Script generated by TTT

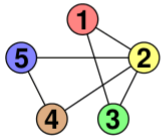
Title: Täubig: GAD (24.06.2014)

Date: Tue Jun 24 13:51:42 CEST 2014

Duration: 135:39 min

Pages: 67

Kantenliste



$\{1,2\},\{1,3\},\{2,3\},\{2,4\},\{2,5\},\{4,5\}$

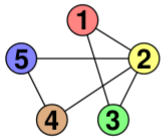
Vorteil:

- Speicherbedarf $O(m+n)$
- Einfügen von Knoten und Kanten in $O(1)$
- Löschen von Kanten per Handle in $O(1)$

Nachteil:

- G.find(Key i, Key j): im worst case $\Theta(m)$
- G.remove(Key i, Key j): im worst case $\Theta(m)$
- Nachbarn nur in $O(m)$ feststellbar

Adjazenzmatrix



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

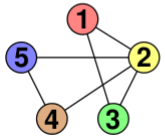
Vorteil:

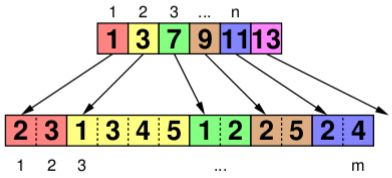
- in $O(1)$ feststellbar, ob zwei Knoten Nachbarn sind
- ebenso Einfügen und Löschen von Kanten

Nachteil:

- kostet $\Theta(n^2)$ Speicher, auch bei Graphen mit $o(n^2)$ Kanten
- Finden aller Nachbarn eines Knotens kostet $O(n)$
- Hinzufügen neuer Knoten ist schwierig

Adjazenzarrays





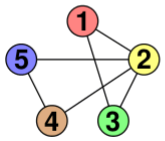
Vorteil:

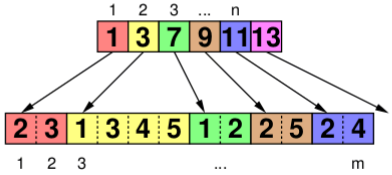
- Speicherbedarf:
  gerichtete Graphen: $n+m+\Theta(1)$
  (hier noch kompakter als Kantenliste mit $2m$)
  ungerichtete Graphen: $n+2m+\Theta(1)$

Nachteil:

- Einfügen und Löschen von Kanten ist schwierig, deshalb nur für statische Graphen geeignet

Adjazenzarrays





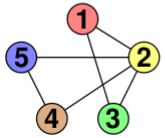
Vorteil:

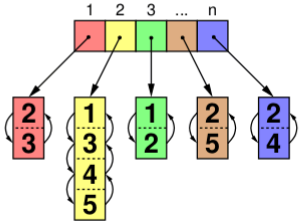
- Speicherbedarf:
  gerichtete Graphen: $n + m + \Theta(1)$
  (hier noch kompakter als Kantenliste mit $2m$)
  ungerichtete Graphen: $n + 2m + \Theta(1)$

Nachteil:

- Einfügen und Löschen von Kanten ist schwierig, deshalb nur für statische Graphen geeignet

Adjazenzlisten





Unterschiedliche Varianten:
einfach/doppelt verkettet, linear/zirkulär

Vorteil:

- Einfügen von Kanten in $O(d)$ oder $O(1)$
- Löschen von Kanten in $O(d)$ (per Handle in $O(1)$)
- mit unbounded arrays etwas cache-effizienter

Nachteil:

- Zeigerstrukturen verbrauchen relativ viel Platz und Zugriffszeit

Adjazenzliste + Hashtabelle

- speichere Adjazenzliste (Liste von adjazenten Knoten bzw. inzidenten Kanten zu jedem Knoten)
- speichere Hashtabelle, die zwei Knoten auf einen Zeiger abbildet, der dann auf die ggf. vorhandene Kante verweist

Zeitaufwand:

- G.find(Key i, Key j): $O(1)$ (worst case)
- G.insert(Edge e): $O(1)$ (im Mittel)
- G.remove(Key i, Key j): $O(1)$ (im Mittel)

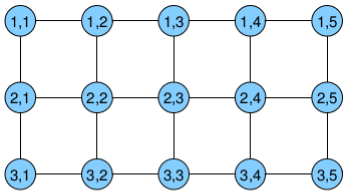
Speicheraufwand: $O(n + m)$

Implizite Repräsentation

Beispiel: Gitter-Graph (grid graph)

- definiert durch zwei Parameter k und ℓ

$$
\begin{aligned}
V &= [1,\ldots,k] \times [1,\ldots,\ell] \\
E &= \big\{((i,j),(i,j')) \in V^2 : |j-j'| = 1\big\} \cup \\
&\quad \big\{((i,j),(i',j)) \in V^2 : |i-i'| = 1\big\}
\end{aligned}
$$



- Kantengewichte könnten in 2 zweidimensionalen Arrays gespeichert werden:
  eins für waagerechte und eins für senkrechte Kanten

Übersicht

Graphtraversierung

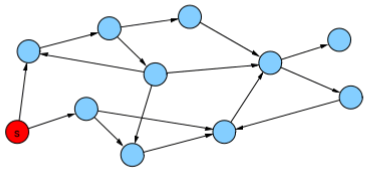
Problem:

Wie kann man die Knoten eines Graphen systematisch durchlaufen?

