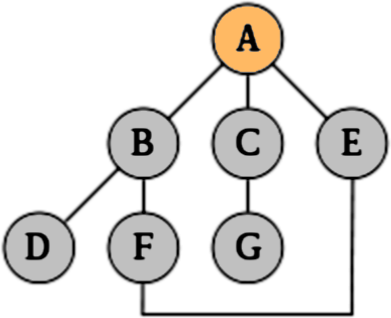
  und d die Tiefe des flachsten Ziels.
- Laufzeit: $\mathcal{O}(b^d)$

Iterative Tiefensuche



Anfang: A

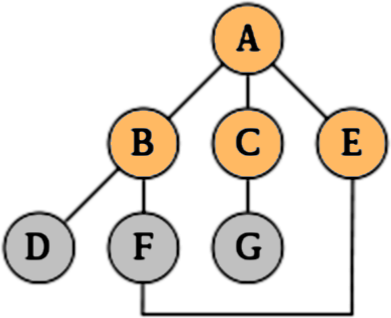
Iterative Tiefensuche



Anfang: A

d = 0: A

Iterative Tiefensuche

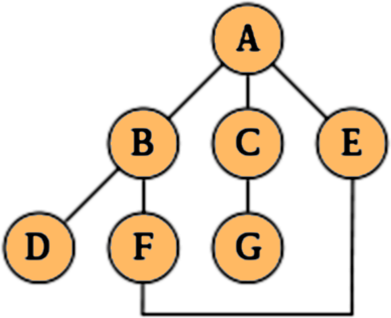


Anfang: A

d = 0: A

d = 1: ABCE

Iterative Tiefensuche



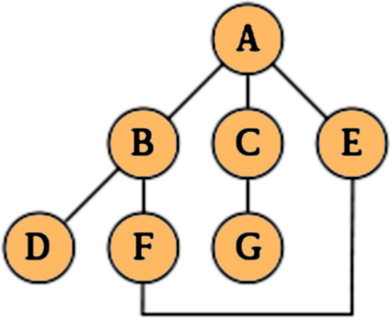
Anfang: A

d = 0: A

d = 1: ABCE

d = 2: ABDFCGEF

Iterative Tiefensuche



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;
}
```

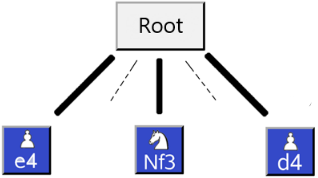
Iterative Tiefensuche

Schach

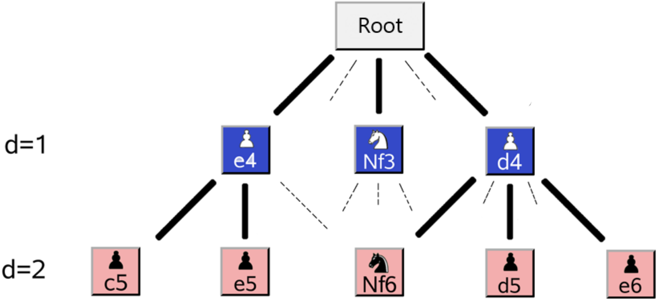
- Besonders sinnvoll für Probleme, die hoch branching factor haben, z.B.: **Schach**
- Im Durchschnitt hat ein Schachspieler für jeder Position 35~38 mögliche Bewegungen
- Iterative Tiefensuche findet die beste Bewegung bis zum eine Tiefe T, innerhalb ein Zeitlimit

Iterative Tiefensuche

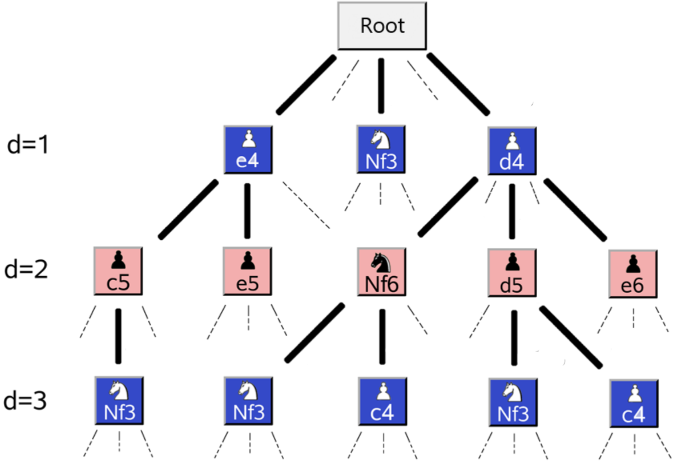
d=1



Iterative Tiefensuche



Iterative Tiefensuche



Starke Zusammenhangskomponenten

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.

