

**Script** generated by TTT

Title: Seidl: GAD (23.06.2015)

Date: Tue Jun 23 13:45:56 CEST 2015

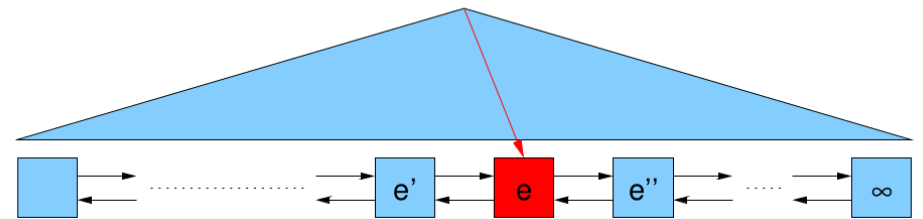
Duration: 150:10 min

Pages: 79

## (a,b)-Baum

remove(k)

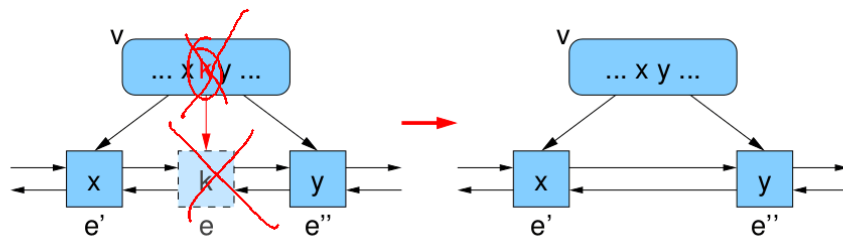
- Abstieg wie bei locate(k) bis Element e in Liste erreicht
- falls  $\text{key}(e) = k$ , entferne e aus Liste (sonst return)



## (a,b)-Baum

remove(k)

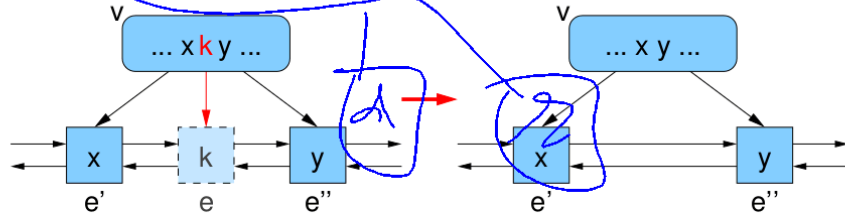
- entferne Handle auf e und Schlüssel k vom Baumknoten v über e (wenn e rechtestes Kind: Schlüsselvertauschung wie bei binärem Suchbaum)
- falls  $d(v) \geq a$ , dann fertig



### (a,b)-Baum

remove(k)

- entferne Handle auf e und Schlüssel k vom Baumknoten v über e (wenn e rechtestes Kind: Schlüsselvertauschung wie bei binärem Suchbaum)
- falls  $d(v) \geq a$ , dann fertig

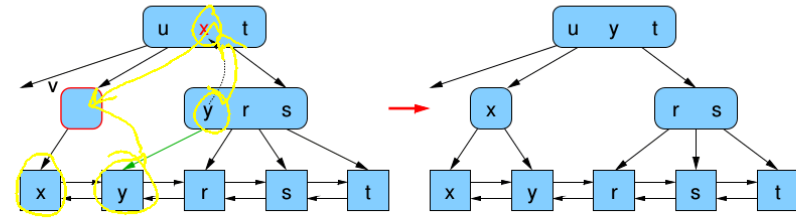


### (a,b)-Baum

remove(k)

- falls  $d(v) < a$  und ein direkter Nachbar  $v'$  von v hat Grad  $> a$ , nimm Kante von  $v'$

Beispiel: (2,4)-Baum

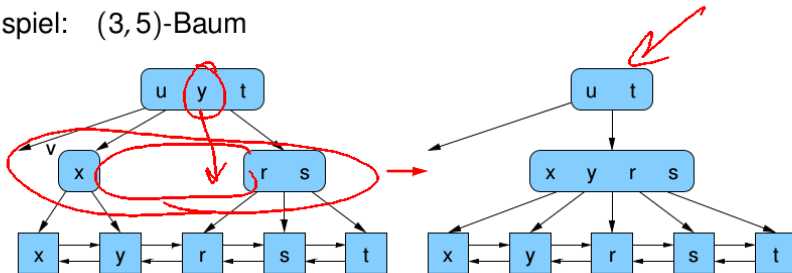


### (a,b)-Baum

remove(k)

- falls  $d(v) < a$  und kein direkter Nachbar von v hat Grad  $> a$ , merge v mit Nachbarn