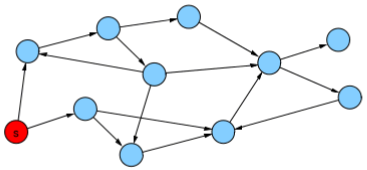
Grundlegende Strategien:

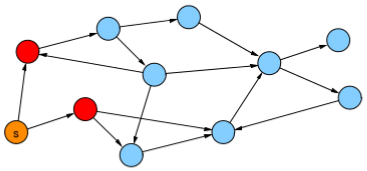
- Breitensuche (breadth-first search, BFS)
- Tiefensuche (depth-first search, DFS)

Breitensuche

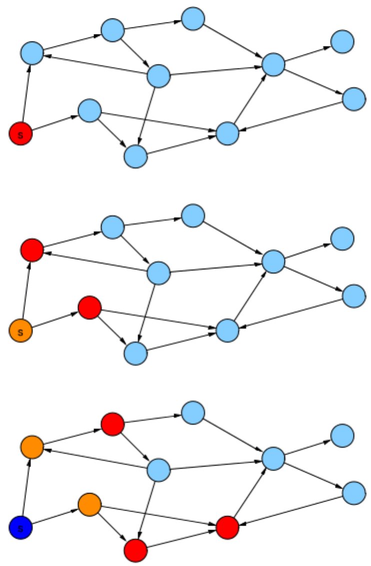


Breitensuche

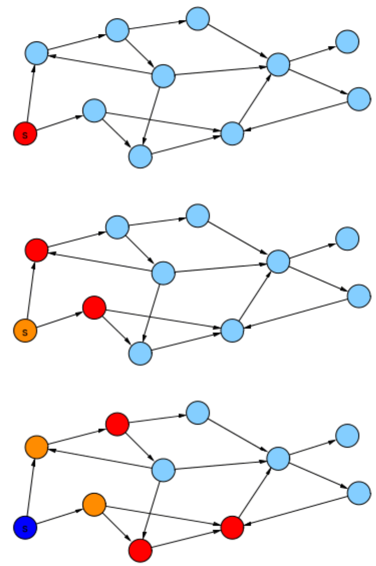


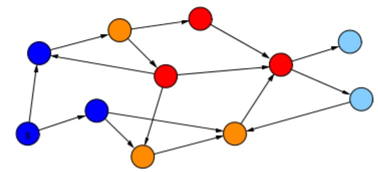


Breitensuche

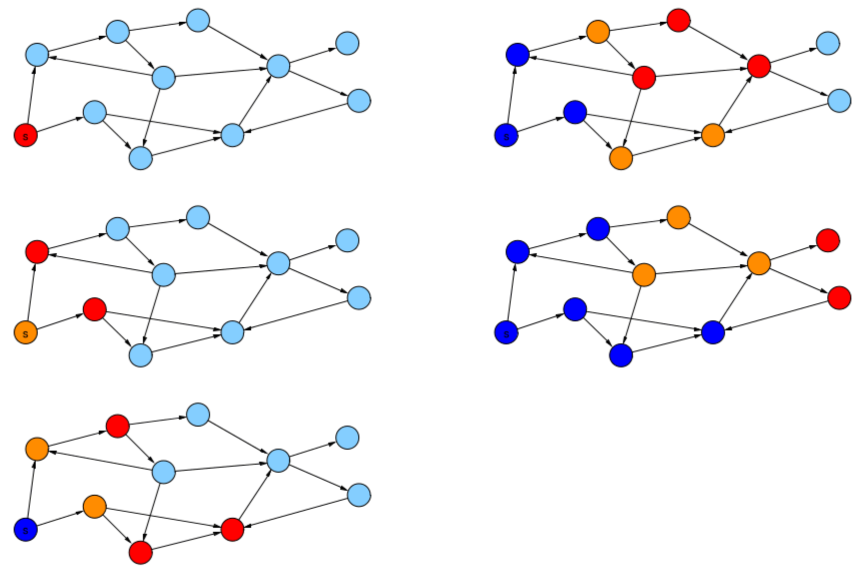


Breitensuche



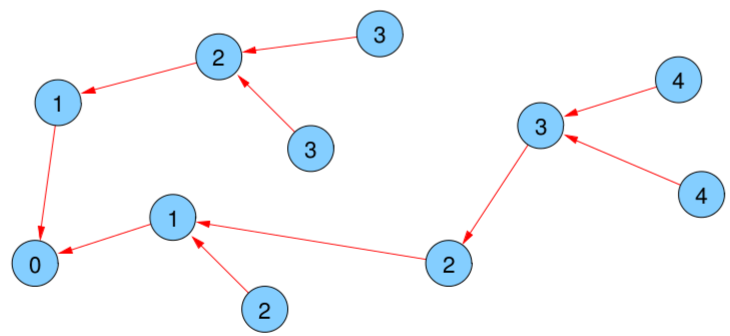


Breitensuche



Breitensuche

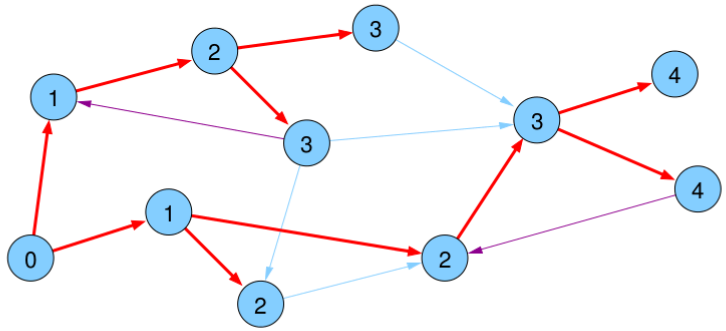
- parent(v): Knoten, von dem v entdeckt wurde
- parent wird beim ersten Besuch von v gesetzt ($\Rightarrow$ eindeutig)



Breitensuche

Kantentypen:

- Baumkanten: zum Kind
- Rückwärtskanten: zu einem Vorfahren
- Kreuzkanten: sonstige



Breitensuche

```
BFS(Node s) {
    d[s] = 0;
    parent[s] = s;
    List<Node> q = ⟨s⟩;
    while ( !q.empty() ) {
        u = q.popFront();
        foreach ((u, v) ∈ E) {
            if (parent[v] == null) {
                q.pushBack(v);
                d[v] = d[u]+1;
                parent[v] = u;
            }
        }
    }
}
```