SCCs finden mittels DFS

- 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

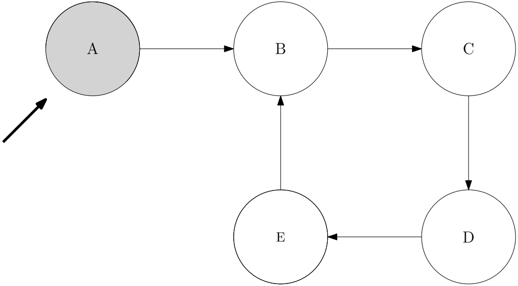
- Zum Lösen der Aufgabe untersuchen, ob das Straßennetz der Stadt aus einer oder mehreren SCCs besteht.
- $\Rightarrow$ Benötigen effizienten Algorithmus zum Finden von SCCs

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

- dfs num(A) = 0
- dfs low(A) = 0



STACK:

- A

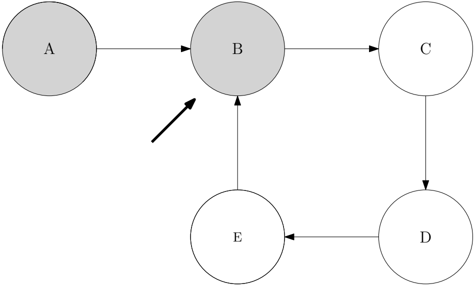
SCCs finden mittels DFS

- dfs num(A) = 0
- dfs low(A) = 0

- dfs num(B) = 1
- dfs low(B) = 1

STACK:

- A
- B

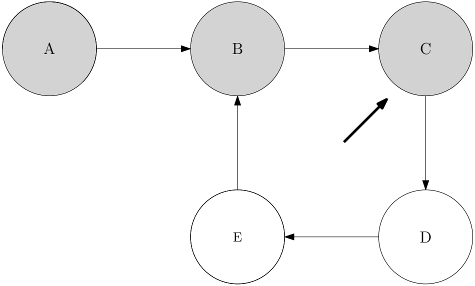


SCCs finden mittels DFS

- dfs num(A) = 0
- dfs low(A) = 0

- dfs num(B) = 1
- dfs low(B) = 1

- dfs num(C) = 2
- dfs low(C) = 2



STACK:

- A
- B
- C

SCCs finden mittels DFS

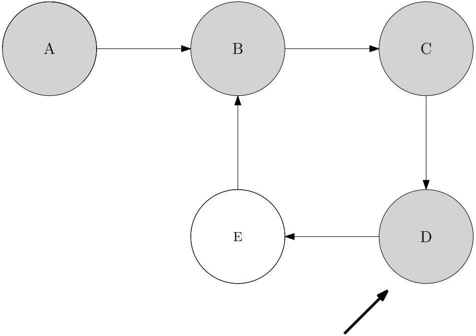
- dfs num(A) = 0
- dfs low(A) = 0

- dfs num(B) = 1
- dfs low(B) = 1

- dfs num(C) = 2
- dfs low(C) = 2

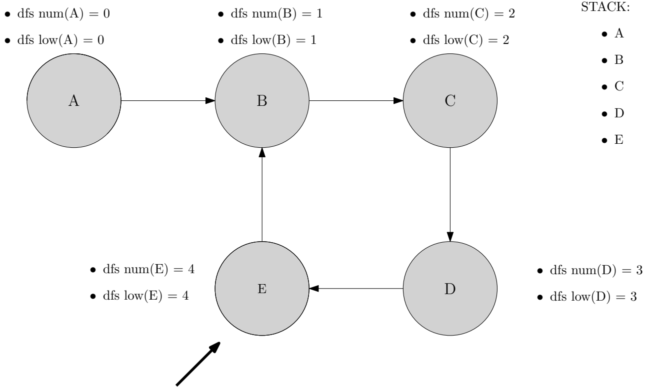
STACK:

- A
- B
- C
- D



- dfs num(D) = 3
- dfs low(D) = 3

SCCs finden mittels DFS



STACK:

- A
- B
- C
- D
- E

SCCs finden mittels DFS

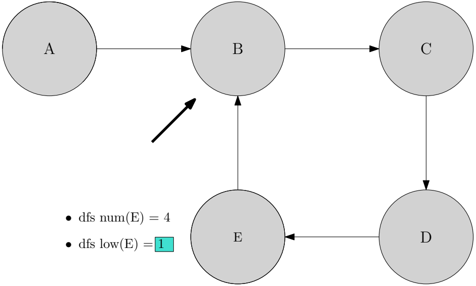
