Script generated by TTT

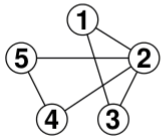
Title: TÃ¤ubig: GAD (27.06.2013)

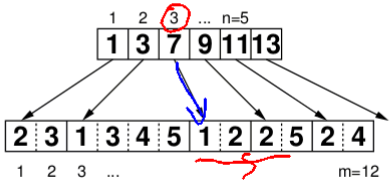
Date: Thu Jun 27 12:01:25 CEST 2013

Duration: 44:45 min

Pages: 27

Adjazenzarray





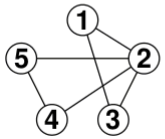
Vorteil:

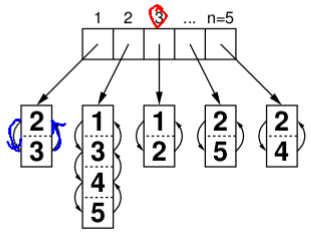
- Speicherbedarf:
  gerichtete Graphen: $n + m + \Theta(1)$
  (hier noch kompakter als Kantenliste mit $2m$)
  ungerichtete Graphen: $n + 2m + \Theta(1)$

Nachteil:

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

Adjazenzliste





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)

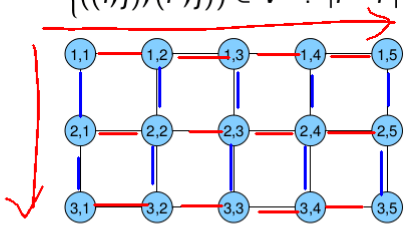
Speicheraufwand: $O(n + m)$

Implizite Repräsentation

Beispiel: Gitter-Graph (grid graph)

- definiert durch zwei Parameter k und ℓ

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



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

Übersicht

Graphtraversierung

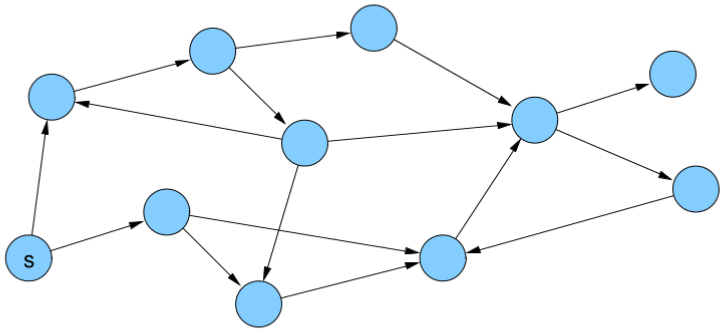
Problem:

Wie kann man die Knoten eines Graphen systematisch durchlaufen?

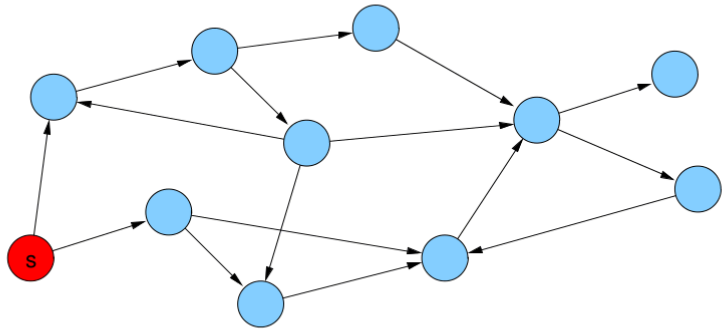
Grundlegende Strategien:

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

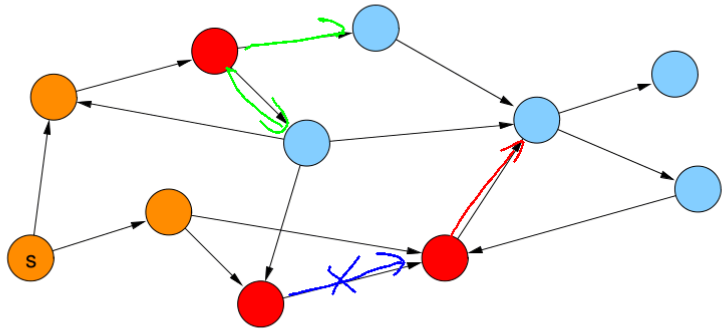
Breitensuche



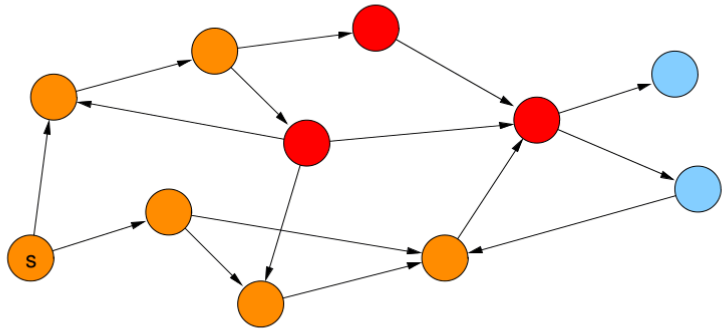
Breitensuche



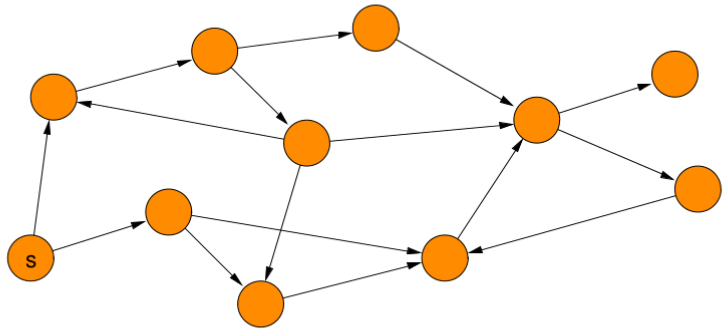
Breitensuche



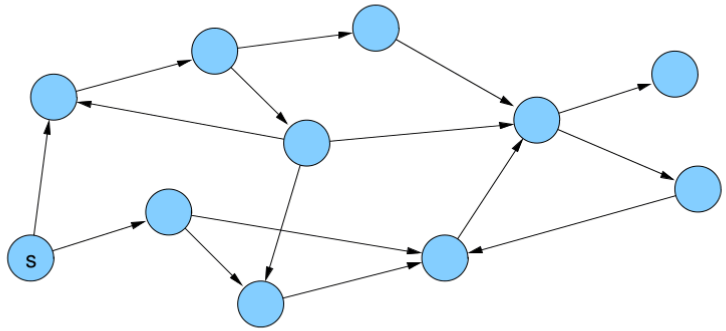
Breitensuche



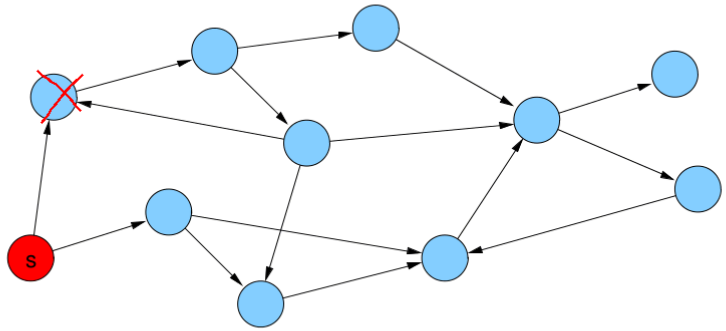
Breitensuche



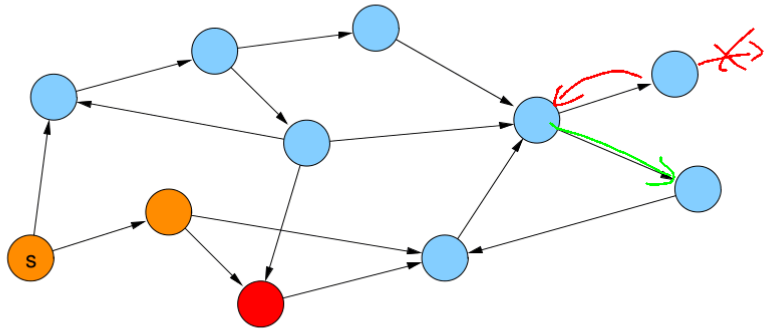
Tiefensuche



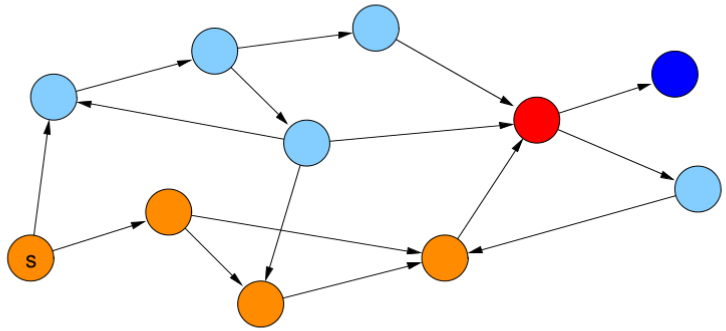
Tiefensuche



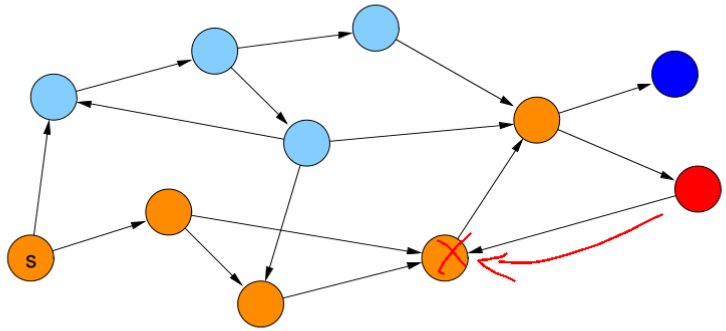
Tiefensuche



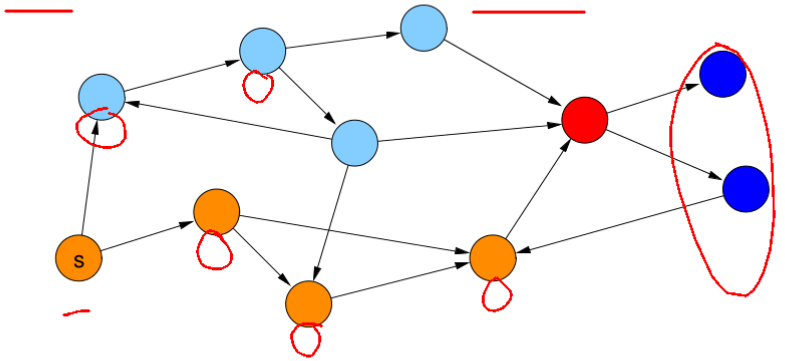
Tiefensuche



Tiefensuche

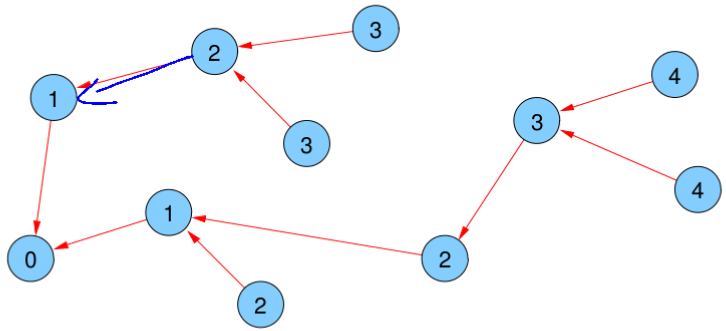


Tiefensuche



Breitensuche

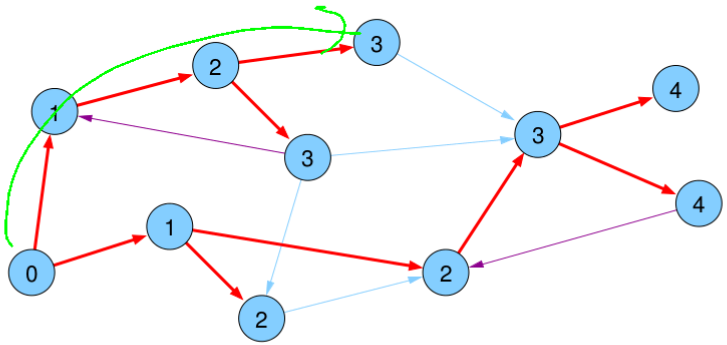
- parent(v): Knoten, von dem v entdeckt wurde
- parent wird beim ersten Besuch von v gesetzt ($\Rightarrow$ eindeutig)