Adjazenzliste + Hashtabelle

- speichere Adjazenzliste (Liste von adjazenten Knoten bzw. inzidenten Kanten zu jedem Knoten)
- speichere Hashtabelle, die zwei Knoten auf einen Zeiger abbildet, der dann auf die ggf. vorhandene Kante verweist

Zeitaufwand:

- G.find(Key i, Key j): $O(1)$ (worst case)
- G.insert(Edge e): $O(1)$ (im Mittel)
- G.remove(Key i, Key j): $O(1)$ (im Mittel)

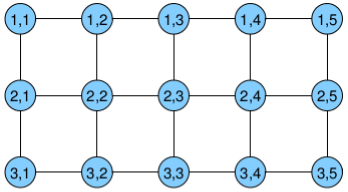
Speicheraufwand: $O(n+m)$

Implizite Repräsentation

Beispiel: Gitter-Graph (grid graph)

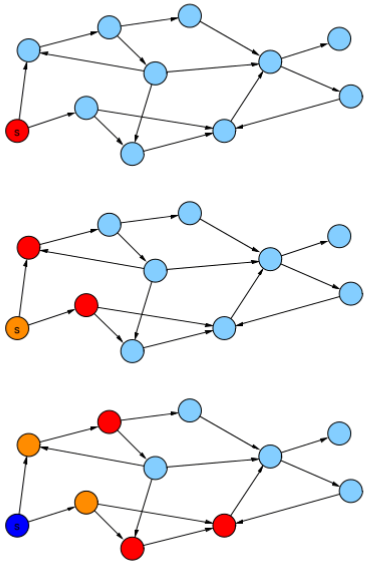
- definiert durch zwei Parameter k und ℓ

$$
\begin{aligned}
V &= [1,\ldots,k] \times [1,\ldots,\ell] \\
E &= \left\{((i,j),(i,j')) \in V^2 : |j-j'| = 1\right\} \cup \\
&\quad \left\{((i,j),(i',j)) \in V^2 : |i-i'| = 1\right\}
\end{aligned}
$$



- Kantengewichte könnten in 2 zweidimensionalen Arrays gespeichert werden:
  eins für waagerechte und eins für senkrechte Kanten

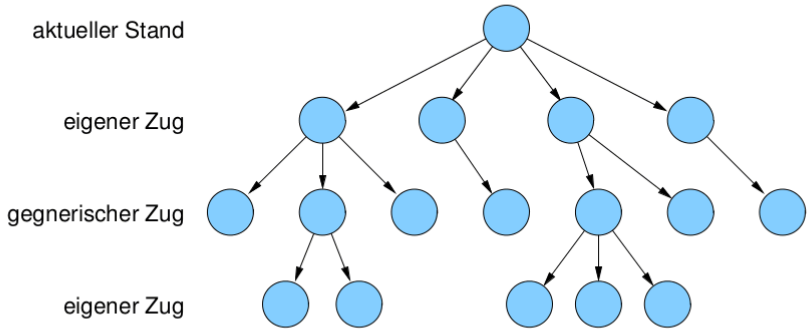
Breitensuche



Breitensuche

Anwendung: Bestimmung des nächsten Zugs bei Spielen

Exploration des Spielbaums



Problem: halte Aufwand zur Suche eines guten Zuges in Grenzen

Breitensuche

Anwendung: Bestimmung des nächsten Zugs bei Spielen

- Standard-BFS: verwendet FIFO-Queue

  ebenenweise Erkundung

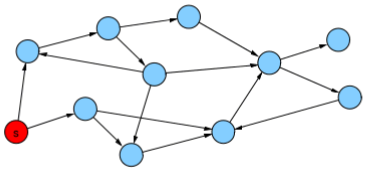
  aber: zu teuer!

- Best-First Search: verwendet Priority Queue

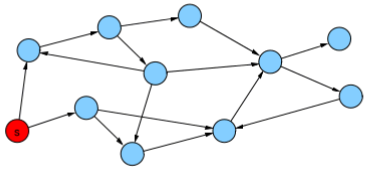
  (z.B. realisiert durch binären Heap)

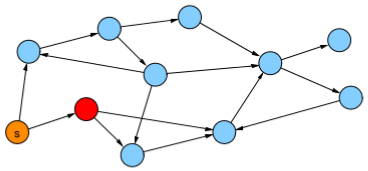
  Priorität eines Knotens wird durch eine Güte-Heuristik des repräsentierten Spielzustands gegeben

