Fortgeschrittene Netzwerk- und Graph-Algorithmen

Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Wintersemester 2010/11

Übersicht

1. Grundlagen
   - Wiederholung bekannter Algorithmen

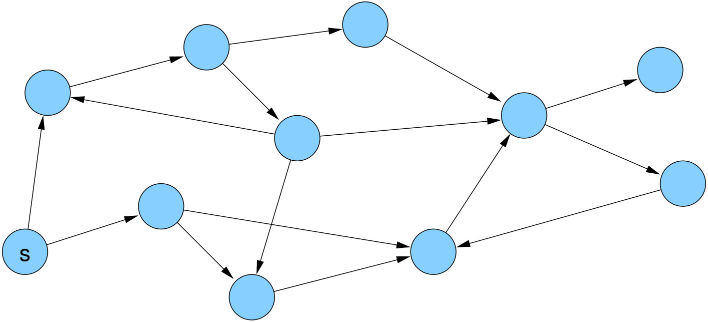
Graphtraversierung

Problem:

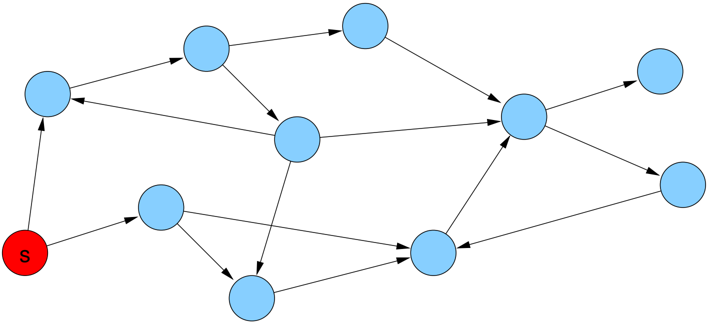
Wie kann man die Knoten eines Graphen systematisch durchlaufen?

- Breitensuche (breadth-first search, bfs)
  - schichtenweise Traversierung in aufsteigendem Abstand vom Ursprungsknoten
- Tiefensuche (depth-first search, dfs)
  - Topologische Sortierung (bei DAGs)
  - Zweifachzusammenhangskomponenten
  - Starke Zusammenhangskomponenten