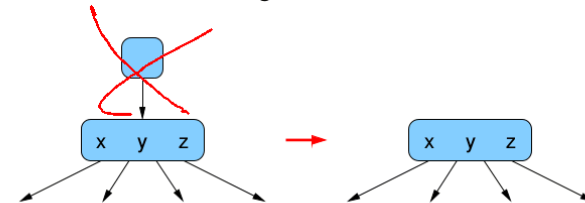
Beispiel: (3,5)-Baum



### (a,b)-Baum

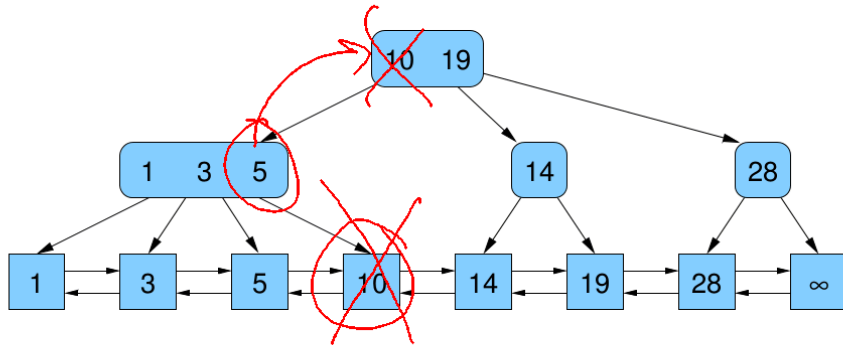
remove(k)

- Verschmelzungen können sich nach oben fortsetzen, ggf. bis zur Wurzel
- falls Grad der Wurzel  $< 2$ : entferne Wurzel neue Wurzel wird das einzige Kind der alten Wurzel

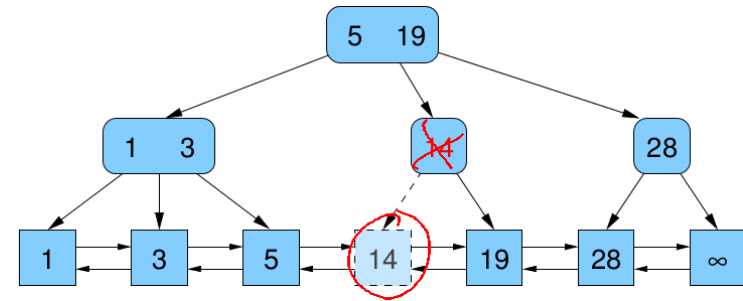


$(a, b)$ -Baum / remove $a = 2, b = 4$ 

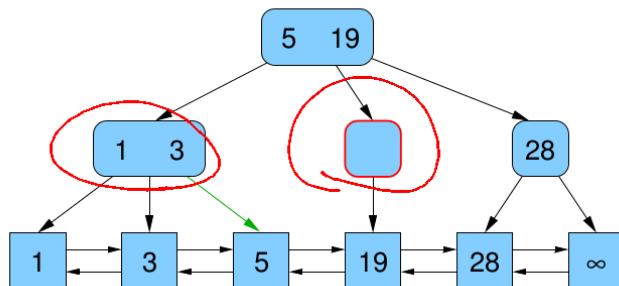
remove(10)

 $(a, b)$ -Baum / remove $a = 2, b = 4$ 

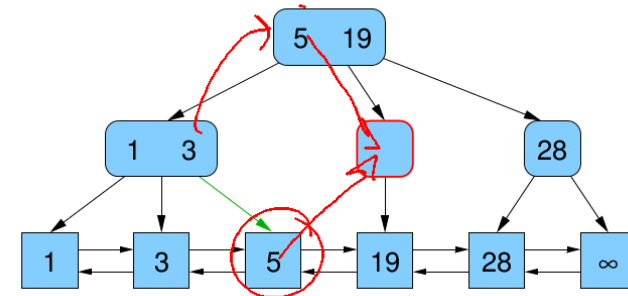
remove(14)

 $(a, b)$ -Baum / remove $a = 2, b = 4$ 

remove(14)

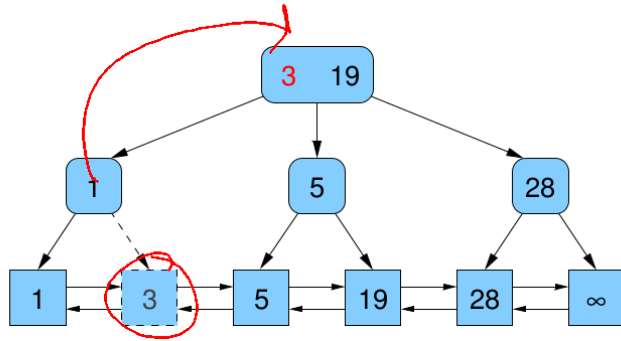
 $(a, b)$ -Baum / remove $a = 2, b = 4$ 

remove(14)

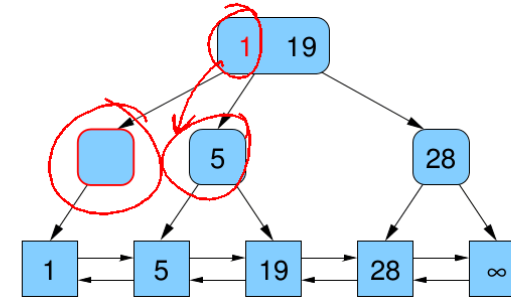


$(a, b)$ -Baum / remove $a = 2, b = 4$ 

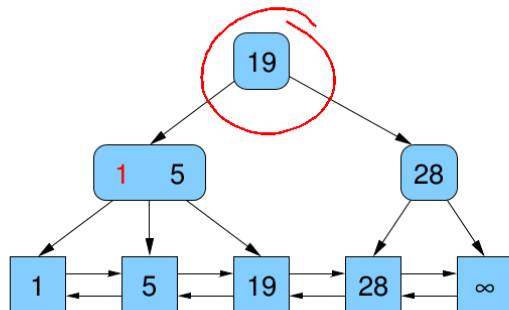
remove(3)