Tiefensuche

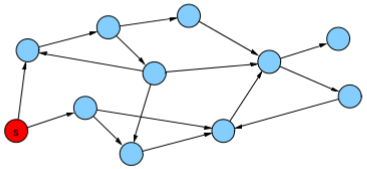


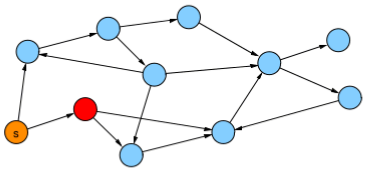
Tiefensuche

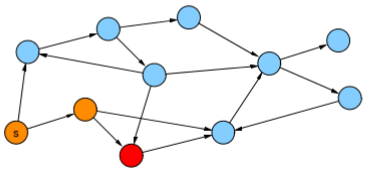




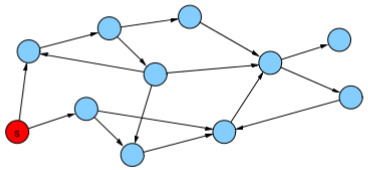
Tiefensuche

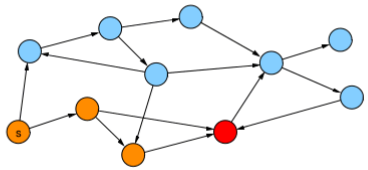


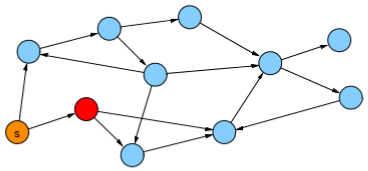


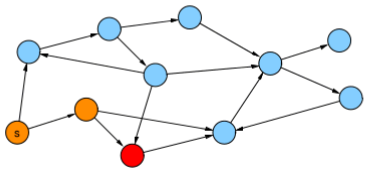


Tiefensuche

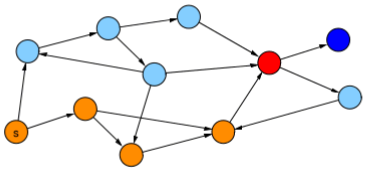




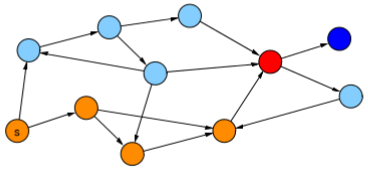


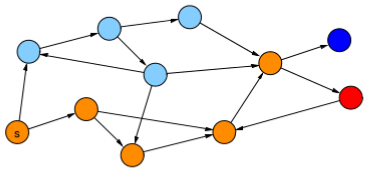


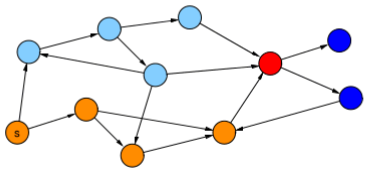
Tiefensuche



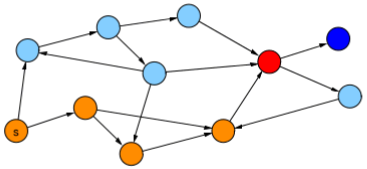
Tiefensuche

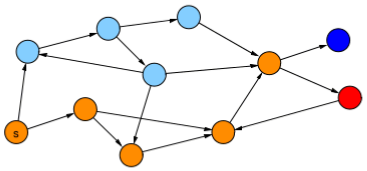


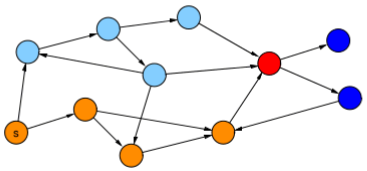




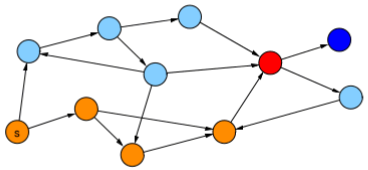
Tiefensuche

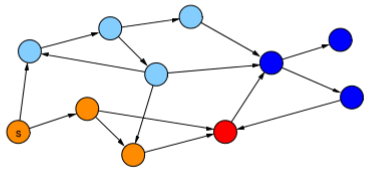


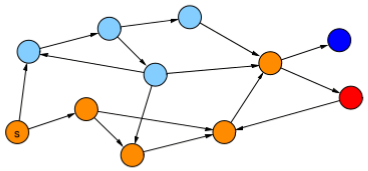


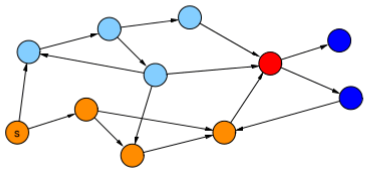


Tiefensuche

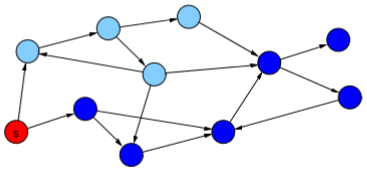




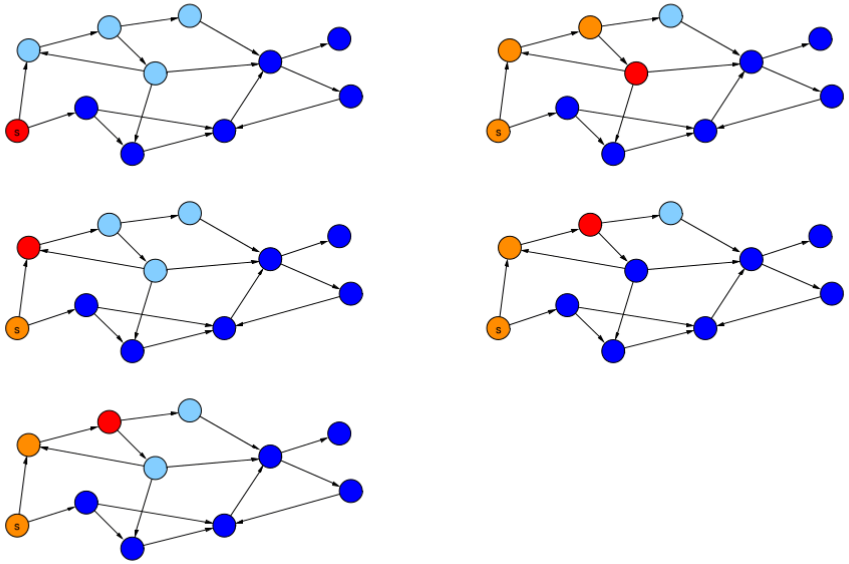




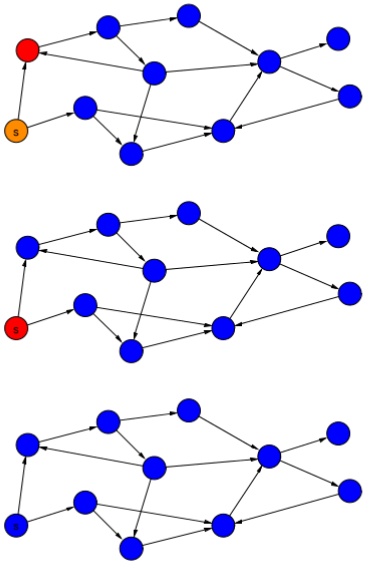
Tiefensuche



Tiefensuche



Tiefensuche



Tiefensuche

Variablen:

- int[ ] dfsNum; // Explorationsreihenfolge
- int[ ] finishNum; // Fertigstellungsreihenfolge
- int dfsCount, finishCount; // Zähler

Methoden:

- init() { dfsCount = 1; finishCount = 1; }
- root(Node s) { dfsNum[s] = dfsCount; dfsCount++; }
- traverseTreeEdge(Node v, Node w)
  { dfsNum[w] = dfsCount; dfsCount++; }
- traverseNonTreeEdge(Node v, Node w) { }
- backtrack(Node u, Node v)
  { finishNum[v] = finishCount; finishCount++; }

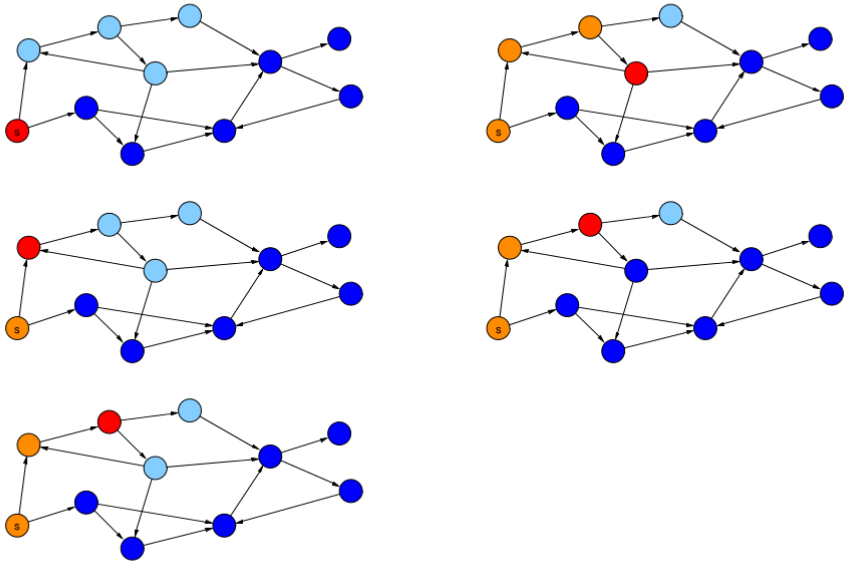
Tiefensuche

Übergeordnete Methode:

```
foreach (v ∈ V)
  Setze v auf nicht markiert;
init();
foreach (s ∈ V)
  if (s nicht markiert) {
    markiere s;
    root(s);
    DFS(s,s);
  }
```

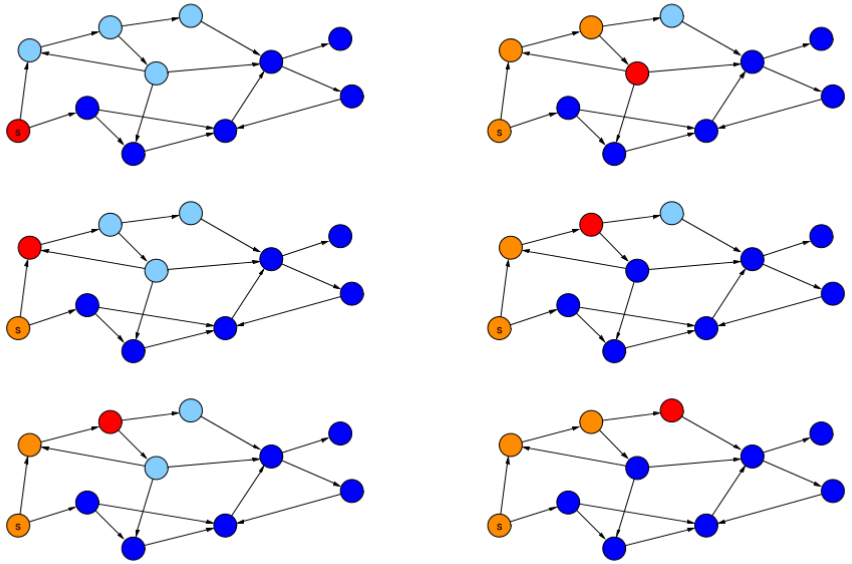
```
DFS(Node u, Node v) {
  foreach ((v, w) ∈ E)
    if (w ist markiert)
      traverseNonTreeEdge(v,w);
    else {
      traverseTreeEdge(v,w);
      markiere w;
      DFS(v,w);
    }
  backtrack(u,v);
}
```

Tiefensuche



Graphen Graphtraversierung

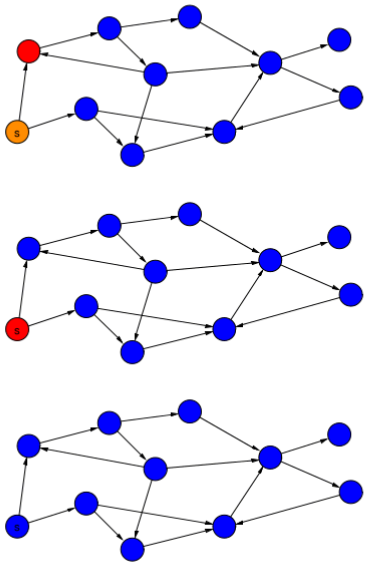
Tiefensuche



H. Täubig (TUM) GAD SS'14 429

Graphen Graphtraversierung

Tiefensuche



Graphen Graphtraversierung

Tiefensuche

Variablen:

- int[ ] dfsNum; // Explorationsreihenfolge
- int[ ] finishNum; // Fertigstellungsreihenfolge
- int dfsCount, finishCount; // Zähler

Methoden:

- init() { dfsCount = 1; finishCount = 1; }
- root(Node s) { dfsNum[s] = dfsCount; dfsCount++; }
- traverseTreeEdge(Node v, Node w)
  { dfsNum[w] = dfsCount; dfsCount++; }
- traverseNonTreeEdge(Node v, Node w) { }
- backtrack(Node u, Node v)
  { finishNum[v] = finishCount; finishCount++; }