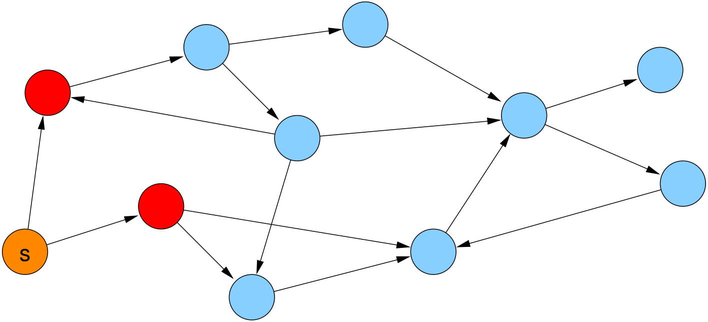
Breitensuche



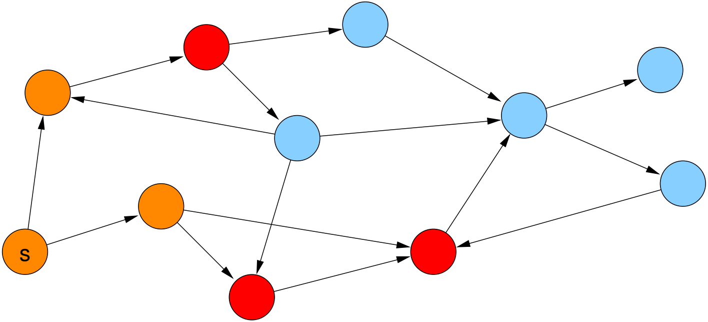
Breitensuche



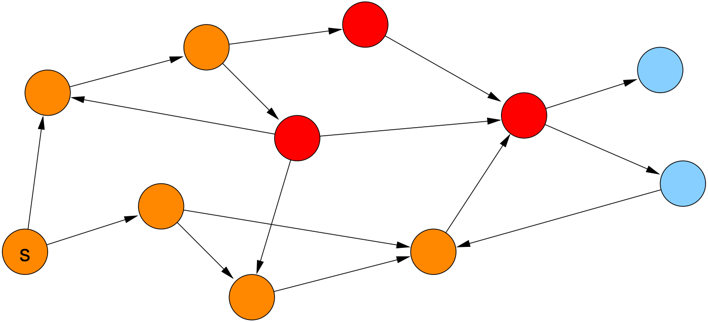
Breitensuche



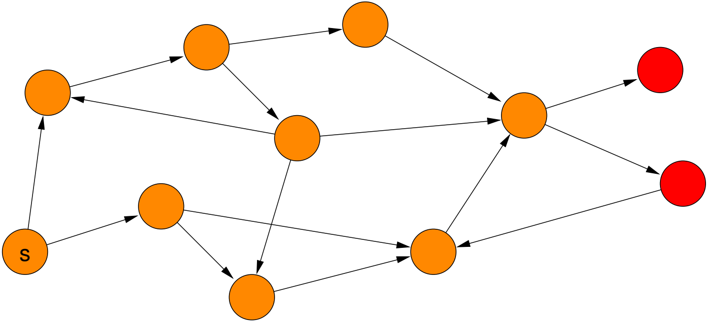
Breitensuche



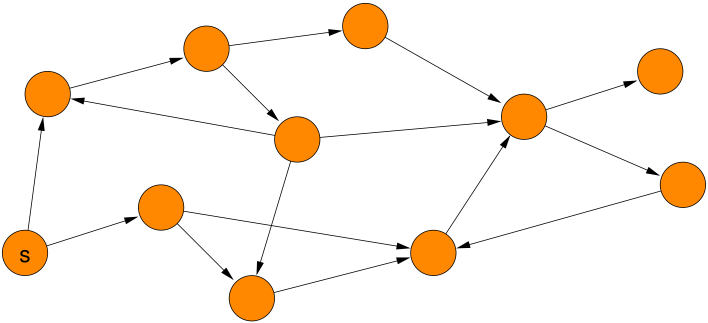
Breitensuche



Breitensuche

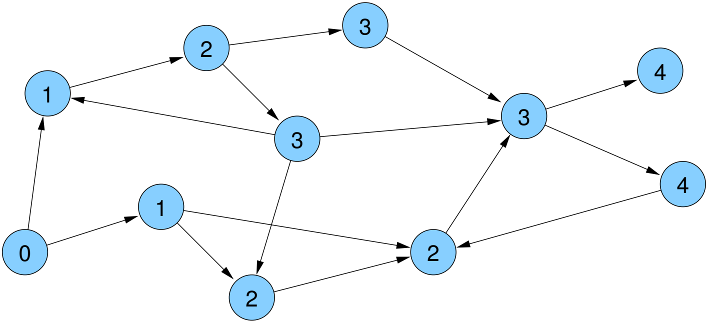


Breitensuche



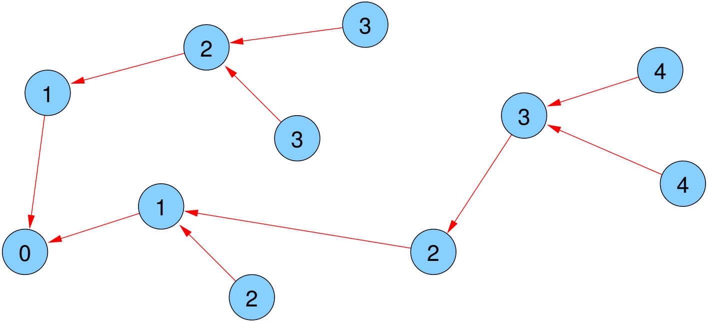
Breitensuche

- Anwendung: Single Source Shortest Path (SSSP) Problem in ungewichteten Graphen
- $d(v)$: Distanz von Knoten v zu s ($d(s) = 0$)



Breitensuche

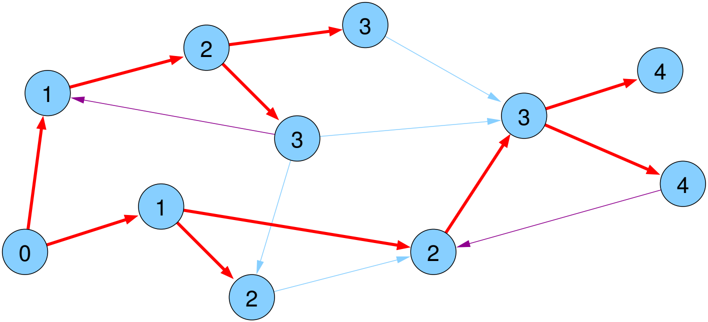
- parent(v): Knoten, von dem v entdeckt wurde
- parent wird beim ersten Besuch von v gesetzt ($\Rightarrow$ eindeutig)



Breitensuche

Kantentypen:

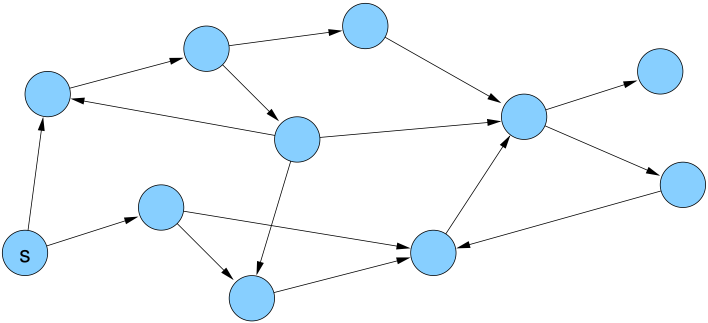
- Baumkanten: zum Kind
- Rückwärtskanten: zu einem Vorfahren
- Kreuzkanten: sonstige



