Karlsruher Institut für Technologie

Graphen I

Gustavo Crivelli, Beini Ma, Matthias Schimek, Matthias Schmitt

ICPC-Praktikum 2015 - Graphen I

Gliederung

Einleitung

Einleitung

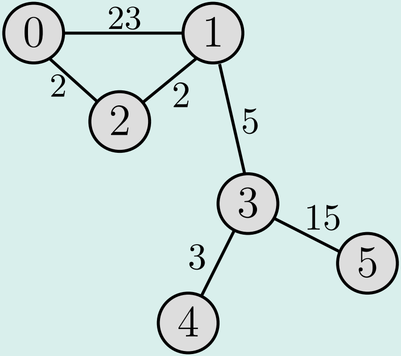
Was ist ein Graph

Graph

- Ein Graph G ist ein geordnetes Paar $G = (V, E)$
- V Menge von Knoten/Vertices
- E Menge von Kanten/Edges
  - gerichteter Graph (ohne Mehrfachkanten) $E \subseteq V \times V$
  - ungerichteter Graph (ohne Mehrfachkanten)
    $E \subseteq V \times V \quad \wedge \quad (u, w) \in E \iff (w, u) \in E$

Einleitung
Beispiel

Ungerichteter Beispielgraph



Einleitung
Implementierungen

Adjazenzmatrix

int[][]

- Speicherverbrauch $\mathcal{O}(|V|^2)$
- über Nachbarn iterieren $\mathcal{O}(|V|)$
- ICPC fast nur bei Floyd-Warschall

Visualisierung

$$\begin{array}{c|c} & \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} 0 & 23 & 2 & 0 & 0 & 0 \\ 23 & 0 & 2 & 0 & 0 & 0 \\ 2 & 2 & 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 & 3 & 15 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 15 & 0 & 0 \end{pmatrix} \end{array}$$

Einleitung

Implementierungen

Adjazenzlist

vector<vector<pair<int,int>>>

- Speicherverbrauch $\mathcal{O}(|V| + |E|)$
- kompakter und effizienter, da $|E| \ll \frac{|V|(|V|-1)}{2} \in \mathcal{O}(|V|^2)$
- über alle k Nachbarn iterieren $\mathcal{O}(k)$
- ICPC Standardwahl

Visualisierung

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

Einleitung

Implementierungen

Kantenlist

vector<tuple<int,int,int>>

- Speicherverbrauch $\mathcal{O}(|E|)$
- sortiert (nach Gewicht) wichtige Representation
- ICPC bei Greedyalgorithmen (z.B. Kruskal)

Visualisierung

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

Karlsruher Institut für Technologie

Breitensuche breadth-first search, BFS

Breitensuche

Breitensuche

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

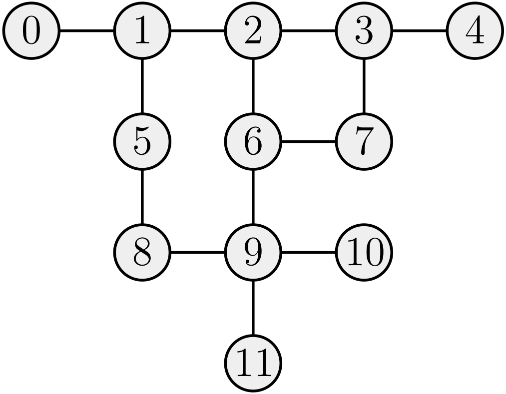
Idee: Besuche den Startknoten, dann dessen Nachbarn, dann deren Nachbarn, usw...

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

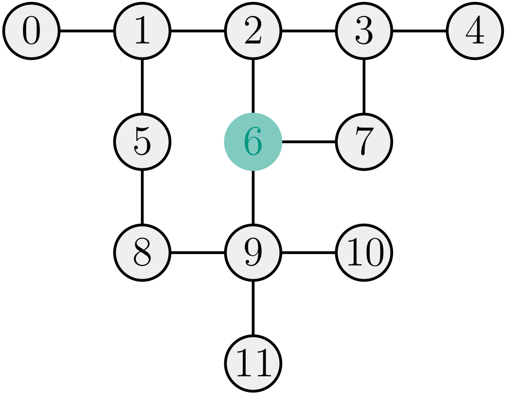
Beispiel



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

Breitensuche

Beispiel

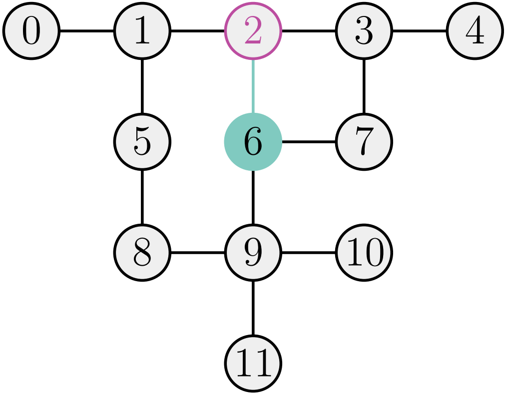


u = 6 q = {}

d = {6}

Breitensuche

Beispiel

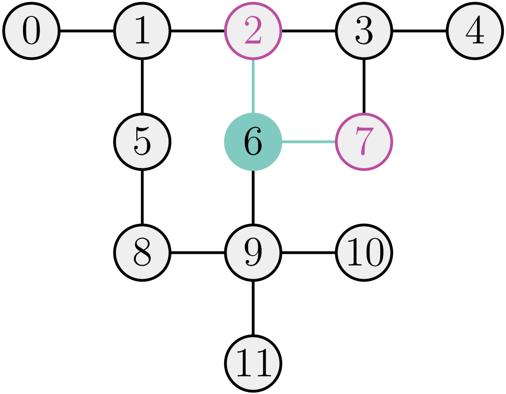


u = 6 q = {2}

d = {6, 2}

Breitensuche

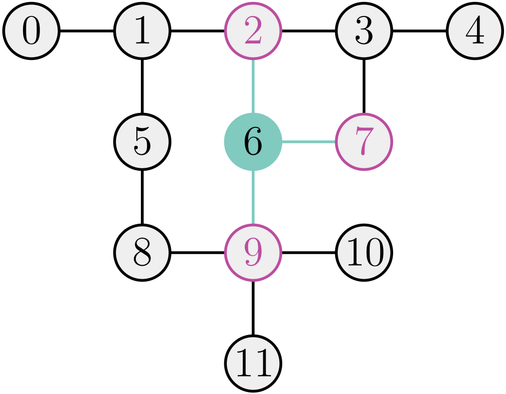
Beispiel



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

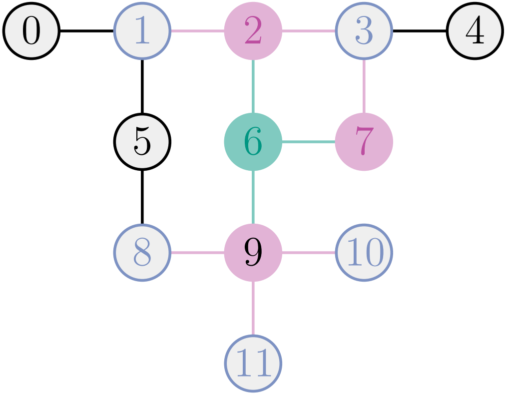
Breitensuche

Beispiel



u = 6 q = {2, 7, 9}
d = {6, 2, 7, 9}

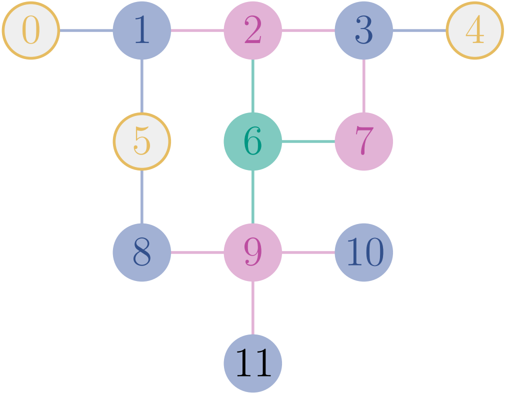
Breitensuche
Beispiel



u = 9 q = {1, 3, 8, 10, 11}
d = {6, 2, 7, 9, 1, 3, 8, 10, 11}

Breitensuche

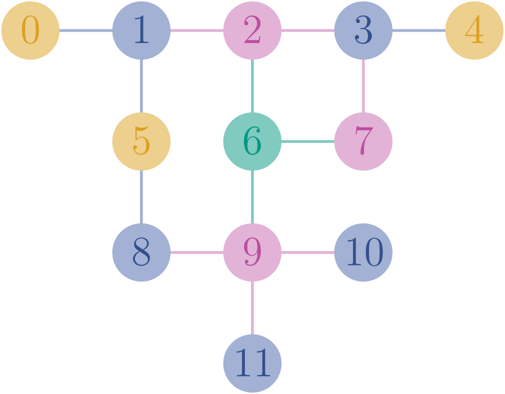
Beispiel



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**: $\mathcal{O}(|V| + |E|)$
- **Speicher**: $\mathcal{O}(|V|)$ da alle endeckten Knoten gespeichert werden
- statt zu markieren, speichere das "Level" von **u** und erhalte die "Kantenentfernung" zu **s**
  $\iff$ Länge des kürzesten ungewichteten Pfads
  (Anzahl der Hops 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 depth-first search, BFS