Tiefensuche

Übergeordnete Methode:

```
foreach (v ∈ V)
   Setze v auf nicht markiert;
init();
foreach (s ∈ V)
   if (s nicht markiert) {
      markiere s;
      root(s);
      DFS(s,s);
   }
```

```
DFS(Node u, Node v) {
   foreach ((v, w) ∈ E)
      if (w ist markiert)
         traverseNonTreeEdge(v,w);
      else {
         traverseTreeEdge(v,w);
         markiere w;
         DFS(v,w);
      }
   backtrack(u,v);
}
```


Tiefensuche

Variablen:

- int[ ] dfsNum; // Explorationsreihenfolge
- int[ ] finishNum; // Fertigstellungsreihenfolge
- int dfsCount, finishCount; // Zähler

Methoden:

- init() { dfsCount = 1; finishCount = 1; }
- root(Node s) { dfsNum[s] = dfsCount; dfsCount++; }
- traverseTreeEdge(Node v, Node w)
  { dfsNum[w] = dfsCount; dfsCount++; }
- traverseNonTreeEdge(Node v, Node w) { }
- backtrack(Node u, Node v)
  { finishNum[v] = finishCount; finishCount++; }

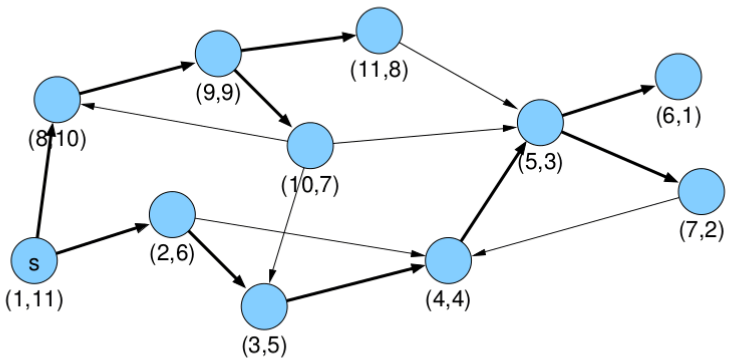
Tiefensuche

Übergeordnete Methode:

```
foreach (v ∈ V)
   Setze v auf nicht markiert;
init();
foreach (s ∈ V)
   if (s nicht markiert) {
      markiere s;
      root(s);
      DFS(s,s);
   }
```

```
DFS(Node u, Node v) {
   foreach ((v, w) ∈ E)
      if (w ist markiert)
         traverseNonTreeEdge(v,w);
      else {
         traverseTreeEdge(v,w);
         markiere w;
         DFS(v,w);
      }
   backtrack(u,v);
}
```


Tiefensuche



DFS-Nummerierung

Beobachtung:

- Knoten im DFS-Rekursionsstack (aktiven Knoten) sind bezüglich dfsNum aufsteigend sortiert

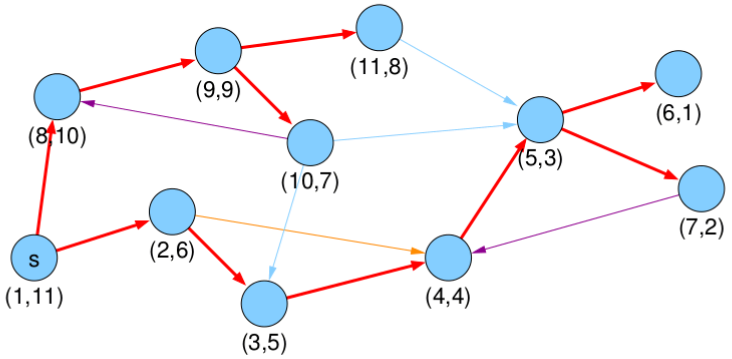
Begründung:

- dfsCount wird nach jeder Zuweisung von dfsNum inkrementiert
- neue aktive Knoten haben also immer die höchste dfsNum

DFS-Nummerierung

Kantentypen:

- Baumkanten: zum Kind
- Vorwärtskanten: zu einem Nachfahren
- Rückwärtskanten: zu einem Vorfahren
- Kreuzkanten: sonstige



DFS-Nummerierung

Beobachtung für Kante (v, w):

Kantentyp	dfsNum[v] < dfsNum[w]	finishNum[v] > finishNum[w]
Baum & Vorwärts	ja	ja
Rückwärts	nein	nein (umgekehrt)
Kreuz	nein	ja

DFS-Nummerierung

Beobachtung für Kante (v, w):

Kantentyp	dfsNum[v] < dfsNum[w]	finishNum[v] > finishNum[w]
Baum & Vorwärts	ja	ja
Rückwärts	nein	nein (umgekehrt)
Kreuz	nein	ja

DFS-Nummerierung

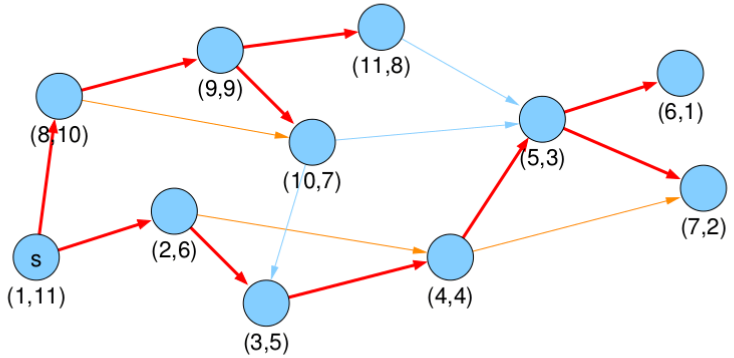
Beobachtung für Kante (v, w):

Kantentyp	dfsNum[v] < dfsNum[w]	finishNum[v] > finishNum[w]
Baum & Vorwärts	ja	ja
Rückwärts	nein	nein (umgekehrt)
Kreuz	nein	ja

DAG-Erkennung per DFS

Anwendung:

- Erkennung von azyklischen gerichteten Graphen (engl. directed acyclic graph / DAG)



- keine gerichteten Kreise

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v, w) \in E$: *finishNum[v] > finishNum[w]*

Beweis.