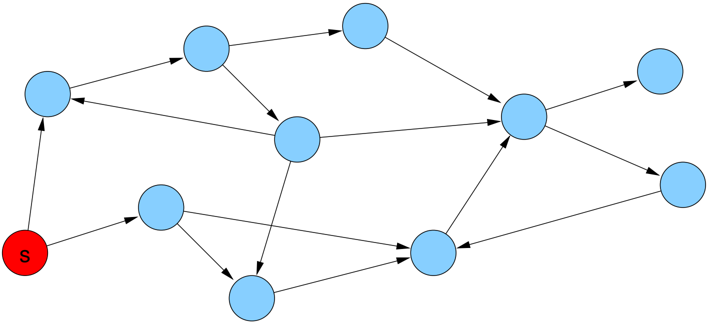
Breitensuche

```
void 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;
            }
        }
    }
}
```

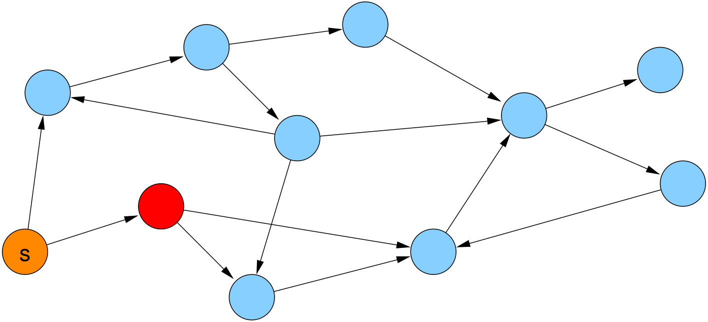
Tiefensuche



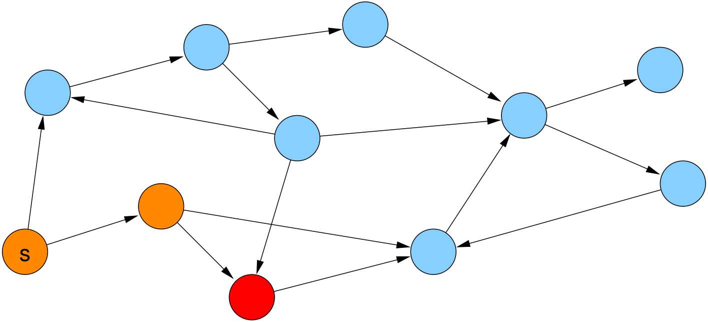
Tiefensuche



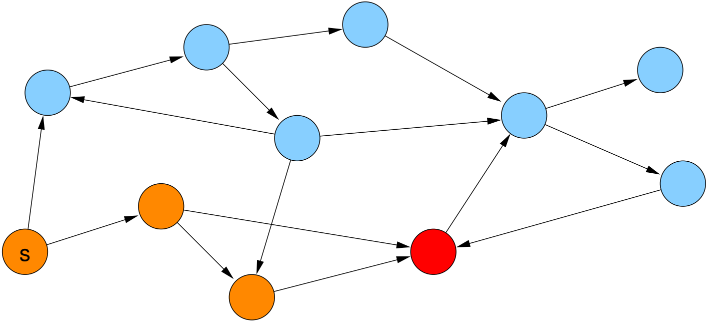
Tiefensuche



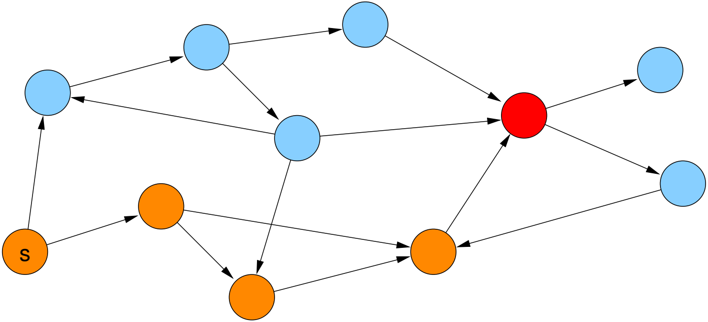
Tiefensuche



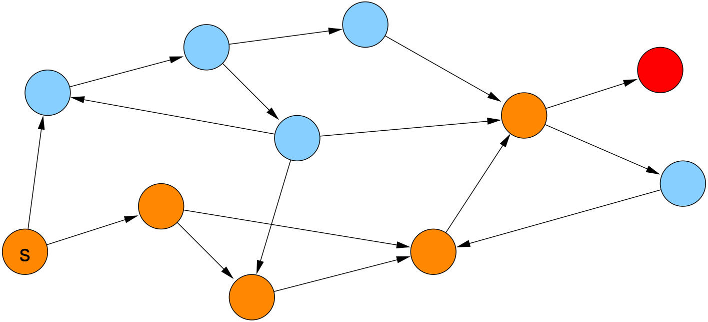
Tiefensuche



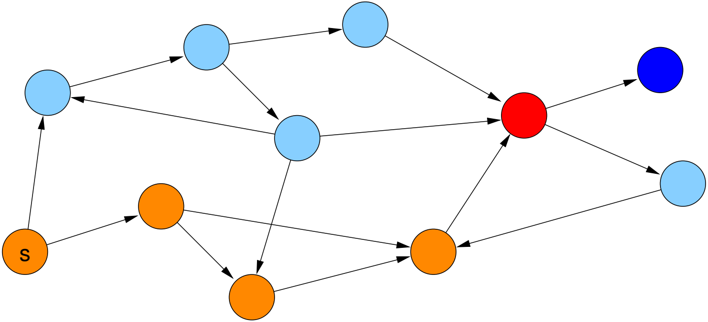
Tiefensuche