 $(a, b)$ -Baum / remove $a = 2, b = 4$ 

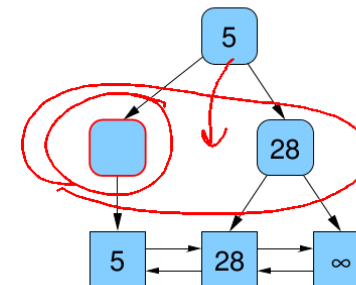
remove(3)

 $(a, b)$ -Baum / remove $a = 2, b = 4$ 

remove(3)

 $(a, b)$ -Baum / remove $a = 2, b = 4$ 

remove(19)



## (a, b)-Baum / remove

### Form-Invariante

- alle Blätter behalten dieselbe Tiefe
- falls alte Wurzel entfernt wird, verringert sich die Tiefe aller Blätter

### Grad-Invariante

- remove verschmilzt Knoten, die Grad  $a - 1$  und  $a$  haben
- wenn  $b \geq 2a - 1$ , dann ist der resultierende Grad  $\leq b$
- remove verschiebt eine Kante von Knoten mit Grad  $> a$  zu Knoten mit Grad  $a - 1$ , danach sind beide Grade in  $[a, b]$
- wenn Wurzel gelöscht, wurden vorher die Kinder verschmolzen, Grad vom letzten Kind ist also  $\geq a$  (und  $\leq b$ )

## Weitere Operationen im (a, b)-Baum

- **min/max-Operation**

verwende first/last-Methode der Liste, um das kleinste bzw. größte Element auszugeben

Zeit:  $O(1)$

- **Range queries** (Bereichsanfragen)

suche alle Elemente im Bereich  $[x, y]$ :

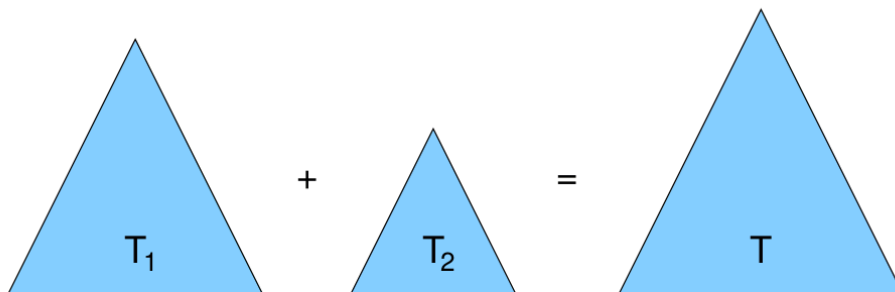
- führe locate( $x$ ) aus und
- durchlaufe die Liste, bis Element  $> y$  gefunden wird

Zeit:  $O(\log n + \text{Ausgabegröße})$

- **Konkatenation / Splitting**

## Konkatenation von (a, b)-Bäumen

- verknüpfe zwei  $(a, b)$ -Bäume  $T_1$  und  $T_2$  mit  $s_1$  bzw.  $s_2$  Elementen und Höhe  $h_1$  bzw.  $h_2$  zu  $(a, b)$ -Baum  $T$
- Bedingung: Schlüssel in  $T_1 \leq$  Schlüssel in  $T_2$



## Weitere Operationen im (a, b)-Baum

- **min/max-Operation**

verwende first/last-Methode der Liste, um das kleinste bzw. größte Element auszugeben

Zeit:  $O(1)$

- **Range queries** (Bereichsanfragen)

suche alle Elemente im Bereich  $[x, y]$ :

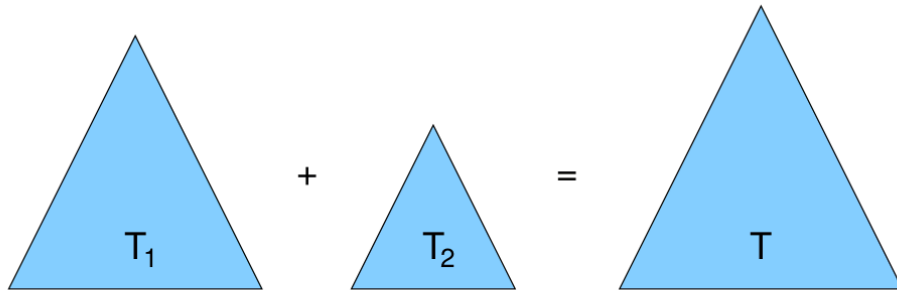
- führe locate( $x$ ) aus und
- durchlaufe die Liste, bis Element  $> y$  gefunden wird

Zeit:  $O(\log n + \text{Ausgabegröße})$

- **Konkatenation / Splitting**

## Konkatenation von (a, b)-Bäumen

- verknüpfe zwei (a, b)-Bäume  $T_1$  und  $T_2$  mit  $s_1$  bzw.  $s_2$  Elementen und Höhe  $h_1$  bzw.  $h_2$  zu (a, b)-Baum  $T$
- Bedingung: Schlüssel in  $T_1 \leq$  Schlüssel in  $T_2$