- dfs num(A) = 0
- dfs low(A) = 0

- dfs num(B) = 1
- dfs low(B) = 1

- dfs num(C) = 2
- dfs low(C) = 2

STACK:

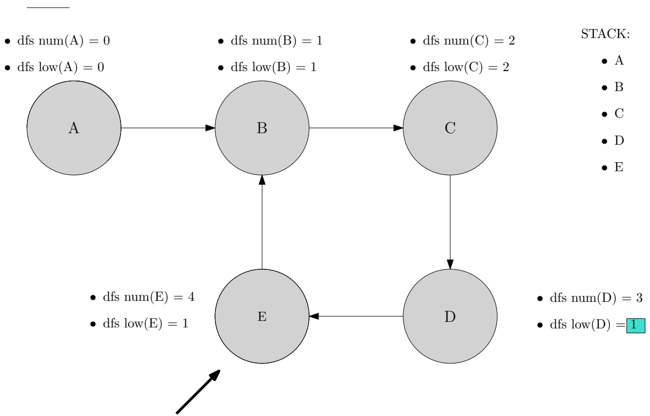
- A
- B
- C
- D
- E



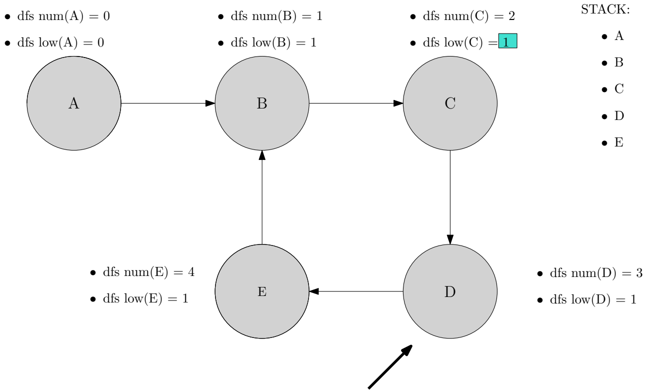
- dfs num(E) = 4
- dfs low(E) = $\boxed{1}$

- dfs num(D) = 3
- dfs low(D) = 3

SCCs finden mittels DFS



SCCs finden mittels DFS



SCCs finden mittels DFS

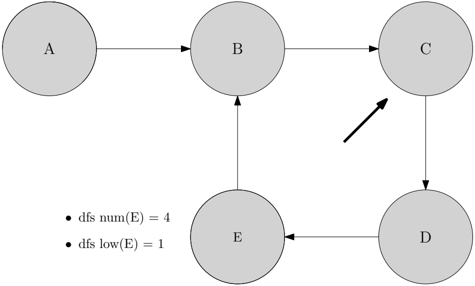
- dfs num(A) = 0
- dfs low(A) = 0

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

- dfs num(C) = 2
- dfs low(C) = 1

STACK:

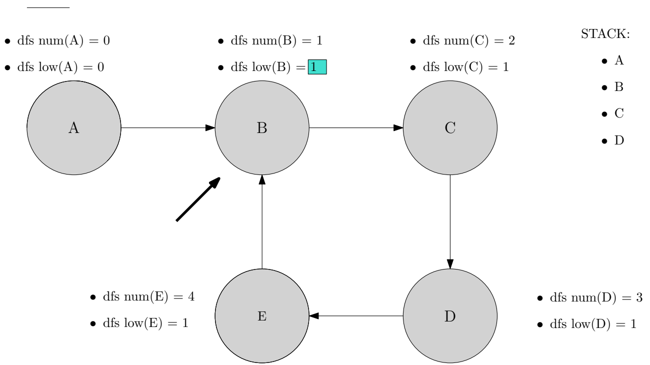
- A
- B
- C
- D
- E



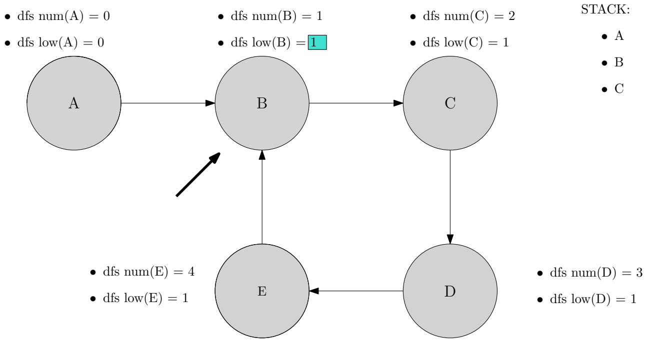
- dfs num(E) = 4
- dfs low(E) = 1

- dfs num(D) = 3
- dfs low(D) = 1

SCCs finden mittels DFS



SCCs finden mittels DFS



SCCs finden mittels DFS

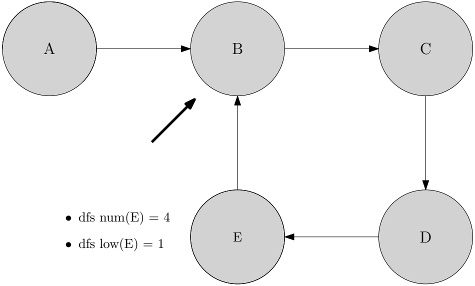
- dfs num(A) = 0
- dfs low(A) = 0

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

- dfs num(C) = 2
- dfs low(C) = 1

STACK:

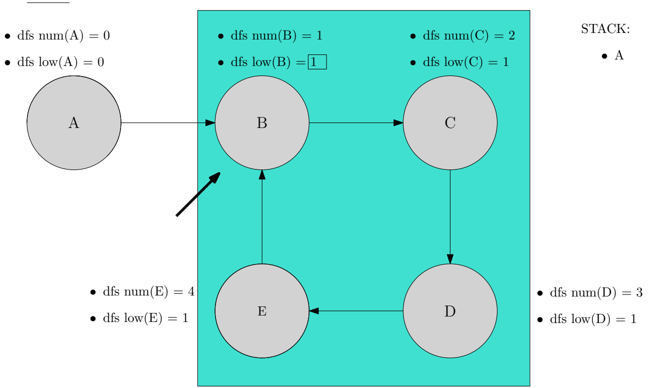
- A
- B



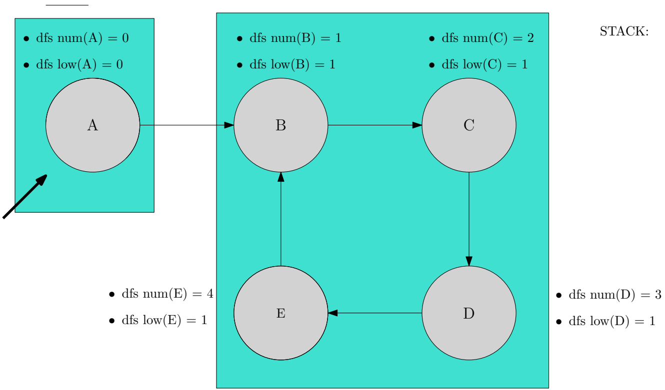
- dfs num(E) = 4
- dfs low(E) = 1