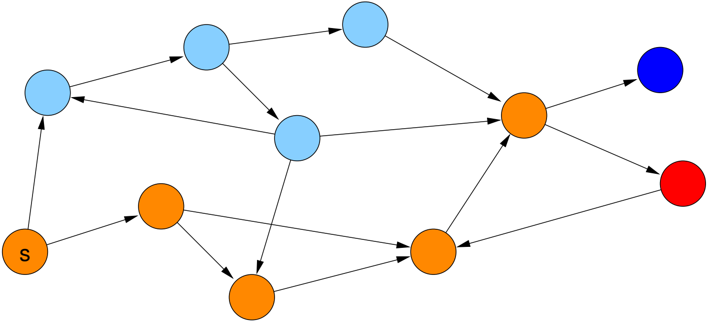
Tiefensuche



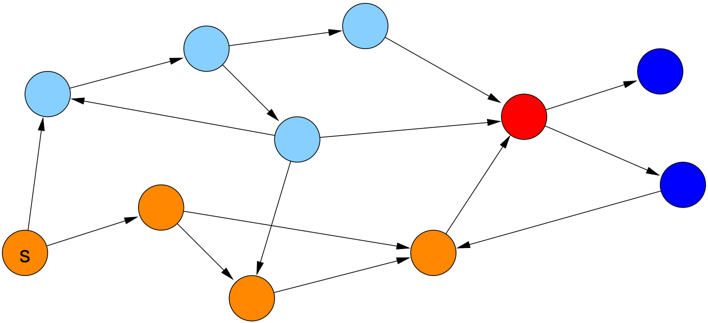
Tiefensuche



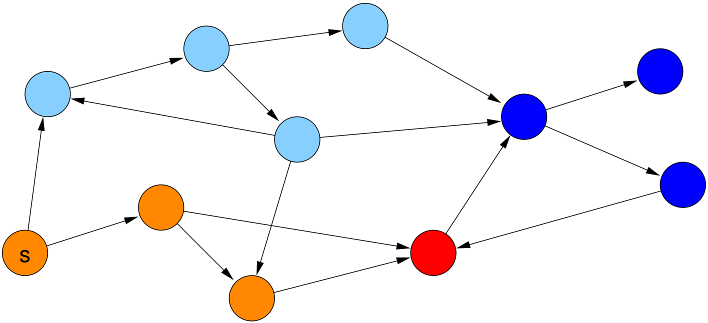
Tiefensuche



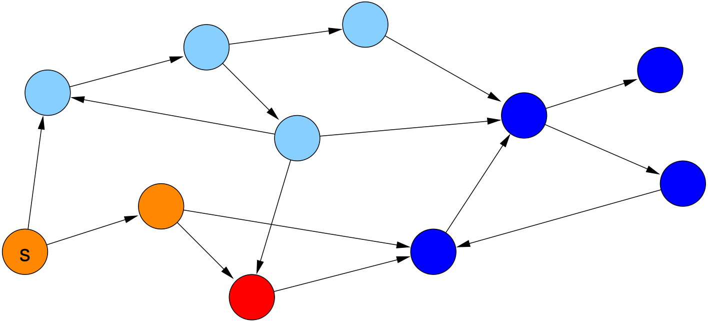
Tiefensuche



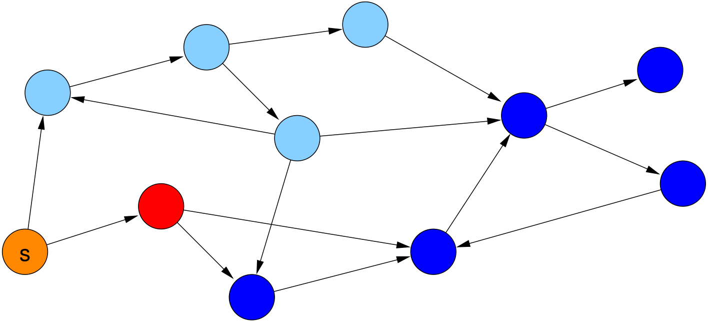
Tiefensuche



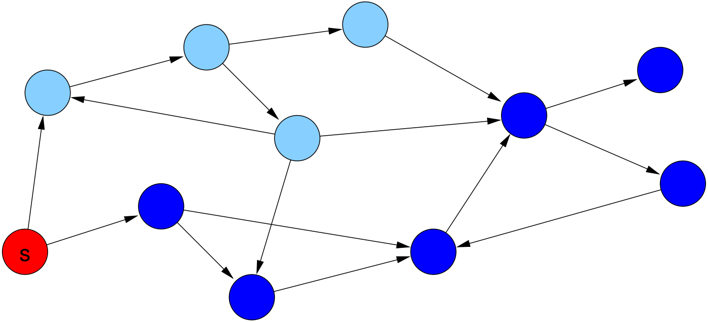
Tiefensuche



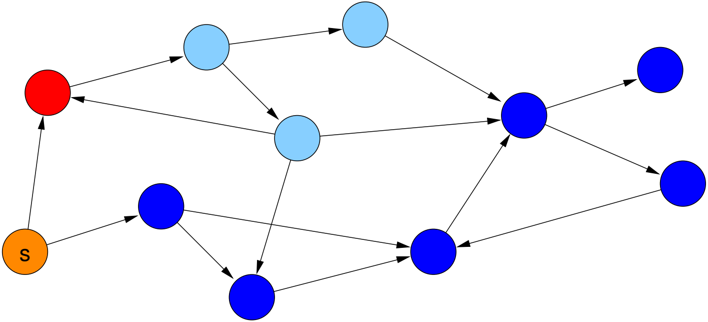
Tiefensuche



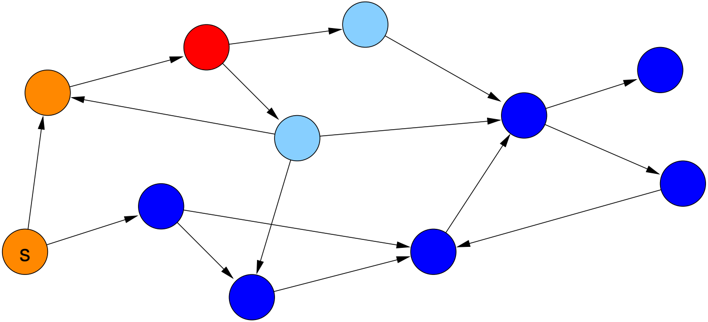
Tiefensuche



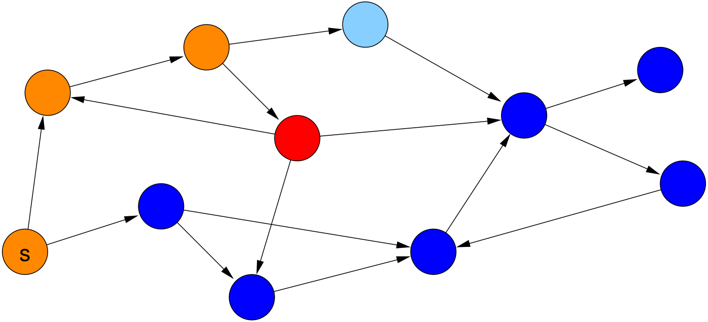
Tiefensuche



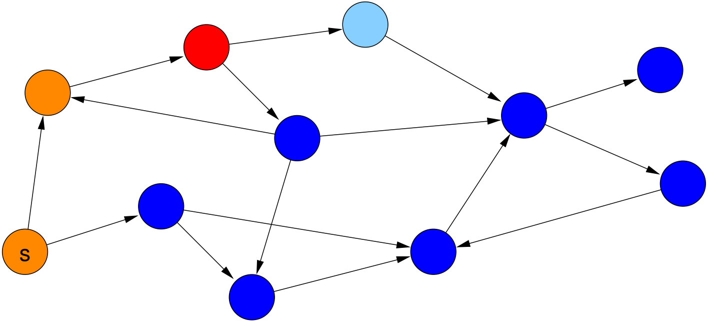