Fakultät für Informatik
Institut für Theoretische Informatik

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

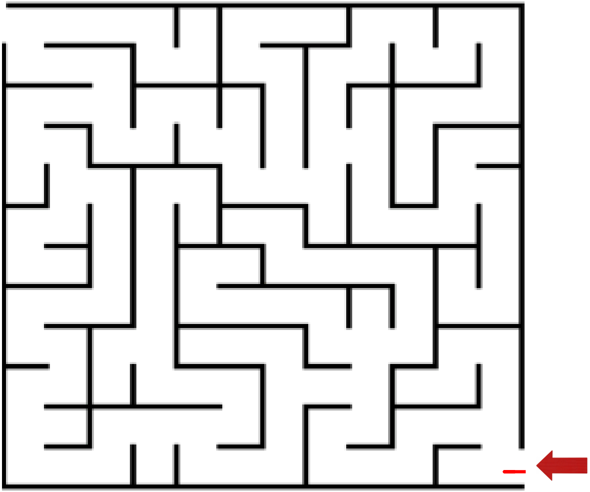
Tiefensuche

Einführung

Strategie

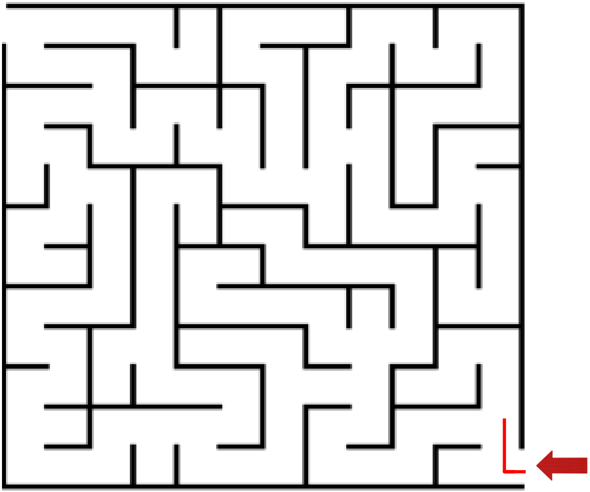
- Die ganze Zeit lang spannen wir unsere rote Schnur und markieren damit bereits gesehene Wege.
- Wenn wir auf eine Gabelung stoßen, gehen wir immer einen Weg, den wir noch nicht gesehen haben.
- Wenn wir auf eine Sackgasse stoßen, gehen wir zurück zu der letzten Gabelung, in der sich noch ein ungesehener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

Tiefensuche
Einführung



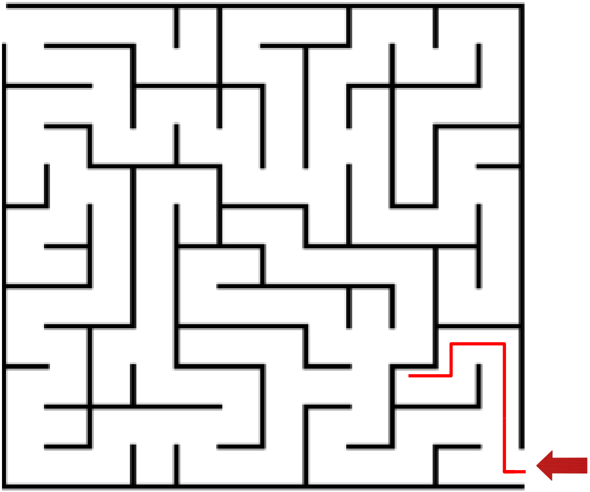
Tiefensuche

Einführung



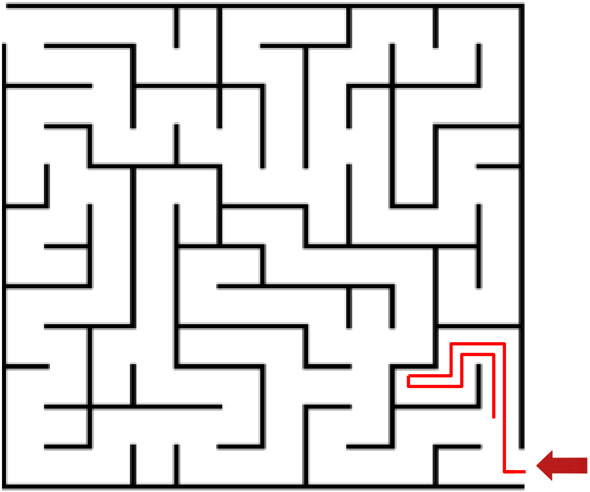
Tiefensuche

Einführung



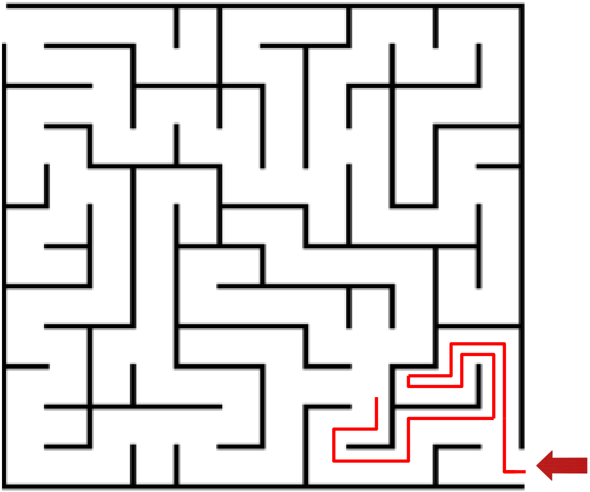
Tiefensuche

Einführung



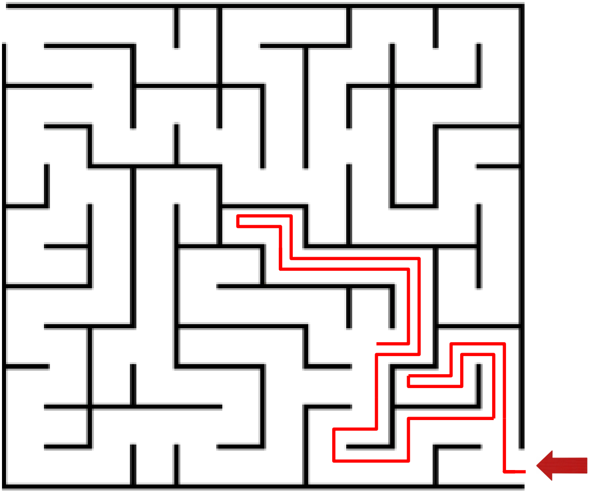
Tiefensuche

Einführung



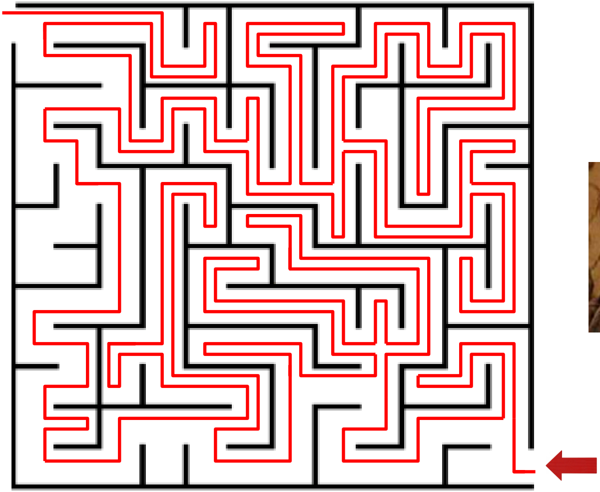
Tiefensuche

Einführung



Tiefensuche

Einführung



Tiefensuche

rekursive Implementierung

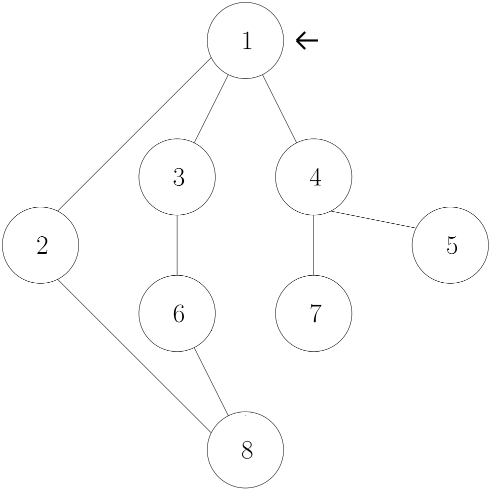
```
typedef vector<int> vi;
typedef pair<int,int> ii;
typedef vector<ii> vii;

vector <vii> AdjList;
vector <bool> dfs_visited;

void dfs(int u) {
        dfs_visited[u] = true;
        for (int i = 0; i < (int) AdjList[u].size(); i++) {
                ii v = AdjList[u][i];
                if (dfs_visited[v.first] == false) {
                        dfs(v.first);
                }
        }
}
```