- dfs num(D) = 3
- dfs low(D) = 1

SCCs finden mittels DFS



SCCs finden mittels DFS



SCCs finden mittels DFS

Sourcecode

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

        for (int j = 0; j < AdjacenceList[u].size(); j++) {
                ii v = AdjacenceList[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 Konten **u** aus erreichbar sind.
- Für vollständige Liste an SCCs **findSCC(int u)** für alle Knoten eines Graphen laufen lassen.

Brücken und Separatoren

Fakultät für Informatik
Institut für Theoretische Informatik

Brücken und Separatoren

Definition

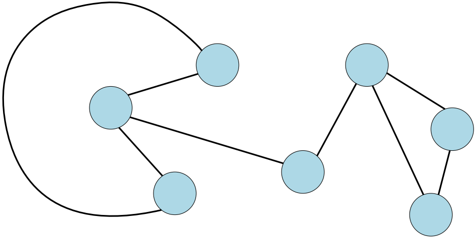
Separatoren und Brücken in ungerichteten Graphen

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.

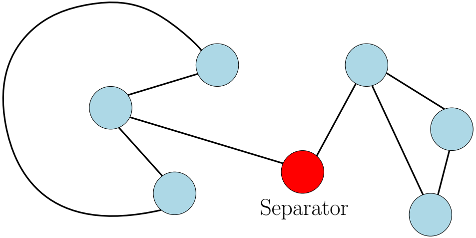
Brücken und Separatoren

Beispiel



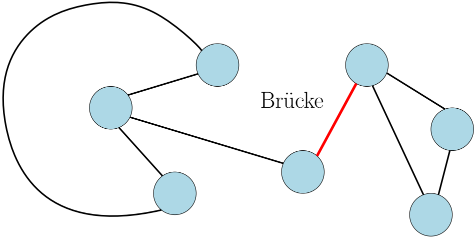
Brücken und Separatoren

Beispiel



Brücken und Separatoren

Beispiel



Brücken und Separatoren

Algorithmen

- 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: $\mathcal{O}(|V| \cdot (|V| + |E|))$ bwz. $\mathcal{O}(|E| \cdot (|V| + |E|))$
- Es existiert Algorithmus in $\mathcal{O}(|V| + |E|)$