Tiefensuche



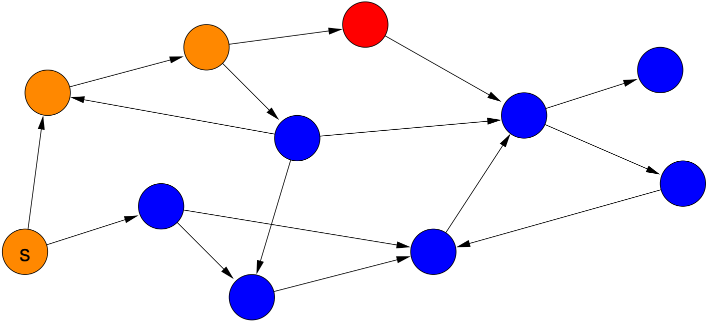
Tiefensuche



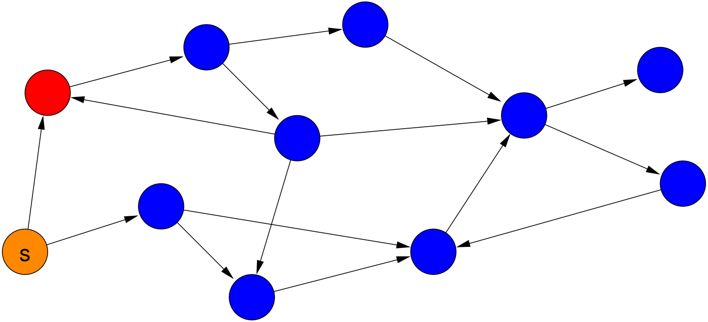
Tiefensuche



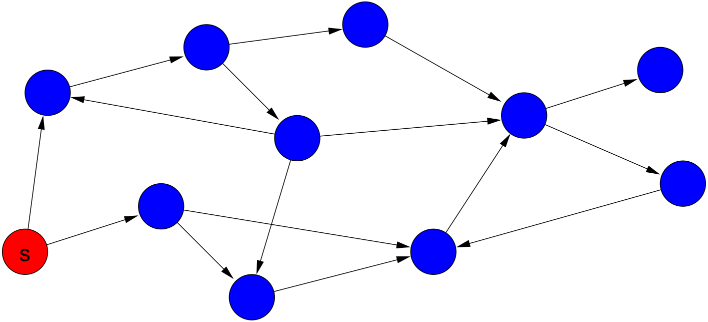
Tiefensuche



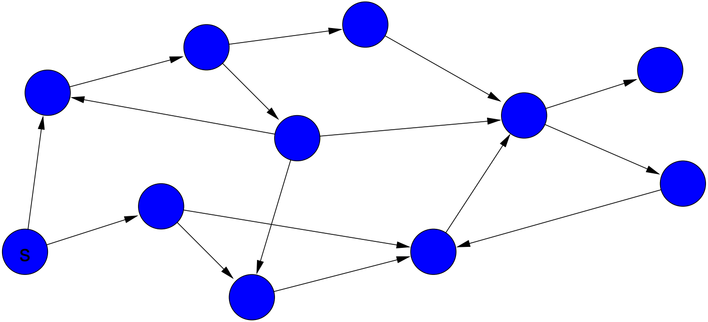
Tiefensuche



Tiefensuche



Tiefensuche



Tiefensuche

Übergeordnete Methode (falls nicht alle Knoten erreicht werden)

```
alle Knoten nicht markiert;
init();
foreach (s ∈ V)
  if (s nicht markiert) {
    markiere s;
    root(s);
    DFS(s,s);
  }
```

Tiefensuche