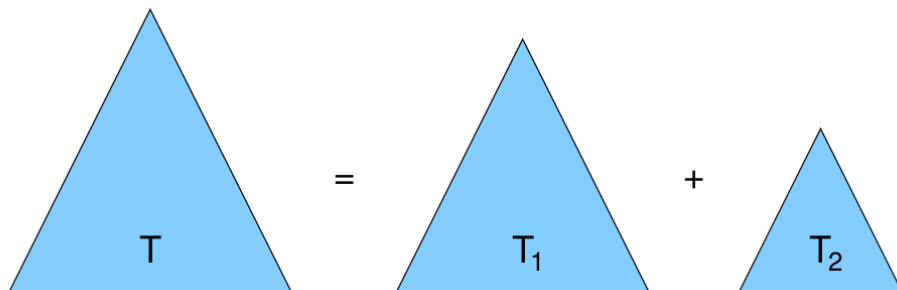
## Konkatenation von (a, b)-Bäumen

- lösche in  $T_1$  das  $\infty$ -Dummy-Element
- wenn danach dessen Vater-Knoten  $< a$  Kinder hat, dann behandle dies wie bei remove
- verschmelze die Wurzel des niedrigeren Baums mit dem entsprechenden äußersten Knoten des anderen Baums, der sich auf dem gleichen Level befindet
- wenn dieser Knoten danach  $> b$  Kinder hat, dann behandle dies wie bei insert

⇒ falls Höhe der Bäume explizit gespeichert: Zeit  $O(1 + |h_1 - h_2|)$   
 ansonsten (mit Höhenbestimmung): Zeit  $O(1 + \max\{h_1, h_2\})$   
 $\subseteq O(1 + \log(\max\{s_1, s_2\}))$

## Aufspaltung eines (a, b)-Baums

- spalte (a, b)-Baum  $T$  bei Schlüssel  $k$  in zwei (a, b)-Bäume  $T_1$  und  $T_2$  auf



## Aufspaltung eines (a, b)-Baums

- Sequenz  $q = \langle w, \dots, x, y, \dots, z \rangle$  soll bei Schlüssel  $y$  in Teile  $q_1 = \langle w, \dots, x \rangle$  und  $q_2 = \langle y, \dots, z \rangle$  aufgespalten werden

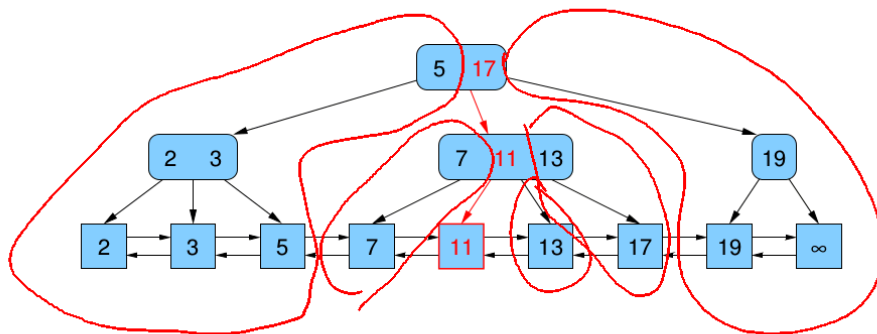
## Aufspaltung eines (a, b)-Baums

- Sequenz  $q = \langle w, \dots, x, y, \dots, z \rangle$  soll bei Schlüssel  $y$  in Teile  $q_1 = \langle w, \dots, x \rangle$  und  $q_2 = \langle y, \dots, z \rangle$  aufgespalten werden
- betrachte Pfad von Wurzel zum Blatt  $y$
- spalte auf diesem Pfad jeden Knoten  $v$  in zwei Knoten  $v_\ell$  und  $v_r$
- $v_\ell$  bekommt Kinder links vom Pfad,  $v_r$  bekommt Kinder rechts vom Pfad (evt. gibt es Knoten ohne Kinder)

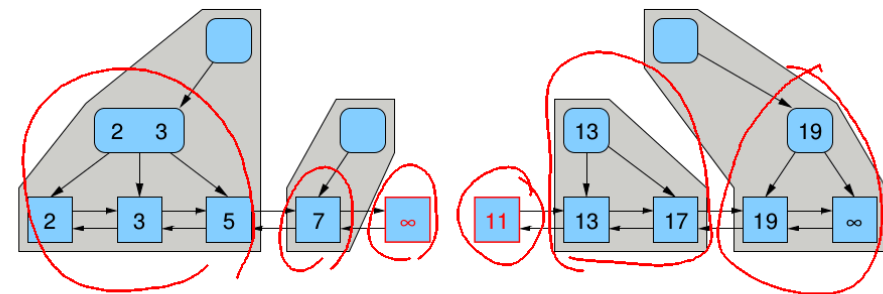
## Aufspaltung eines (a, b)-Baums

- Sequenz  $q = \langle w, \dots, x, y, \dots, z \rangle$  soll bei Schlüssel  $y$  in Teile  $q_1 = \langle w, \dots, x \rangle$  und  $q_2 = \langle y, \dots, z \rangle$  aufgespalten werden
- betrachte Pfad von Wurzel zum Blatt  $y$
- spalte auf diesem Pfad jeden Knoten  $v$  in zwei Knoten  $v_\ell$  und  $v_r$
- $v_\ell$  bekommt Kinder links vom Pfad,  $v_r$  bekommt Kinder rechts vom Pfad (evt. gibt es Knoten ohne Kinder)
- Knoten mit Kind(ern) werden als Wurzeln von (a, b)-Bäumen interpretiert
- Konkatenation der linken Bäume zusammen mit einem neuen  $\infty$ -Dummy ergibt einen Baum für die Elemente bis  $x$
- Konkatenation von  $\langle y \rangle$  zusammen mit den rechten Bäumen ergibt einen Baum für die Elemente ab  $y$