- (2)⇒(3): Wenn (2), dann gibt es nur Baum-, Vorwärts- und Kreuz-Kanten. Für alle gilt (3).
- (3)⇒(2): Für Rückwärtskanten gilt sogar die umgekehrte Relation finishNum[v]<finishNum[w].
  Wenn (3), dann kann es also keine Rückwärtskanten geben (2).

...

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v, w) \in E$: *finishNum[v] > finishNum[w]*

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v,w) \in E : finishNum[v] > finishNum[w]$

Beweis.

- $\neg(2) \Rightarrow \neg(1)$: Wenn Rückwärtskante (v,w) existiert, gibt es einen gerichteten Kreis ab Knoten w (und G ist kein DAG).
- $\neg(1) \Rightarrow \neg(2)$: Wenn es einen gerichteten Kreis gibt, ist mindestens eine von der DFS besuchte Kante dieses Kreises eine Rückwärtskante (Kante zu einem schon besuchten Knoten, dieser muss Vorfahr sein).

□

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v,w) \in E : finishNum[v] > finishNum[w]$

Beweis.

- $(2) \Rightarrow (3)$: Wenn (2), dann gibt es nur Baum-, Vorwärts- und Kreuz-Kanten. Für alle gilt (3).
- $(3) \Rightarrow (2)$: Für Rückwärtskanten gilt sogar die umgekehrte Relation finishNum[v]<finishNum[w].
  Wenn (3), dann kann es also keine Rückwärtskanten geben (2).

...

DFS-Nummerierung

Beobachtung für Kante (v,w):

Kantentyp	dfsNum[v] < dfsNum[w]	finishNum[v] > finishNum[w]
Baum & Vorwärts	ja	ja
Rückwärts	nein	nein (umgekehrt)
Kreuz	nein	ja

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v,w) \in E : finishNum[v] > finishNum[w]$

Beweis.

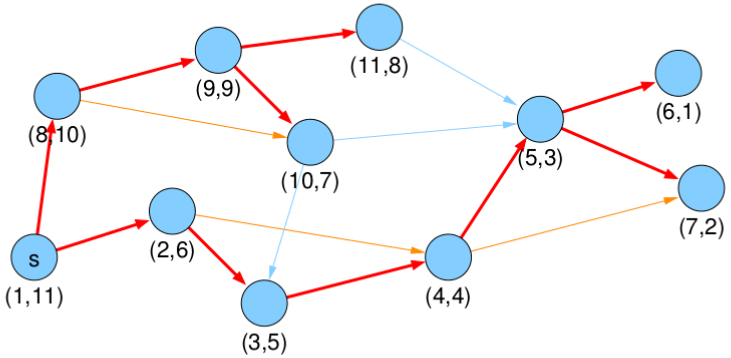
- $(2) \Rightarrow (3)$: Wenn (2), dann gibt es nur Baum-, Vorwärts- und Kreuz-Kanten. Für alle gilt (3).
- $(3) \Rightarrow (2)$: Für Rückwärtskanten gilt sogar die umgekehrte Relation finishNum[v]<finishNum[w].
  Wenn (3), dann kann es also keine Rückwärtskanten geben (2).

...

DAG-Erkennung per DFS

Anwendung:

- Erkennung von azyklischen gerichteten Graphen (engl. directed acyclic graph / DAG)



- keine gerichteten Kreise

Zusammenhang in Graphen

Definition

Ein ungerichteter Graph heißt zusammenhängend, wenn es von jedem Knoten einen Pfad zu jedem anderen Knoten gibt.

Ein maximaler zusammenhängender induzierter Teilgraph wird als Zusammenhangskomponente bezeichnet.

Die Zusammenhangskomponenten eines ungerichteten Graphen können mit DFS oder BFS in $\mathcal{O}(n+m)$ bestimmt werden.

Knoten-Zusammenhang

Definition

Ein ungerichteter Graph $G = (V, E)$ heißt k-fach zusammenhängend (oder genauer gesagt k-*knoten*zusammenhängend), falls

- $|V| > k$ und
- für jede echte Knotenteilmenge $X \subset V$ mit $|X| < k$ der Graph $G - X$ zusammenhängend ist.

Knoten-Zusammenhang

Definition

Ein ungerichteter Graph $G = (V, E)$ heißt k-fach zusammenhängend (oder genauer gesagt k-*knoten*zusammenhängend), falls

- $|V| > k$ und
- für jede echte Knotenteilmenge $X \subset V$ mit $|X| < k$ der Graph $G - X$ zusammenhängend ist.

Bemerkung:

- "zusammenhängend" ist im wesentlichen gleichbedeutend mit "1-knotenzusammenhängend"

  Ausnahme: Graph mit nur einem Knoten ist zusammenhängend, aber nicht 1-zusammenhängend

Artikulationsknoten und Blöcke

Definition

Ein Knoten v eines Graphen G heißt Artikulationsknoten (engl. *cut-vertex*), wenn sich die Anzahl der Zusammenhangskomponenten von G durch das Entfernen von v erhöht.

Artikulationsknoten und Blöcke

Definition

Ein Knoten v eines Graphen G heißt Artikulationsknoten (engl. *cut-vertex*), wenn sich die Anzahl der Zusammenhangskomponenten von G durch das Entfernen von v erhöht.

Definition

Die Zweifachzusammenhangskomponenten eines Graphen sind die maximalen Teilgraphen, die 2-fach zusammenhängend sind.