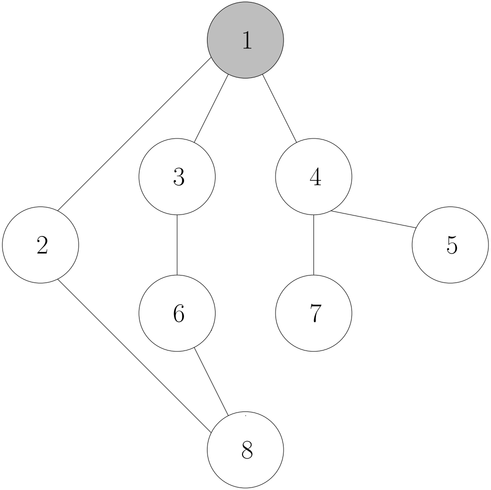
Tiefensuche

graphische Darstellung

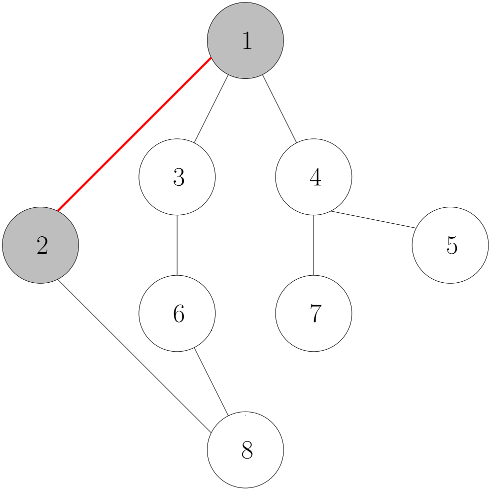


Tiefensuche

graphische Darstellung

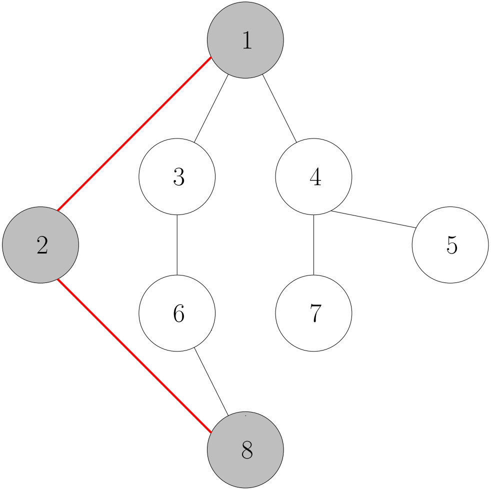


Tiefensuche
graphische Darstellung



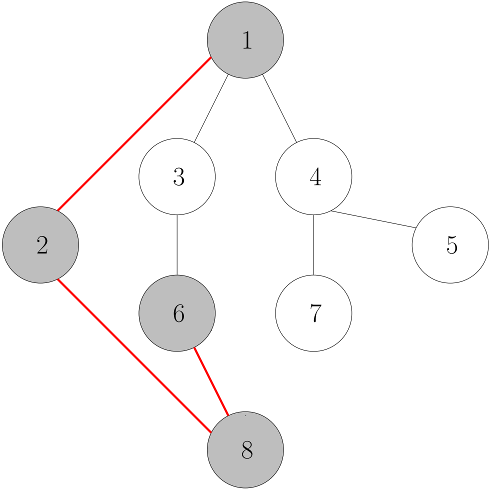
Tiefensuche

graphische Darstellung



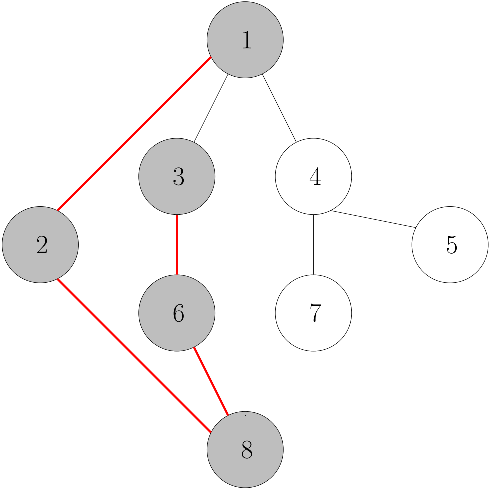
Tiefensuche

graphische Darstellung

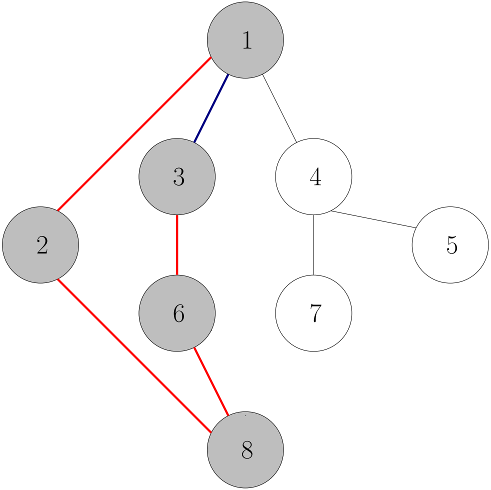


Tiefensuche

graphische Darstellung

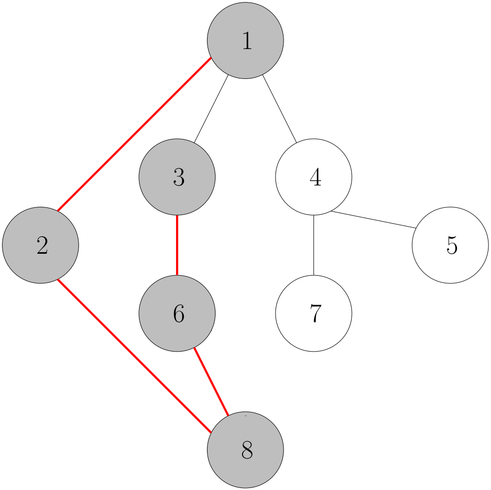


Tiefensuche
graphische Darstellung



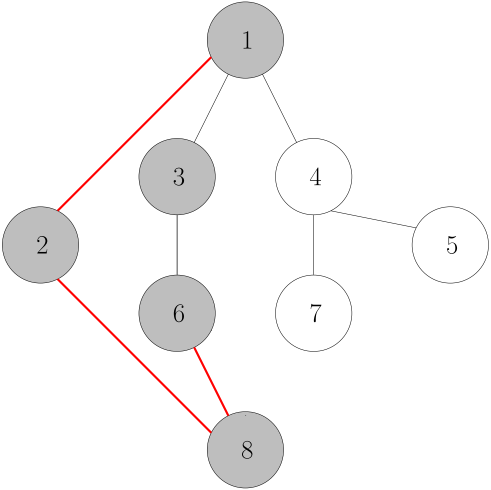
Tiefensuche

graphische Darstellung

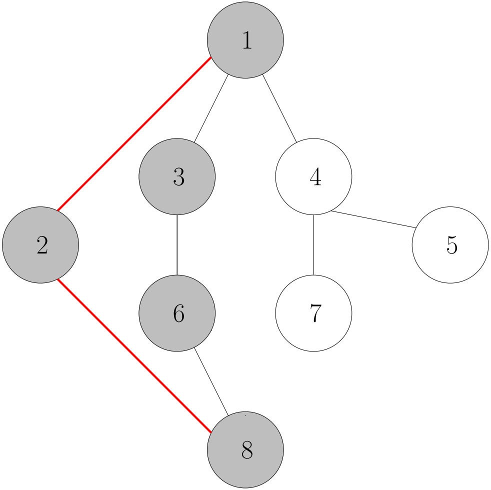


Tiefensuche

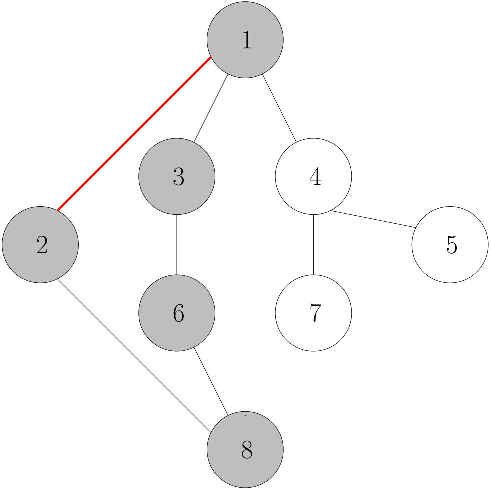
graphische Darstellung



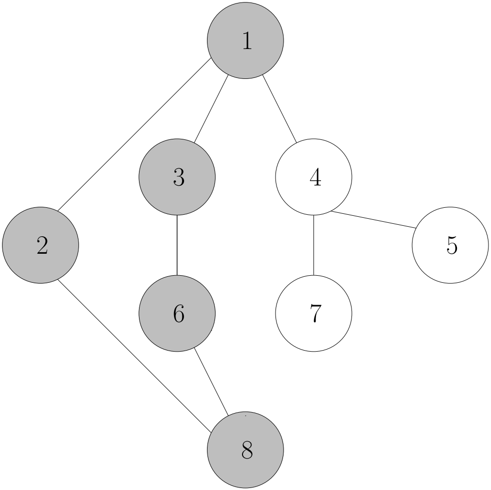
Tiefensuche
graphische Darstellung



Tiefensuche
graphische Darstellung

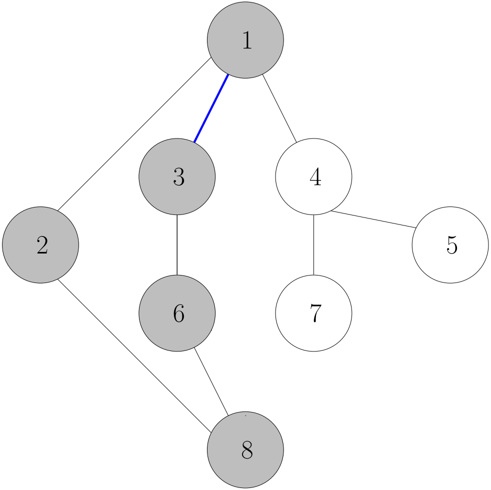


Tiefensuche
graphische Darstellung



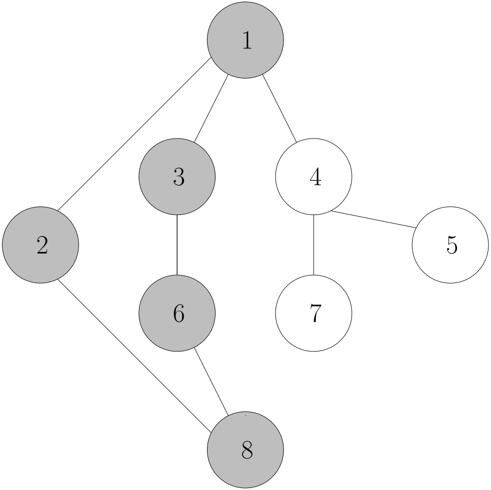
Tiefensuche

graphische Darstellung

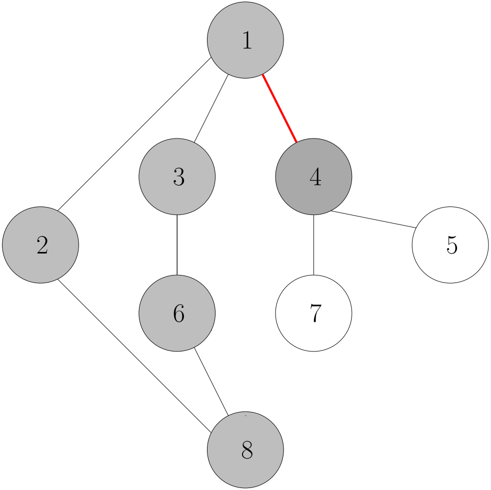


Tiefensuche

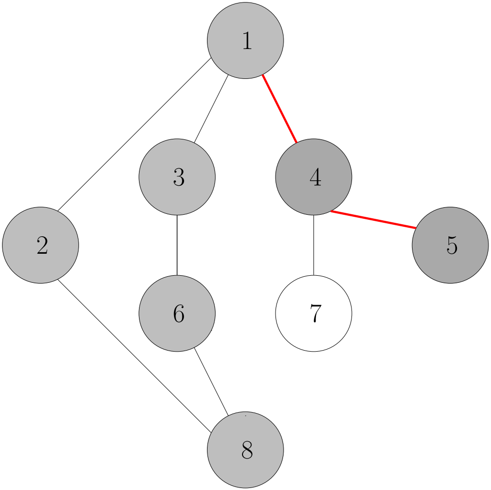
graphische Darstellung



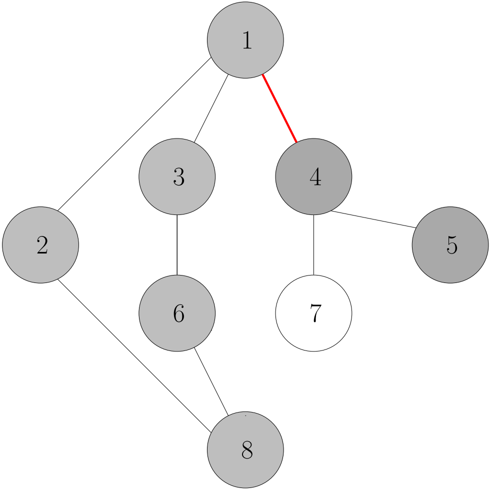
Tiefensuche
graphische Darstellung



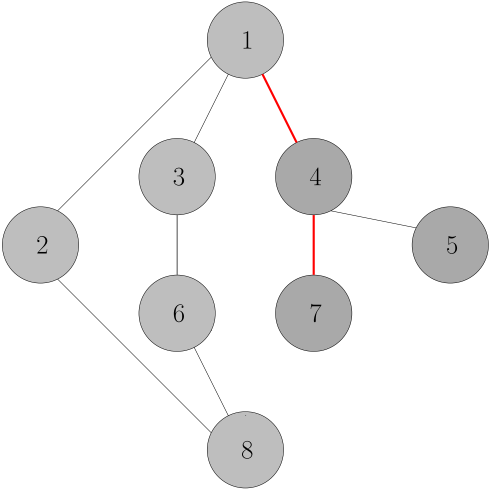
Tiefensuche
graphische Darstellung



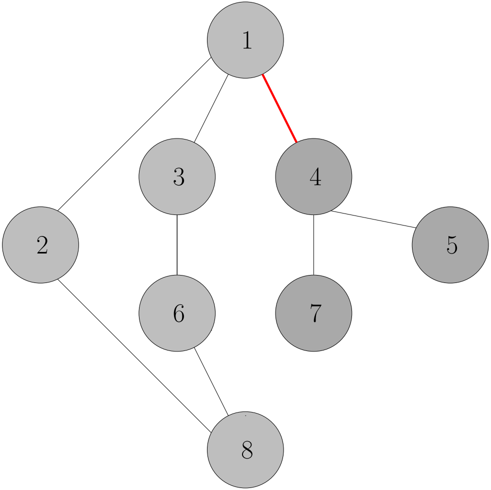
Tiefensuche
graphische Darstellung



Tiefensuche
graphische Darstellung

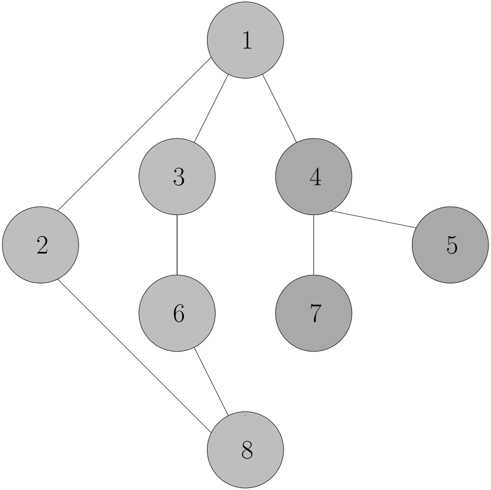


Tiefensuche
graphische Darstellung



Tiefensuche

graphische Darstellung



Tiefensuche
informell

- Laufzeit: $\mathcal{O}(|V| + |E|)$, wenn der Graph als Adjazenzliste gespeichert ist
- Die Tiefensuche ist nicht optimal, wird im Allgemeinen nicht die kürzeste Verbindung zum Ziel gefunden.
- Der Speicherbrauch ist linear, da nur Informationen darüber gespeichert werden, ob ein Knoten schon besucht wurde.

Tiefensuche

mögliche Probleme

- Der Graph ist unendlich groß, in diesem Fall gibt es keine Möglichkeit, für jeden Knoten Informationen zu speichern.