## Aufspaltung eines (a, b)-Baums



## Aufspaltung eines (a, b)-Baums



## Aufspaltung eines (a, b)-Baums

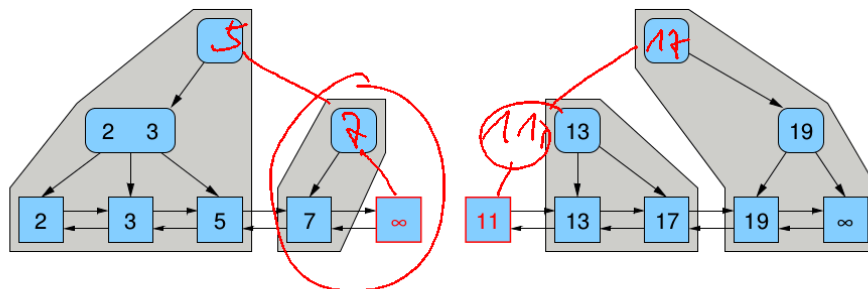
- Sequenz  $q = \langle w, \dots, x, y, \dots, z \rangle$  soll bei Schlüssel  $y$  in Teile  $q_1 = \langle w, \dots, x \rangle$  und  $q_2 = \langle y, \dots, z \rangle$  aufgespalten werden
- betrachte Pfad von Wurzel zum Blatt  $y$
- spalte auf diesem Pfad jeden Knoten  $v$  in zwei Knoten  $v_\ell$  und  $v_r$
- $v_\ell$  bekommt Kinder links vom Pfad,  $v_r$  bekommt Kinder rechts vom Pfad (evt. gibt es Knoten ohne Kinder)
- Knoten mit Kind(ern) werden als Wurzeln von (a, b)-Bäumen interpretiert
- Konkatenation der linken Bäume zusammen mit einem neuen  $\infty$ -Dummy ergibt einen Baum für die Elemente bis  $x$
- Konkatenation von  $\langle y \rangle$  zusammen mit den rechten Bäumen ergibt einen Baum für die Elemente ab  $y$

## Aufspaltung eines (a, b)-Baums

- diese  $O(\log n)$  Konkatenationen können in Gesamtzeit  $O(\log n)$  erledigt werden
- Grund: die linken Bäume haben echt monoton fallende, die rechten echt monoton wachsende Höhe
- Seien z.B.  $r_1, r_2, \dots, r_k$  die Wurzeln der linken Bäume und  $h_1 > h_2 > \dots > h_k$  deren Höhen
- verbinde zuerst  $r_{k-1}$  und  $r_k$  in Zeit  $O(1 + h_{k-1} - h_k)$ , dann  $r_{k-2}$  mit dem Ergebnis in Zeit  $O(1 + h_{k-2} - h_{k-1})$ , dann  $r_{k-3}$  mit dem Ergebnis in Zeit  $O(1 + h_{k-3} - h_{k-2})$  usw.
- Gesamtzeit:

$$O\left(\sum_{1 \leq i < k} (1 + h_i - h_{i+1})\right) = O(k + h_1 - h_k) \in O(\log n)$$

## Aufspaltung eines (a, b)-Baums



## Effizienz von insert / remove-Folgen

## Satz

Es gibt eine Folge von  $n$  insert- und remove-Operationen auf einem anfangs leeren  $[2, 3]$ -Baum, so dass die Gesamtanzahl der Knotenaufspaltungen und -verschmelzungen in  $\Omega(n \log n)$  ist.

Beweis: siehe Übung



## Effizienz von insert / remove-Folgen

## Satz

Für  $(a, b)$ -Bäume, die die erweiterte Bedingung  $b \geq 2a$  erfüllen, gilt:

Für jede Folge von  $n$  insert- und remove-Operationen auf einem anfangs leeren  $(a, b)$ -Baum ist die Gesamtanzahl der Knotenaufspaltungen und -verschmelzungen in  $O(n)$ .

Beweis: amortisierte Analyse, nicht in dieser Vorlesung

## Übersicht

## 9 Graphen

- Netzwerke und Graphen
- Graphrepräsentation
- Graphtraversierung
- Kürzeste Wege
- Minimale Spannbäume

## Netzwerk

Objekt bestehend aus

- Elementen und
- Interaktionen bzw. Verbindungen zwischen den Elementen

eher *informales* Konzept

- keine exakte Definition
- Elemente und ihre Verbindungen können ganz unterschiedlichen Charakter haben
- manchmal manifestiert in **real** existierenden Dingen, manchmal nur gedacht (**virtuell**)

## Beispiele für Netzwerke

- Kommunikationsnetze: Internet, Telefonnetz
- Verkehrsnetze: Straßen-, Schienen-, Flug-, Nahverkehrsnetz
- Versorgungsnetzwerke: Strom, Wasser, Gas, Erdöl
- wirtschaftliche Netzwerke: Geld- und Warenströme, Handel
- biochemische Netzwerke: Metabolische und Interaktionsnetzwerke
- biologische Netzwerke: Gehirn, Ökosysteme
- soziale / berufliche Netzwerke: virtuell oder explizit (Communities)
- Publikationsnetzwerke: Zitationsnetzwerk, Koautor-Netzwerk