Ein Block ist ein maximaler zusammenhängender Teilgraph, der keinen Artikulationsknoten enthält.

Die Menge der Blöcke besteht aus den Zweifachzusammenhangskomponenten, den Brücken (engl. *cut edges*), sowie den isolierten Knoten.

Blöcke und DFS

Modifizierte DFS nach R. E. Tarjan:

- `num[v]`: DFS-Nummer von v
- `low[v]`: minimale Nummer `num[w]` eines Knotens w, der von v aus über beliebig viele (≥ 0) Baumkanten (abwärts), evt. gefolgt von einer einzigen Rückwärtskante (aufwärts) erreicht werden kann

- `low[v]`: Minimum von
  - `num[v]`
  - `low[w]`, wobei w ein Kind von v im DFS-Baum ist (Baumkante)
  - `num[w]`, wobei $\{v, w\}$ eine Rückwärtskante ist

Artikulationsknoten und DFS

Lemma

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph und T ein DFS-Baum in G.

Ein Knoten $a \in V$ ist genau dann ein Artikulationsknoten, wenn

- *a die Wurzel von T ist und mindestens 2 Kinder hat, oder*
- *a nicht die Wurzel von T ist und es ein Kind b von a mit `low[b]`$\geq$`num[a]` gibt.*

Beweisidee

Der Algorithmus beruht auf der Tatsache, dass in Zweifach(knoten)zusammenhangskomponenten zwischen jedem Knotenpaar mindestens zwei (knoten-)disjunkte Wege existieren. Das entspricht einem Kreis.

Artikulationsknoten und DFS

Lemma

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph und T ein DFS-Baum in G.

Ein Knoten $a \in V$ ist genau dann ein Artikulationsknoten, wenn

- *a die Wurzel von T ist und mindestens 2 Kinder hat, oder*
- *a nicht die Wurzel von T ist und es ein Kind b von a mit $\texttt{low[b]} \geq \texttt{num[a]}$ gibt.*

Beweisidee

Der Algorithmus beruht auf der Tatsache, dass in Zweifach(knoten)zusammenhangskomponenten zwischen jedem Knotenpaar mindestens zwei (knoten-)disjunkte Wege existieren. Das entspricht einem Kreis.

Graphen | Graphtraversierung

Artikulationsknoten und DFS

Lemma

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph und T ein DFS-Baum in G.

Ein Knoten $a \in V$ ist genau dann ein Artikulationsknoten, wenn

- *a die Wurzel von T ist und mindestens 2 Kinder hat, oder*
- *a nicht die Wurzel von T ist und es ein Kind b von a mit $\texttt{low[b]} \geq \texttt{num[a]}$ gibt.*

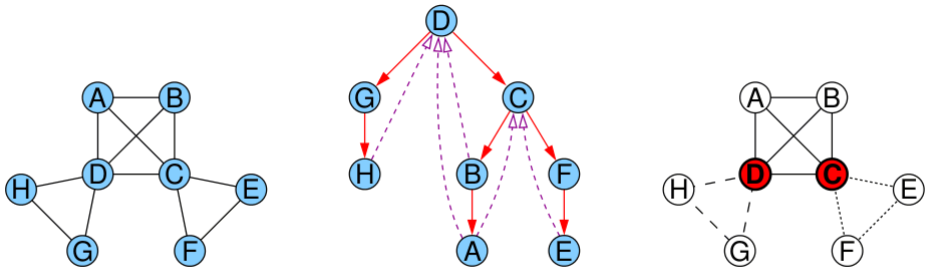
Beweisidee

Der Algorithmus beruht auf der Tatsache, dass in Zweifach(knoten)zusammenhangskomponenten zwischen jedem Knotenpaar mindestens zwei (knoten-)disjunkte Wege existieren. Das entspricht einem Kreis.

H. Täubig (TUM) | GAD | SS'14 | 444

Artikulationsknoten und Blöcke per DFS

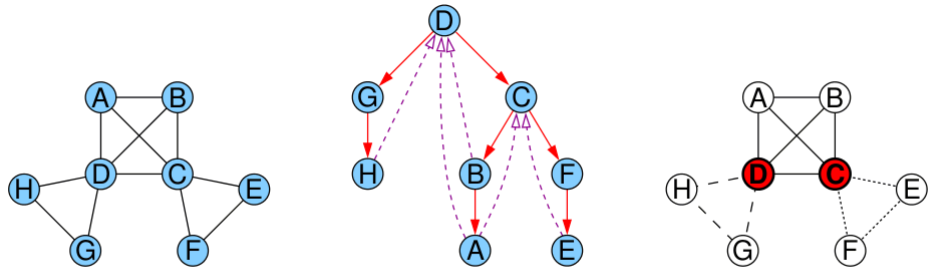
- bei Aufruf der DFS für Knoten v wird $\texttt{num}[v]$ bestimmt und $\texttt{low}[v]$ mit $\texttt{num}[v]$ initialisiert
- nach Besuch eines Nachbarknotens w:
  Update von $\texttt{low}[v]$ durch Vergleich mit
  - $\texttt{low}[w]$ nach Rückkehr vom rekursiven Aufruf, falls (v, w) eine Baumkante war
  - $\texttt{num}[w]$, falls (v, w) eine Rückwärtskante war



Graphen | Graphtraversierung

Artikulationsknoten und Blöcke per DFS

- Kanten werden auf einem anfangs leeren Stack gesammelt
- Rückwärtskanten kommen direkt auf den Stack (ohne rek. Aufruf)
- Baumkanten kommen vor dem rekursiven Aufruf auf den Stack
- nach Rückkehr von einem rekursiven Aufruf werden im Fall $\texttt{low}[w] \geq \texttt{num}[v]$ die obersten Kanten vom Stack bis einschließlich der Baumkante $\{v, w\}$ entfernt und bilden den nächsten Block



H. Täubig (TUM) | GAD | SS'14 | 446