- Es ist nach einem optimalen Ergebnis bezüglich der Pfadlänge gefragt.
- Durch die vielen Rekursionsebenen wird das Rekursionslimit vom Betriebssystem überschritten.

Tiefensuche

nichtrekursive Implementierung

```
void nrdfs(int u) {
        dfs_visited[u] = true;
        stack<ii> myStack;
        int pos = 0;
        int i = u;
        while (!myStack.empty() || pos < AdjList[i].size()) {
                if (pos < (int) AdjList[i].size()) {
                        ii v = AdjList[i][pos];
                        if (dfs_visited[v.first] == false) {
                                ii p (i, pos + 1);
                                myStack.push(p);
                                i = v.first;
                                pos = 0;
                                dfs_visited[v.first] = true;
                        } else {
```

Tiefensuche

nichtrekursive Implementierung

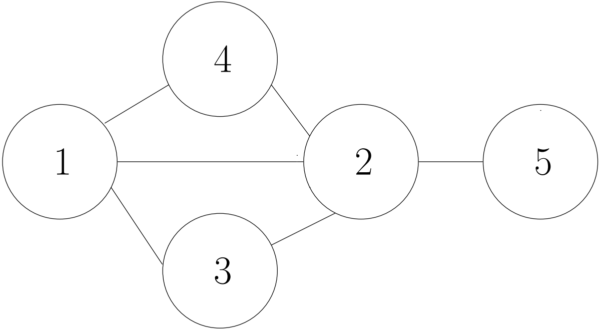
```
                        pos++;
                    }
                } else {
                    ii p = myStack.top();
                    myStack.pop();
                    i = p.first;
                    pos = p.second;
                }
        }
}
```

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.

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

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.

Bipartite Graphen

Bipartite Graphen

Definition

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

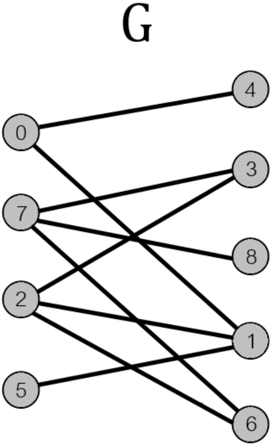
- G heißt **bipartit** falls sich seine Knoten in zwei disjunkte Teilmengen A, B aufteilen lassen, so dass es zwischen den Knoten innerhalb einer Teilmenge keine Kanten gibt.
- Das heißt, für jede Kante $\{u, v\} \in E$ gilt entweder $u \in A$ und $v \in B$ oder $u \in B$ und $v \in A$.

Bipartite Graphen

UVa 10004 Bicoloring

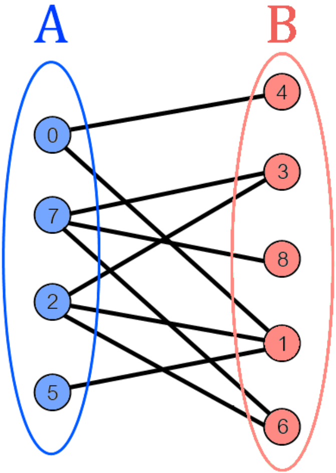
Ein ungerichteter Graph wird als Eingabe gegeben. Die Aufgabe ist zu bestimmen ob der Graph eingefärbt werden kann, so dass keine zwei benachbarten Knoten die gleiche Farbe haben.