```
void DFS(Node u, Node v) {
   foreach ((v, w) ∈ E)
      if (is_marked(w))
         traverseNonTreeEdge(v,w);
      else {
         traverseTreeEdge(v,w);
         mark(w);
         DFS(v,w);
      }
   backtrack(u,v);
}
```

Tiefensuche

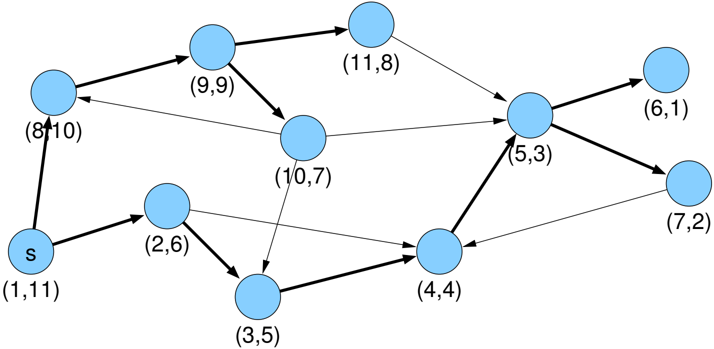
Variablen:

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

Methoden:

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

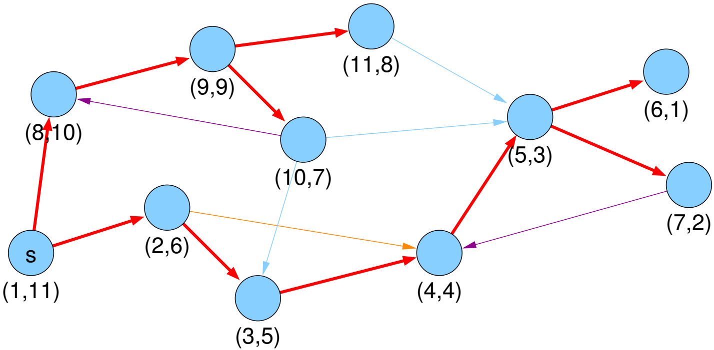
Tiefensuche



DFS-Nummerierung

Kantentypen:

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



DFS-Nummerierung

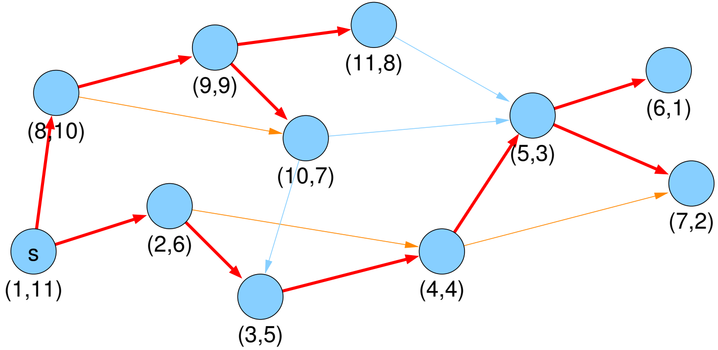
Beobachtung für Kante (v, w):

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

DFS-Nummerierung

Anwendung:

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



- keine gerichteten Kreise

DFS-Nummerierung

Lemma

Folgende Aussagen sind äquivalent:

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

Beweis.

- (2)⇒(3): wenn (2), dann gibt es nur Baum-, Vorwärts- und Kreuzkanten
  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)

...

DFS-Nummerierung

Lemma

Folgende Aussagen sind äquivalent:

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

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)

$\square$

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-Zweifachzusammenhang

Definition

Ein ungerichteter Graph $G = (V, E)$ heißt 2-fach zusammenhängend (oder genauer gesagt 2-*knoten*zusammenhängend), falls

- $|V| > 2$ und
- für jeden Knoten $v \in V$ der Graph $G - \{v\}$ zusammenhängend ist.

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. D.h. 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 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

Blöcke und DFS

Lemma

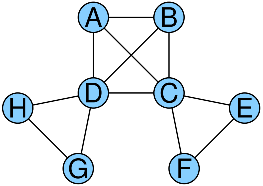
Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph und T ein DFS-Baum in G.

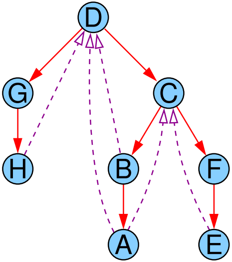
Ein Knoten $a \in V$ ist genau dann ein Artikulationsknoten, wenn

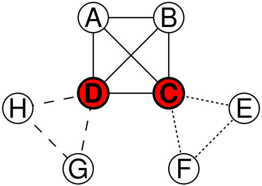
- *a die Wurzel von T ist und mindestens 2 Kinder hat, oder*
- *a nicht die Wurzel von T ist und*
  es ein Kind b von a mit `low[b]`$\geq$`num[a]` gibt.

Blöcke und DFS

Die Kanten werden auf einem Stack gesammelt und nach der Erkennung eines Artikulationsknotens wird der gesamte Block abgepflückt.







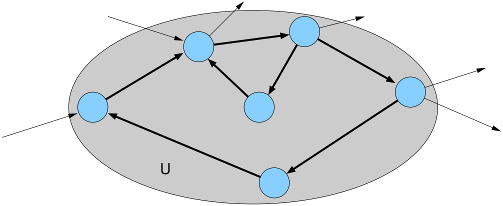
Starke Zusammenhangskomponenten

Definition

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

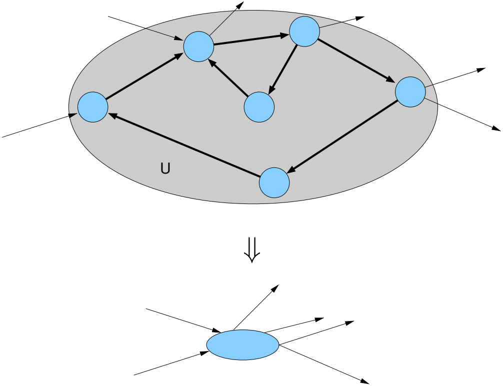
$U \subseteq V$ heißt stark zusammenhängend genau dann, wenn für alle $u, v \in U$ ein gerichteter Pfad von u nach v in G existiert

$U \subseteq V$ heißt starke Zusammenhangskomponente (ZHK) von G, wenn U stark zusammenhängend und (inklusions-)maximal ist



Starke Zusammenhangskomponenten

Beobachtung:
Schrumpft man alle starken Zusammenhangskomponenten zu einzelnen Knoten, ergibt sich ein DAG.