Breitensuche

Kantentypen:

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

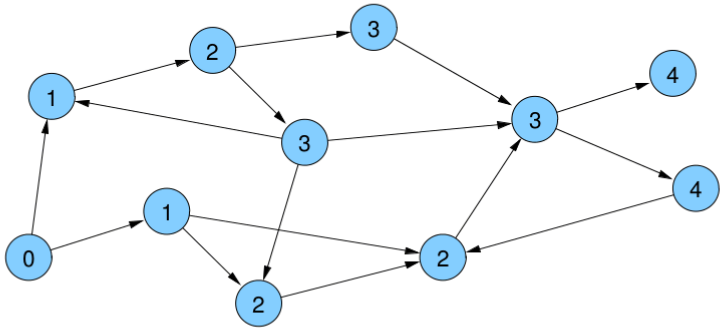


Breitensuche

```
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;
      }
    }
  }
}
```


Breitensuche

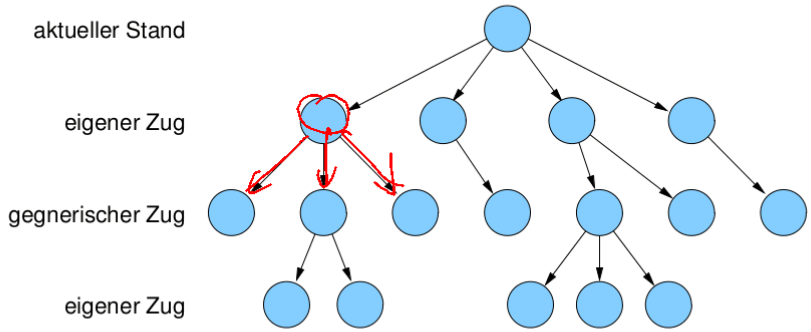
Anwendung: Single Source Shortest Path (SSSP) Problem in ungewichteten Graphen



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)

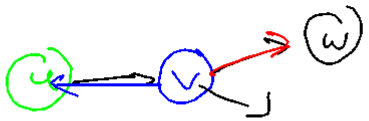
  Priorität eines Knotens wird durch eine Güte-Heuristik des repräsentierten Spielzustands gegeben

Tiefensuche

Übergeordnete Methode (falls nicht alle Knoten erreicht werden)

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


Tiefensuche




```
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