Bipartite Graphen
Beispiel



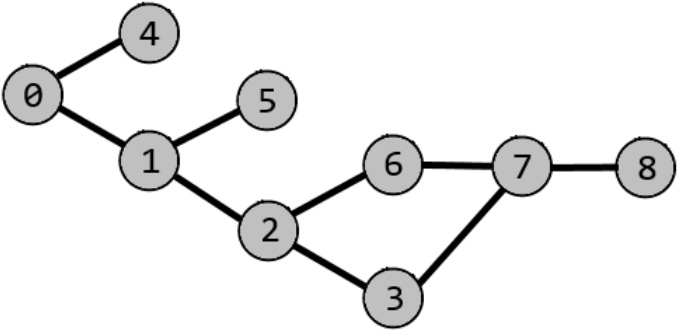
Bipartite Graphen

Beispiel



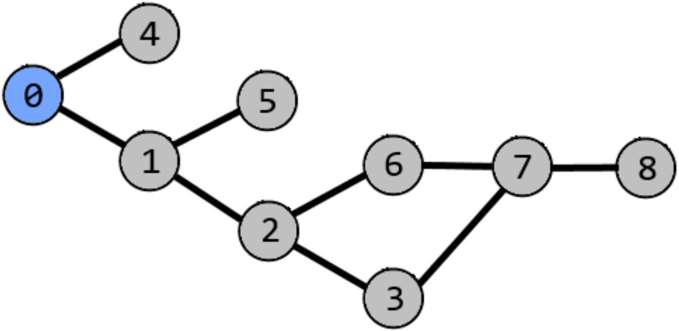
Bipartite Graph Check

DFS Algorithmus



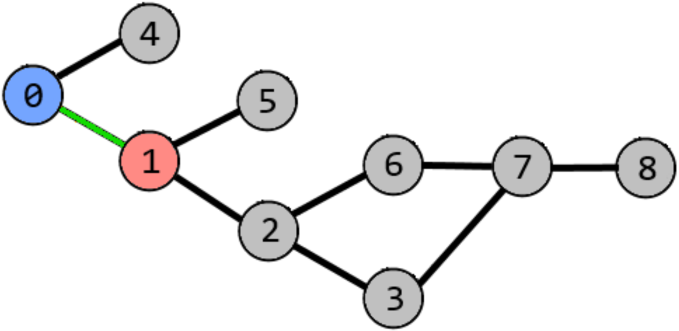
Bipartite Graph Check

DFS Algorithmus



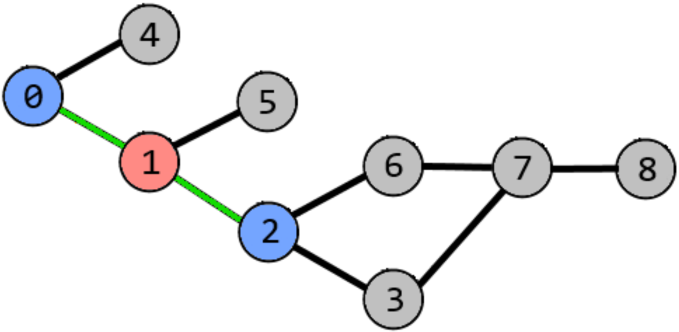
Bipartite Graph Check

DFS Algorithmus



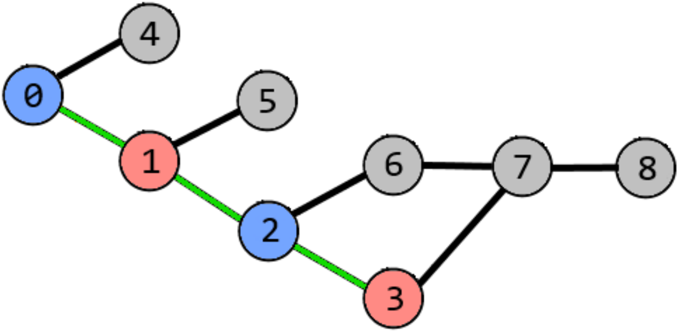
Bipartite Graph Check

DFS Algorithmus



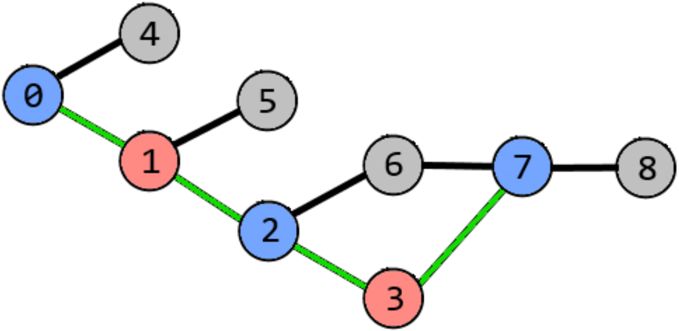
Bipartite Graph Check

DFS Algorithmus



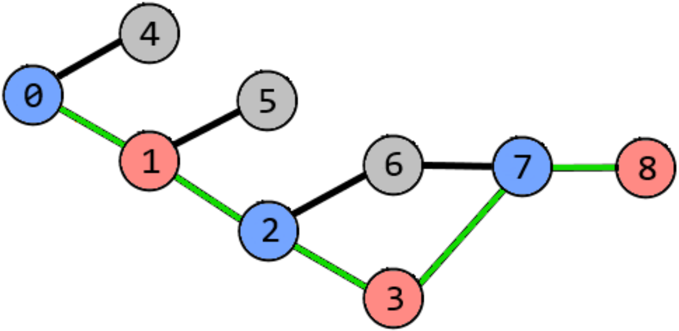
Bipartite Graph Check

DFS Algorithmus



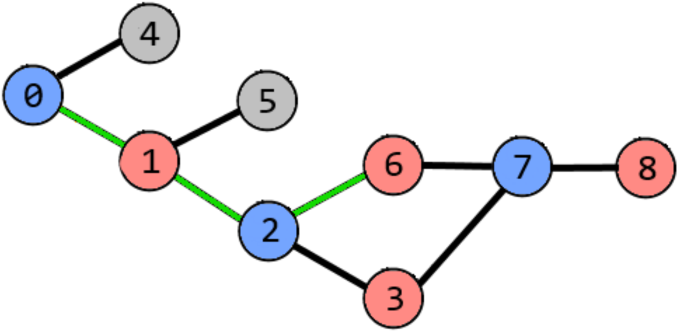
Bipartite Graph Check

DFS Algorithmus



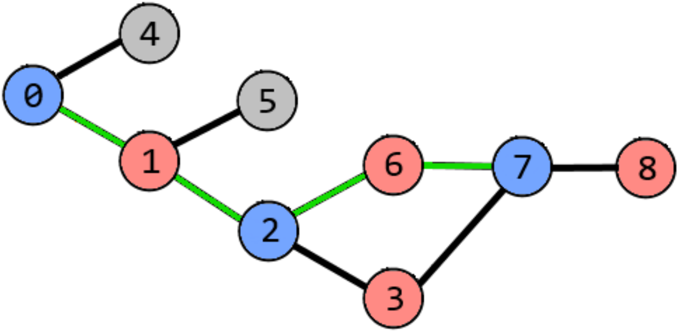
Bipartite Graph Check

DFS Algorithmus



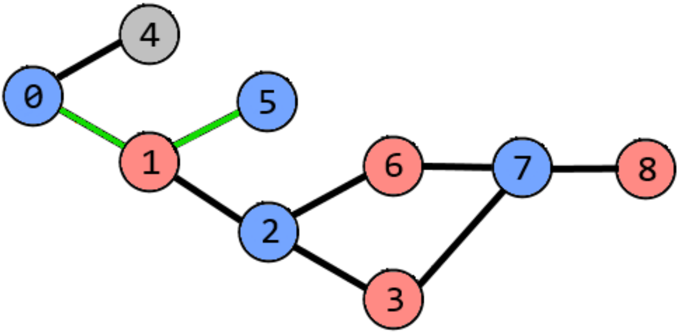
Bipartite Graph Check

DFS Algorithmus



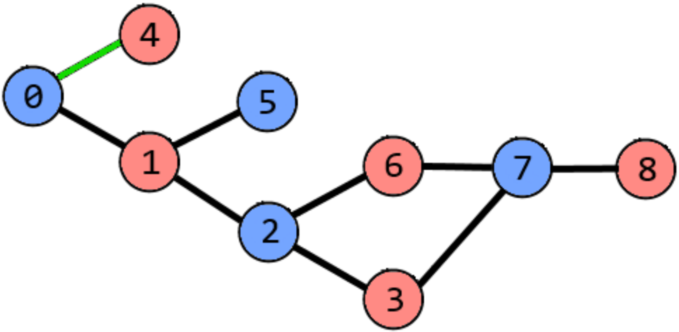
Bipartite Graph Check

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 checkDFS(int v, int color) //int main() −> checkDFS(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

Begrenzte Tiefensuche

- **Begrenzte Tiefensuche**: wird als eine DFS durchgeführt, die nicht tiefer als eine Tiefe T sucht (tiefere Knoten werden ignoriert).
- Für sich allein nur in besonderen Situationen nützlich, ist aber eine wichtige Komponente in Iterative Tiefensuche.