Starke Zhk. und DFS / Variante 1

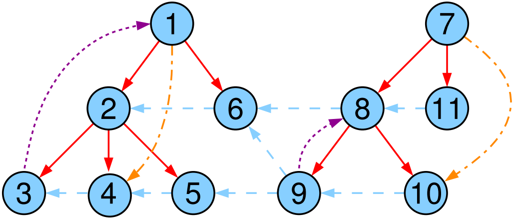
Modifizierte DFS nach R. E. Tarjan:

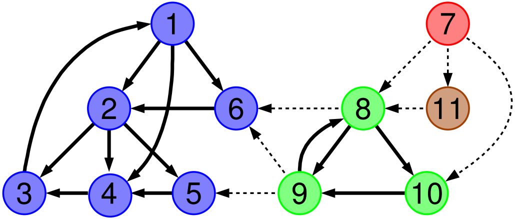
- `num[`v`]`: DFS-Nummer von v
- `low[`v`]`: minimale Nummer `num[`w`]` eines Knotens w, der von v aus über beliebig viele (≥ 0) Baumkanten abwärts, evt. gefolgt von einer einzigen Rückwärtskante oder einer Querkante zu einer ZHK, deren Wurzel echter Vorfahre von v ist, erreicht werden kann

- `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
  - `num[`w`]`, wobei $\{v, w\}$ eine Querkante ist und die Wurzel der starken Zusammenhangskomponente von w ist Vorfahre von v

Starke Zusammenhangskomponenten

- Knoten v ist genau dann Wurzel einer starken Zusammenhangskomponente, wenn `num[`v`]`=`low[`v`]`





Starke Zhk. und DFS / Variante 2

Prinzip:

- Unterscheidung in fertige / unfertige Knoten
- Zusammenhangskomponente heißt geschlossen, falls sie nur fertige Knoten enthält, ansonsten heißt sie offen
- Knoten in geschlossenen Komponenten heißen geschlossen, sonst offen
- Repräsentant einer ZHK ist der Knoten mit der kleinsten dfsNum

Starke Zhk. und DFS / Variante 2

Beobachtungen (Invarianten):

- geschlossene Knoten sind immer fertig,
  offene Knoten können fertig oder (noch) aktiv sein
- Kanten von geschlossenen Knoten führen immer zu geschlossenen Knoten
- Der Pfad vom Startknoten zum aktuellen Knoten enthält die Repräsentanten aller offenen ZHKs.
- Betrachtet man die Knoten in offenen ZHKs sortiert nach DFS-Nummern, so partitionieren die Repräsentanten diese Folge in die offenen ZHKs.

Starke Zhk. und DFS / Variante 2

Prinzip: betrachte Kante $e = (v, w)$

- Kante zu schon bekanntem Knoten w in offener ZHK (Rückwärts-/Querkante):
  falls v und w momentan noch in unterschiedlichen ZHKs liegen, müssen diese zusammen mit allen ZHKs dazwischen zu einer einzigen ZHK verschmolzen werden

  bei Vorwärtskanten ist nichts zu tun
- Kante zu Knoten w in geschlossener ZHK (Querkante):
  von w gibt es keinen Weg zu v, sonst wäre die ZHK von w noch nicht geschlossen, also bleiben die ZHKs unverändert
- Kante zu unbekanntem Knoten w (Baumkante):
  neue ZHK für w

Wenn Knoten keine ausgehenden Kanten mehr hat:

- Knoten fertig
- wenn Knoten Repräsentant seiner ZHK ist, dann ZHK schließen

Starke Zhk. und DFS / Variante 2

2. Variante:

- Verwaltung der unfertigen Knoten in Stack oNodes (in Reihenfolge steigender dfsNum)
- Verwaltung der Repräsentanten der offenen ZHKs in Stack oReps

Starke Zhk. und DFS / Variante 2

- void init() {
  component = new int[n];
  oReps = ⟨ ⟩;
  oNodes = ⟨ ⟩;
  dfsCount = 1;
  }

- void root(Node w) / traverseTreeEdge(Node v, Node w) {
  oReps.push(w); // Repräsentant einer neuen ZHK
  oNodes.push(w); // neuer offener Knoten
  dfsNum[w] = dfsCount;
  dfsCount++;
  }

Starke Zhk. und DFS / Variante 2

- void traverseNonTreeEdge(Node v, Node w) {
  if (w $\in$ oNodes)   // verschmelze ZHKs
  while (dfsNum[w] < dfsNum[oReps.top()])
  oReps.pop();
  }

- void backtrack(Node u, Node v) {
  if (v == oReps.top()) {   // v Repräsentant?
  oReps.pop();   // ja: entferne v
  do {   // und offene Knoten bis v
  w = oNodes.pop();
  component[w] = v;
  } while (w!=v);
  }
  }

Starke Zhk. und DFS / Variante 2

Zeit: $\mathcal{O}(n+m)$

Begründung:

- init, root: $\mathcal{O}(1)$
- traverseTreeEdge: $(n-1) \times \mathcal{O}(1)$
- backtrack, traverseNonTreeEdge:
  da jeder Knoten höchstens einmal in oReps und oNodes landet,
  insgesamt $\mathcal{O}(n+m)$
- DFS-Gerüst: $\mathcal{O}(n+m)$
- gesamt: $\mathcal{O}(n+m)$