## Graph

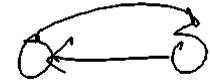
formales / abstraktes Objekt bestehend aus

- Menge von **Knoten**  $V$  (engl. vertices, nodes)
- Menge von **Kanten**  $E$  (engl. edges, lines, links), die jeweils ein Paar von Knoten verbinden
- Menge von Eigenschaften der Knoten und / oder Kanten

Notation:

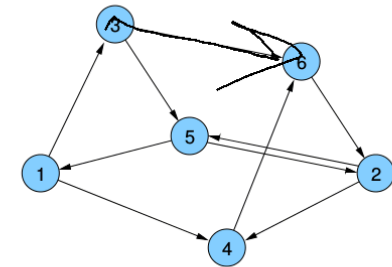
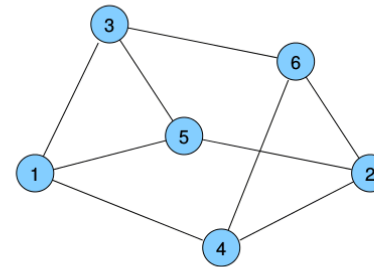
- $G = (V, E)$   
manchmal auch  $G = (V, E, w)$  im Fall gewichteter Graphen
- Anzahl der Knoten:  $n = |V|$   
Anzahl der Kanten:  $m = |E|$

## Gerichtete und ungerichtete Graphen



Kanten bzw. Graphen

- ungerichtet:  $E \subseteq \{\{v, w\} : v \in V, w \in V\}$   
(ungeordnetes Paar von Knoten bzw. 2-elementige Teilmenge)
- gerichtet:  $E \subseteq \{(v, w) : v \in V, w \in V\}$ , also  $E \subseteq V \times V$   
(geordnetes Paar von Knoten)



## Gerichtete und ungerichtete Graphen

Anwendungen:

- Ungerichtete Graphen:  
**symmetrische** Beziehungen (z.B.  $\{v, w\} \in E$  genau dann, wenn Person  $v$  und Person  $w$  verwandt sind)
- Gerichtete Graphen:  
**asymmetrische** Beziehungen (z.B.  $(v, w) \in E$  genau dann, wenn Person  $v$  Person  $w$  mag)  
**kreisfreie** Beziehungen (z.B.  $(v, w) \in E$  genau dann, wenn Person  $v$  Vorgesetzter von Person  $w$  ist)

hier:

- Modellierung von ungerichteten durch gerichtete Graphen
- Ersetzung ungerichteter Kanten durch je zwei antiparallele gerichtete Kanten

## Nachbarn: Adjazenz, Inzidenz, Grad

- Sind zwei Knoten  $v$  und  $w$  durch eine Kante  $e$  verbunden, dann nennt man
  - $v$  und  $w$  **adjazent** bzw. benachbart
  - $v$  und  $e$  **inzident** (ebenso  $w$  und  $e$ )
- Anzahl der Nachbarn eines Knotens  $v$ : **Grad**  $\deg(v)$   
bei gerichteten Graphen:
  - Eingangsgrad:  $\deg^-(v) = |\{(w, v) \in E\}|$
  - Ausgangsgrad:  $\deg^+(v) = |\{(v, w) \in E\}|$

## Annahmen



- Graph (also Anzahl der Knoten und Kanten) ist **endlich**
- Graph ist **einfach**, d.h.  $E$  ist eine Menge und keine Multimenge (anderenfalls heißt  $G$  Multigraph)
- Graph enthält **keine Schleifen** (Kanten von  $v$  nach  $v$ )

## Gewichtete Graphen

In Abhängigkeit vom betrachteten Problem wird Kanten und / oder Knoten oft eine Eigenschaft (z.B. eine Farbe oder ein numerischer Wert, das **Gewicht**) zugeordnet (evt. auch mehrere), z.B.

- Distanzen (in Längen- oder Zeiteinheiten)
- Kosten
- Kapazitäten / Bandbreite
- Ähnlichkeiten
- Verkehrsdichte

Wir nennen den Graphen dann

- **knotengewichtet** bzw.
- **kantengewichtet**

Beispiel:  $w : E \mapsto \mathbb{R}$

Schreibweise:  $w(e)$  für das Gewicht einer Kante  $e \in E$

## Wege, Pfade und Kreise

- **Weg** (engl. *walk*) in einem Graphen  $G = (V, E)$ : alternierende Folge von Knoten und Kanten  $x_0, e_1, \dots, e_k, x_k$ , so dass
  - $\forall i \in [0, k] : x_i \in V$  und
  - $\forall i \in [1, k] : e_i = \{x_{i-1}, x_i\}$  bzw.  $e_i = (x_{i-1}, x_i) \in E$ .
- **Länge** eines Weges: Anzahl der enthaltenen Kanten
- Ein Weg ist ein **Pfad**, falls er (in sich) kantendisjunkt ist, falls also gilt:  $e_i \neq e_j$  für  $i \neq j$ .
- Ein Pfad ist ein **einfacher Pfad**, falls er (in sich) knotendisjunkt ist, falls also gilt:  $x_i \neq x_j$  für  $i \neq j$ .
- Ein Weg heißt **Kreis** (engl. *cycle*), falls  $x_0 = x_k$ .