Iterative Tiefensuche

Rekursive begrenzte DFS Algorithmus

```
vector<node> goalPath;
int limitedDFS(node v, int depth)
{
        if(v == goal) {
                goalPath.push_back(v);
                return 1;
        }

        if(depth > 0) {
                vector<node> neighbors = neighborhood(v);
                for( int i = 0; i < neighbors.size(); i++) {
                        if( limitedDFS(neighbors[i], depth-1) == 1) {
                                goalPath.push_back(v);
                                return 1;
        }       }       }
        return 0;
}
```

Iterative Tiefensuche

Iterative Tiefensuche

- Idee: Führe eine Folge begrenzter Tiefensuchen mit ansteigender Tiefengrenze durch bis eine Lösung gefunden wird.
- Sei b der durchschnittliche Knotengrad und d die minimale Lösungstiefe eines gegebenen Graphen G.

  Der Zeitaufwand der iterativen Tiefensuche ist dann $(d+1)+db+(d-1)b^2+(d-2)b^3+...+2b^{d-1}+b^d=\mathcal{O}(b^d)$ und der Speicheraufwand beträgt $\mathcal{O}(bd)$.
  - BFS $\Rightarrow 1+b+b^2+...+b^d=\mathcal{O}(b^d)$

```
void iterativeDFS(node origin) {
        int depth = 0;
        while( limitedDFS(origin,depth) != 1)
                depth++;
        return;
}
```

Iterative Tiefensuche

Iterative Tiefensuche

- Idee: Führe eine Folge begrenztzter Tiefensuchen mit ansteigender Tiefengrenze durch bis eine Lösung gefunden wird.
- Sei b der durchschnittliche Knotengrad und d die minimale Lösungstiefe eines gegebenen Graphen G.

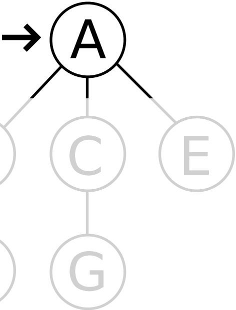
  Der Zeitaufwand der iterativen Tiefensuche ist dann $(d+1) + db + (d-1)b^2 + (d-2)b^3 + ... + 2b^{d-1} + b^d = \mathcal{O}(b^d)$ und der Speicheraufwand beträgt $\mathcal{O}(bd)$.
  - BFS $\Rightarrow 1 + b + b^2 + ... + b^d = \mathcal{O}(b^d)$

```
void iterativeDFS(node origin) {
        int depth = 0;
        while( limitedDFS(origin,depth) != 1)
                depth++;
        return;
}
```


Fakultät für Informatik
Institut für Theoretische Informatik

Iterative Tiefensuche

Beispiel (gerichteter Graph)

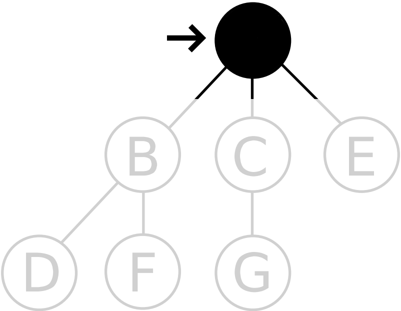


Wurzel: A

d = 0: A

Iterative Tiefensuche

Beispiel (gerichteter Graph)



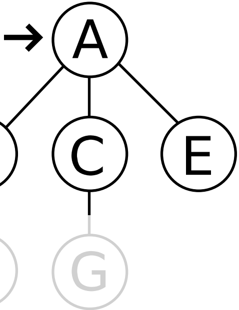
Wurzel: A

d = 0: A

Fakultät für Informatik
Institut für Theoretische Informatik

Iterative Tiefensuche

Beispiel (gerichteter Graph)



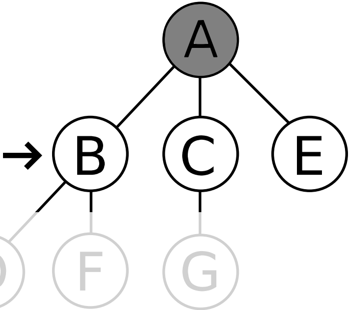
Wurzel: A

d = 0: A

d = 1: A

Iterative Tiefensuche

Beispiel (gerichteter Graph)



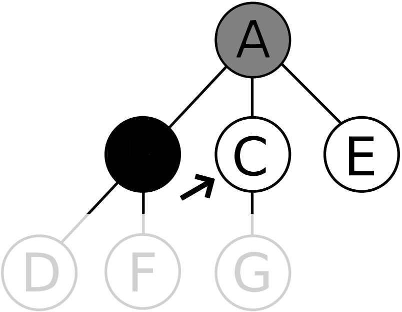
Wurzel: A

d = 0: A

d = 1: AB

Iterative Tiefensuche

Beispiel (gerichteter Graph)



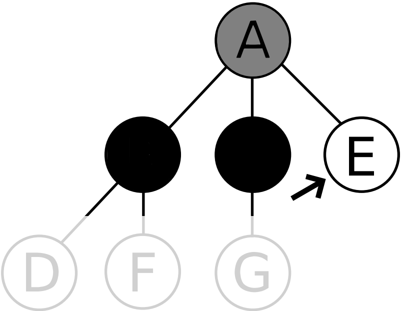
Wurzel: A

d = 0: A

d = 1: ABC

Iterative Tiefensuche

Beispiel (gerichteter Graph)



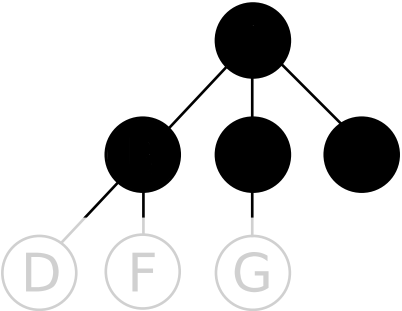
Wurzel: A

d = 0: A

d = 1: ABCE

Iterative Tiefensuche

Beispiel (gerichteter Graph)



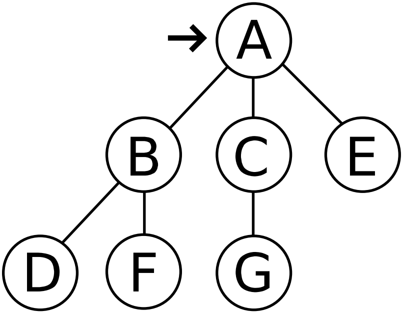
Wurzel: A

d = 0: A

d = 1: ABCE

Iterative Tiefensuche

Beispiel (gerichteter Graph)



Wurzel: A

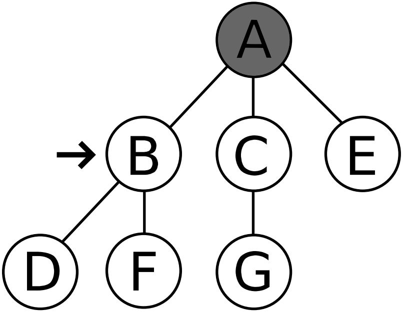
d = 0: A

d = 1: ABCE

d = 2: A

Iterative Tiefensuche

Beispiel (gerichteter Graph)



Wurzel: A

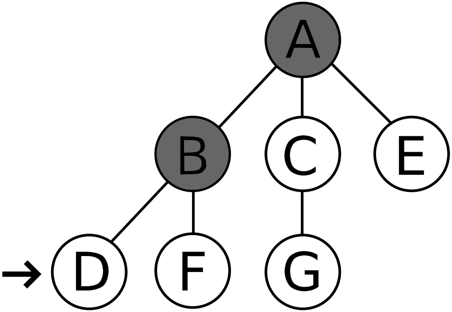
d = 0: A

d = 1: ABCE

d = 2: AB

Iterative Tiefensuche

Beispiel (gerichteter Graph)



Wurzel: A

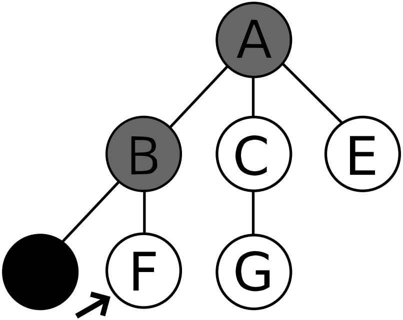
d = 0: A

d = 1: ABCE

d = 2: ABD

Iterative Tiefensuche

Beispiel (gerichteter Graph)



Wurzel: A

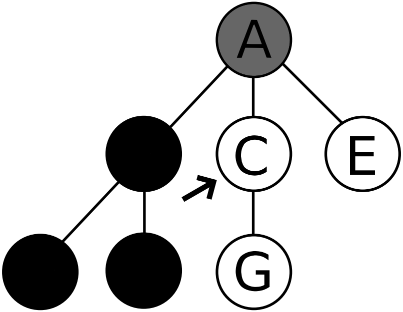
d = 0: A

d = 1: ABCE

d = 2: ABDF

Iterative Tiefensuche

Beispiel (gerichteter Graph)



Wurzel: A

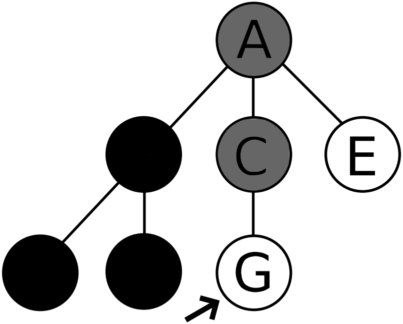
d = 0: A

d = 1: ABCE

d = 2: ABDFC

Iterative Tiefensuche

Beispiel (gerichteter Graph)



Wurzel: A

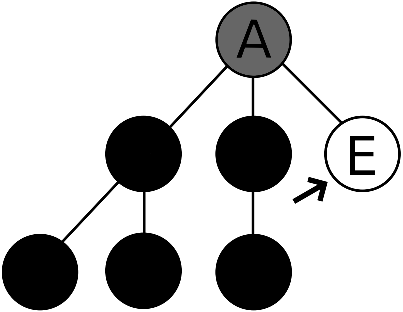
d = 0: A

d = 1: ABCE

d = 2: ABDFCG

Iterative Tiefensuche

Beispiel (gerichteter Graph)