- Basiert auf DFS und ähnelt Algorithmus zum Finden von SCCs

Brücken und Separatoren

Idee des effizienten Algorithmus'

- 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)

Brücken und Separatoren

Idee des effizienten Algorithmus'

- 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)

Brücken und Separatoren

Idee des effizienten Algorithmus'

- 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)

Brücken und Separatoren

Idee des effizienten Algorithmus'

- 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)

Brücken und Separatoren

Idee des effizienten Algorithmus'

- 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
Idee des effizienten Algorithmus'

- 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.

Fakultät für Informatik
Institut für Theoretische Informatik

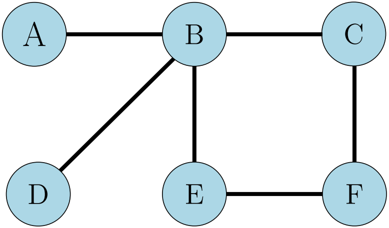
Brücken und Separatoren

Illustration des Algorithmus’

- dfs num(B) = 1
- dfs low(B) = 1

- dfs num(A) = 0
- dfs low(A) = 0

- dfs num(C) = 2
- dfs low(C) = 2



- dfs num(D) = 3
- dfs low(D) = 3

- dfs num(F) = 5
- dfs low(F) = 1

- dfs num(E) = 4
- dfs low(E) = 1