Wurzel: A

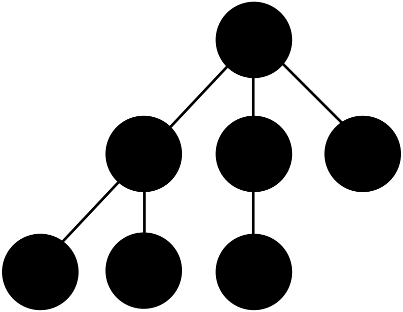
d = 0: A

d = 1: ABCE

d = 2: ABDFCGE

Iterative Tiefensuche

Beispiel (gerichteter Graph)



Wurzel: A

d = 0: A

d = 1: ABCE

d = 2: ABDFCGE

Iterative Tiefensuche

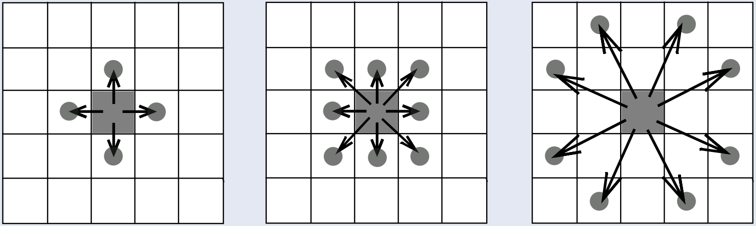
Warte mal: jede Iteration besucht die vorherigen Knoten noch einmal! Warum dann nicht BFS oder DFS benutzen?

Fakultät für Informatik
Institut für Theoretische Informatik

Iterative Tiefensuche

Bewertung

- Einige Graphen können mit herkömmlichen Suchverfahren nicht durchgelaufen werden, bspw. implizite Graphen die unendlich Groß sind und nicht im Speicher gehalten werden können.



Von-Neumann-Nachbarnschaft   Moore-Nachbarnschaft   Pferd-Nachbarnschaft

Iterative Tiefensuche

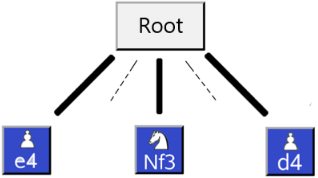
Schach

- Auch sinnvoll für Probleme, die einen sehr hohen branching factor b haben, z.B.: **Schach**
- Im Durchschnitt hat ein Schachspieler für jede Position 35~38 mögliche Züge. Das heißt, nach 5 Zügen werden ~10^7 Positionen möglich, nach 10 Zügen ~10^{15}.
- Adaptierte iterative Tiefensuche berechnet den besten Zug bis entweder die Zeit abgelaufen ist oder die maximale Suchtiefe erreicht ist.
- Suche kann mit Heuristiken verbessert werden, so dass die vielversprechendsten Richtungen zuerst erforscht sind.

Iterative Tiefensuche

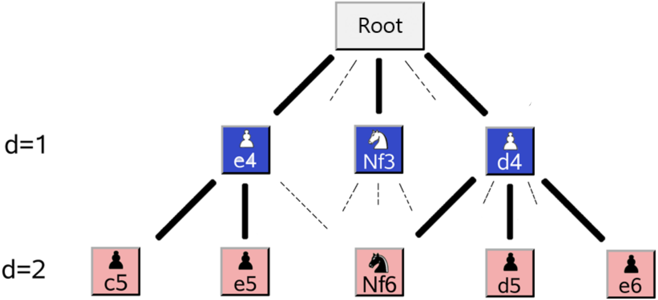
Schach

d=1



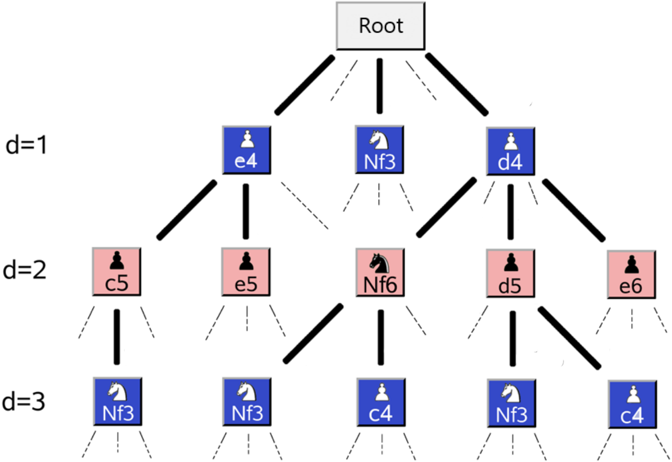
Iterative Tiefensuche

Schach



Iterative Tiefensuche

Schach



Iterative Tiefensuche

Source code: www.craftychess.com/crafty-23.4.zip

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
Idee des 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):** min $\{\textbf{dfs\_num(v)} \mid v \textit{ erreichbar von } u\}$
- SCCs werden mittels dem von der DFS erzeugten Spannbaum gefunden
- Wenn Backedge von einem Knoten u zur Wurzel r eines Teilbaums, sind alle Knoten auf
  dem Weg zwischen u und v in einer SCC.

SCCs finden mittels DFS

Idee des 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):** $\min \{\textbf{dfs\_num(v)} | v \textit{ erreichbar von } u\}$
- SCCs werden mittels dem von der DFS erzeugten Spannbaum gefunden
- Wenn Backedge von einem Knoten u zur Wurzel r eines Teilbaums, sind alle Knoten auf dem Weg zwischen u und v in einer SCC.

SCCs finden mittels DFS
Idee des 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):** $\min\{\textbf{dfs\_num(v)} \mid v \textit{ erreichbar von } u\}$
- SCCs werden mittels dem von der DFS erzeugten Spannbaum gefunden
- Wenn Backedge von einem Knoten u zur Wurzel r eines Teilbaums, sind alle Knoten auf dem Weg zwischen u und v in einer SCC.

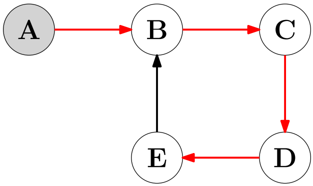
SCCs finden mittels DFS

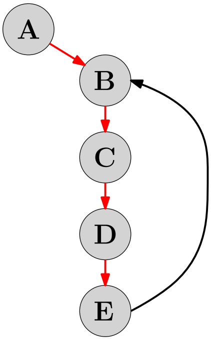
Idee des Algorithmus

- 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):** $\min\{\textbf{dfs\_num(v)} | v \textit{ erreichbar von } u\}$
- SCCs werden mittels dem von der DFS erzeugten Spannbaum gefunden
- Wenn Backedge von einem Knoten u zur Wurzel r eines Teilbaums, sind alle Knoten auf dem Weg zwischen u und v in einer SCC.

SCCs finden mittels DFS

DFS-Spannbaum





SCCs finden mittels DFS

Idee des 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):** min $\{$**dfs_num(v)**$|v\ \textit{erreichbar von}\ u\}$

- SCCs werden mittels dem von der DFS erzeugten Spannbaum gefunden
- Wenn Backedge von einem Knoten u zur Wurzel r eines Teilbaums, sind alle Knoten auf
  dem Weg zwischen u und v in einer SCC.
- Wenn **dfs_low(v)** = **dfs_num(v)**, dann ist v die "Wurzel" einer SCC
- Knoten werden - sobald besucht - auf **STACK** gespeichert und wenn Wurzel gefunden, ausgegeben

SCCs finden mittels DFS

Idee des Algorithmus

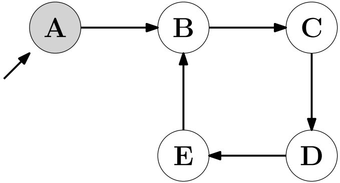
- Führe eine DFS im Graph durch.
- Besuchte Knoten erhalten zwei Nummern:
  1. **dfs_num(u):** Speichert Schritt, in dem Knoten u von DFS besucht wurde.
  2. **dfs_low(u):** min $\{$**dfs_num(v)**$|$ v *erreichbar von* $u\}$
- SCCs werden mittels dem von der DFS erzeugten Spannbaum gefunden
- Wenn Backedge von einem Knoten u zur Wurzel r eines Teilbaums, sind alle Knoten auf
  dem Weg zwischen u und v in einer SCC.
- Wenn **dfs_low(v)** = **dfs_num(v)**, dann ist v die "Wurzel" einer SCC
- Knoten werden - sobald besucht - auf **STACK** gespeichert und wenn Wurzel gefunden, ausgegeben

SCCs finden mittels DFS
Idee des 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):** min $\{\textbf{dfs\_num(v)} | v \text{ erreichbar von } u\}$
- SCCs werden mittels dem von der DFS erzeugten Spannbaum gefunden
- Wenn Backedge von einem Knoten *u* zur Wurzel *r* eines Teilbaums, sind alle Knoten auf
  dem Weg zwischen *u* und *v* in einer SCC.
- Wenn **dfs_low(v)** = **dfs_num(v)**, dann ist *v* die "Wurzel" einer SCC
- Knoten werden - sobald besucht - auf **STACK** gespeichert und wenn Wurzel gefunden, ausgegeben

SCCs finden mittels DFS

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



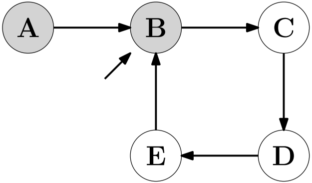
STACK:

- **A**

SCCs finden mittels DFS

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

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



STACK:

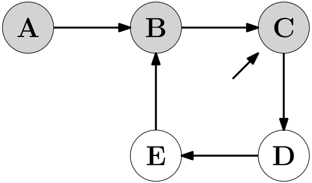
- A
- B

SCCs finden mittels DFS

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

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

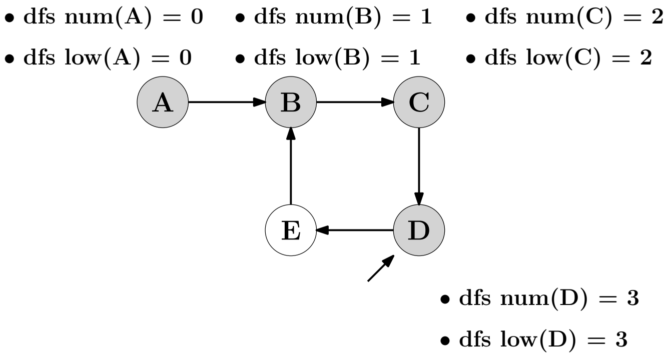
- dfs num(C) = 2
- dfs low(C) = 2



STACK:

- A
- B
- C

SCCs finden mittels DFS



STACK:

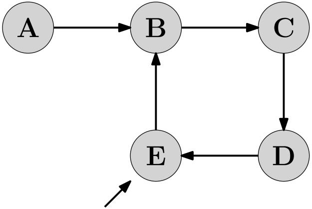
- **A**
- **B**
- **C**
- **D**

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



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

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

STACK:

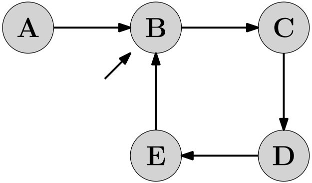
- A
- B
- C
- D
- E

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



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

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

STACK:

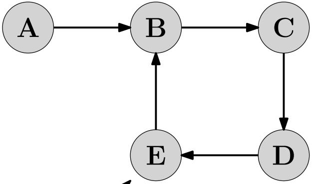
- A
- B
- C
- D
- E

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



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

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

STACK:

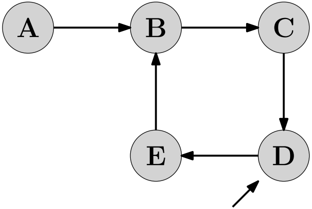
- A
- B
- C
- D
- E

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) = $\boxed{1}$



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

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

STACK:

- A
- B
- C
- D
- E

Fakultät für Informatik
Institut für Theoretische Informatik

SCCs finden mittels DFS

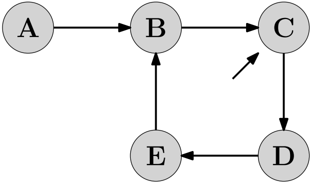
- dfs num(A) = 0
- dfs low(A) = 0

- 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

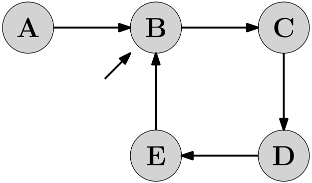
- dfs num(D) = 3
- dfs low(D) = 1

SCCs finden mittels DFS

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

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

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



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

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

STACK:

- A
- B
- C
- D

SCCs finden mittels DFS

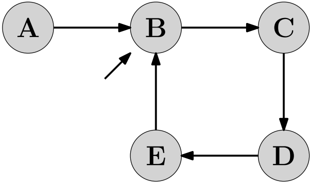
- dfs num(A) = 0
- dfs low(A) = 0

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

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

STACK:

- A
- B
- C



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

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

Fakultät für Informatik
Institut für Theoretische Informatik

SCCs finden mittels DFS

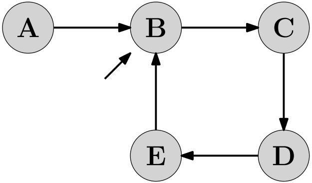
- $\text{dfs num}(A) = 0$
- $\text{dfs low}(A) = 0$

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

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

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

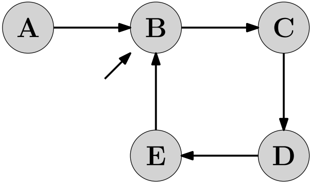
- dfs num(A) = 0
- dfs low(A) = 0

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

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

STACK:

- **A**



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

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

SCCs finden mittels DFS

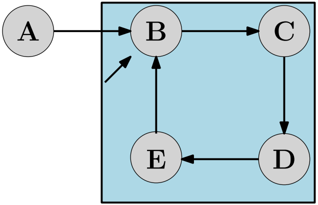
- dfs num(A) = 0
- dfs low(A) = 0

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

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

STACK:

- A



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

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

SCCs finden mittels DFS

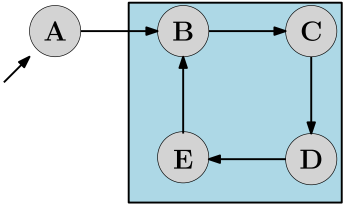
- dfs num(A) = 0
- dfs low(A) = 0

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

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

STACK:

-



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

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

SCCs finden mittels DFS

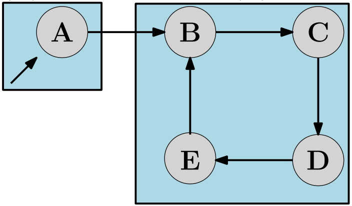
- dfs num(A) = 0
- dfs low(A) = 0

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

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

STACK:

-



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

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

SCCs finden mittels DFS
Sourcecode

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

        for (int j = 0; j < AdjacenceList[u].size(); j++) {
                ii v = AdjacenceList[u][j]; // Pair
                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

Brücken und Separatoren

Definition

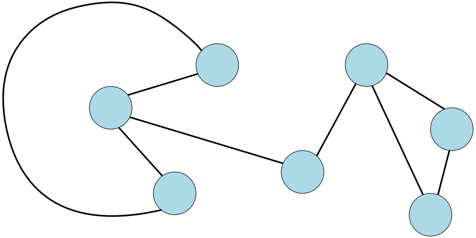
Separatoren und Brücken in ungerichteten Graphen

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.

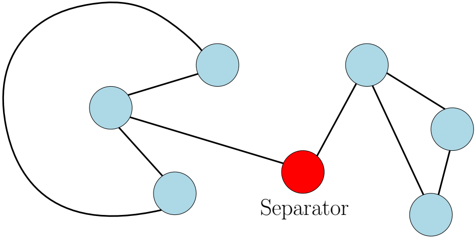
Brücken und Separatoren

Beispiel



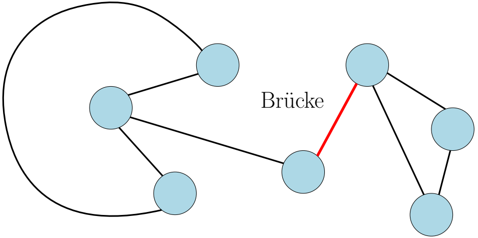
Brücken und Separatoren

Beispiel



Brücken und Separatoren

Beispiel



Fakultät für Informatik
Institut für Theoretische Informatik

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'

Numerierung

- 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):** min $\{\textbf{dfs\_num(v)} \mid v \text{ } \textit{erreichbar von } u\}$

- 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'

Numerierung

- 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):** min $\{\mathbf{dfs\_num(v)} \mid v \textit{ erreichbar von } u\}$

- 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'

Numerierung

- 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):** $\min \{\textbf{dfs\_num(v)} \mid v \textit{ erreichbar von } u\}$

- 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'

Numerierung

- 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):** min $\{$**dfs_num(v)**$|$ v *erreichbar von* $u\}$

- 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'

Numerierung

- 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):** min $\{$**dfs_num(v)**$|v$ *erreichbar von* $u\}$

- 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

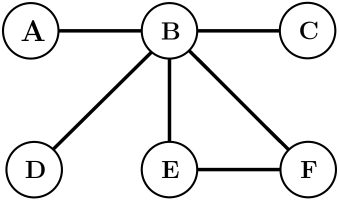
Brücken und Separatoren

Illustration des Algorithmus’

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

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

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



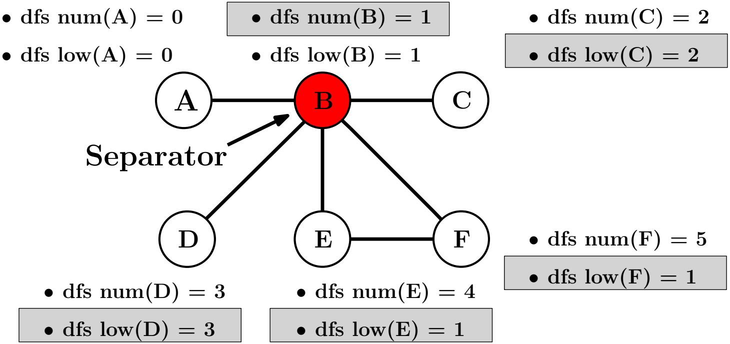
- dfs num(F) = 5
- dfs low(F) = 1

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

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

Brücken und Separatoren

Illustration des Algorithmus’



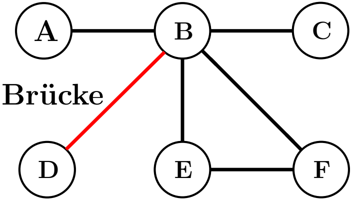
Brücken und Separatoren

Illustration des Algorithmus’

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

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

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



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

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

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