## Wege, Pfade und Kreise

- **Weg** (engl. *walk*) in einem Graphen  $G = (V, E)$ : alternierende Folge von Knoten und Kanten  $x_0, e_1, \dots, e_k, x_k$ , so dass
  - $\forall i \in [0, k] : x_i \in V$  und
  - $\forall i \in [1, k] : e_i = \{x_{i-1}, x_i\}$  bzw.  $e_i = (x_{i-1}, x_i) \in E$ .
- **Länge** eines Weges: Anzahl der enthaltenen Kanten
- Ein Weg ist ein **Pfad**, falls er (in sich) kantendisjunkt ist, falls also gilt:  $e_i \neq e_j$  für  $i \neq j$ .
- Ein Pfad ist ein **einfacher Pfad**, falls er (in sich) knotendisjunkt ist, falls also gilt:  $x_i \neq x_j$  für  $i \neq j$ .
- Ein Weg heißt **Kreis** (engl. *cycle*), falls  $x_0 = x_k$ .

## Operationen

**Graph**  $G$ : Datenstruktur (Typ/Klasse, Variable/Objekt) für Graphen

**Node**: Datenstruktur für Knoten, **Edge**: Datenstruktur für Kanten

Operationen:

- $G.insert(Edge\ e)$ :  $E := E \cup \{e\}$
- $G.remove(Key\ i, Key\ j)$ :  $E := E \setminus \{e\}$   
für Kante  $e = (v, w)$  mit  $key(v) = i$  und  $key(w) = j$
- $G.insert(Node\ v)$ :  $V := V \cup \{v\}$
- $G.remove(Key\ i)$ : sei  $v \in V$  der Knoten mit  $key(v) = i$   
 $V := V \setminus \{v\}$ ,  $E := E \setminus \{(x, y) : x = v \vee y = v\}$
- $G.find(Key\ i)$ : gib Knoten  $v$  mit  $key(v) = i$  zurück
- $G.find(Key\ i, Key\ j)$ : gib Kante  $(v, w)$  mit  $key(v) = i$  und  $key(w) = j$  zurück

## Operationen

Anzahl der Knoten **konstant**

$$\Rightarrow V = \{0, \dots, n-1\}$$

- (Knotenschlüssel durchnummeriert)

Anzahl der Knoten **variabel**

- Hashing kann verwendet werden für ein Mapping der  $n$  Knoten in den Bereich  $\{0, \dots, O(n)\}$
- $\Rightarrow$  nur konstanter Faktor der Vergrößerung gegenüber statischer Datenstruktur

## Operationen

Anzahl der Knoten **konstant**

$$\Rightarrow V = \{0, \dots, n-1\}$$

- (Knotenschlüssel durchnummeriert)

Anzahl der Knoten **variabel**

- Hashing kann verwendet werden für ein Mapping der  $n$  Knoten in den Bereich  $\{0, \dots, O(n)\}$
- $\Rightarrow$  nur konstanter Faktor der Vergrößerung gegenüber statischer Datenstruktur

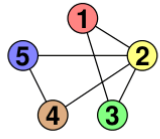
## Graphrepräsentation

Darstellung von Graphen im Computer?

Vor- und Nachteile bei z.B. folgenden Fragen:

- Sind zwei gegebene Knoten  $v$  und  $w$  **adjazent**?
- Was sind die **Nachbarn** eines Knotens?
- Welche Knoten sind (direkte oder indirekte) **Vorgänger** bzw. **Nachfolger** eines Knotens  $v$  in einem gerichteten Graphen?
- Wie aufwendig ist das **Einfügen** oder **Löschen** eines Knotens bzw. einer Kante?

## Kantenliste



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

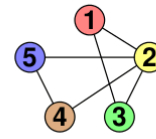
Vorteil:

- 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



|   |   |   |   |   |
|---|---|---|---|---|
| 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 |

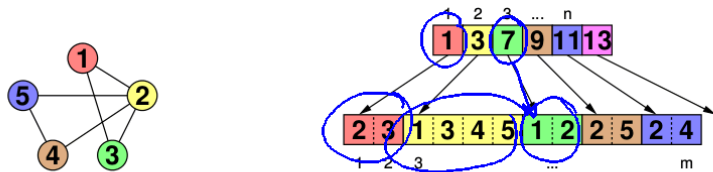
Vorteil:

- 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



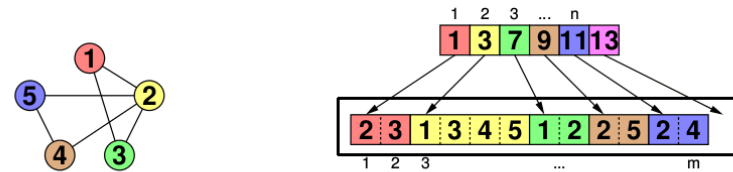
Vorteil:

- 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



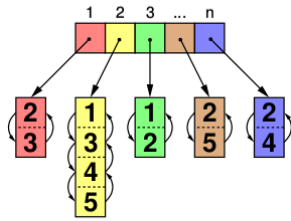
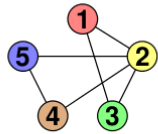
Vorteil:

- 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:

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

Speicheraufwand:  $O(n + m)$

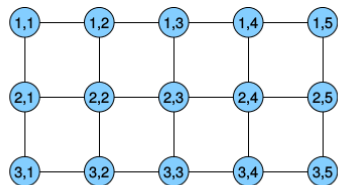
## Implizite Repräsentation

Beispiel: **Gitter-Graph** (grid graph)

- definiert durch zwei Parameter  $k$  und  $\ell$

$$V = [1, \dots, k] \times [1, \dots, \ell]$$

$$E = \left\{ ((i, j), (i, j')) \in V^2 : |j - j'| = 1 \right\} \cup \left\{ ((i, j), (i', j)) \in V^2 : |i - i'| = 1 \right\}$$



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

## Übersicht

- Graphen
  - Netzwerke und Graphen
  - Graphrepräsentation
  - Graphtraversierung
  - Kürzeste Wege
  - Minimale Spannbäume

# Graphtraversierung

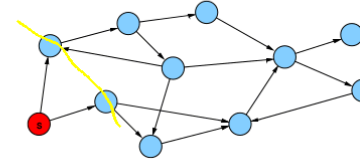
Problem:

Wie kann man die Knoten eines Graphen **systematisch durchlaufen**?

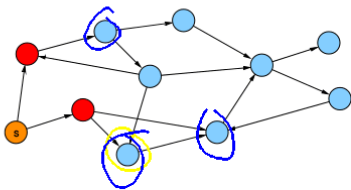
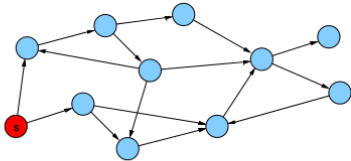
Grundlegende Strategien:

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

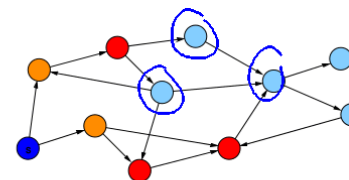
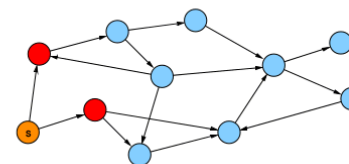
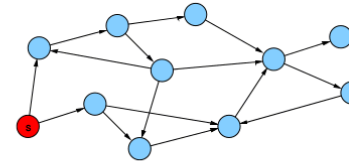
# Breitensuche



# Breitensuche

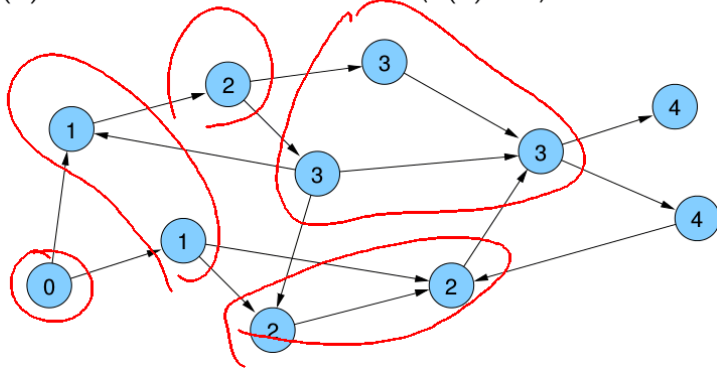


# Breitensuche



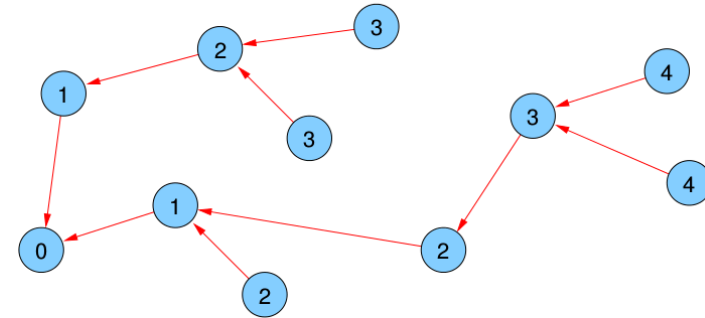
## Breitensuche

- $d(v)$ : Distanz von Knoten  $v$  zu  $s$  ( $d(s) = 0$ )



## Breitensuche

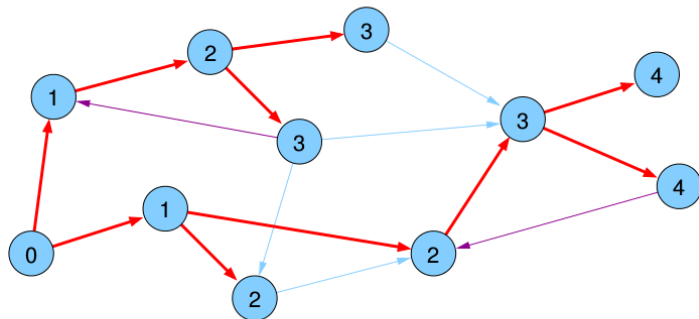
- $\text{parent}(v)$ : Knoten, von dem  $v$  entdeckt wurde
- $\text{parent}$  wird beim ersten Besuch von  $v$  gesetzt ( $\Rightarrow$  eindeutig)



## Breitensuche

Kantentypen:

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



## Breitensuche

```

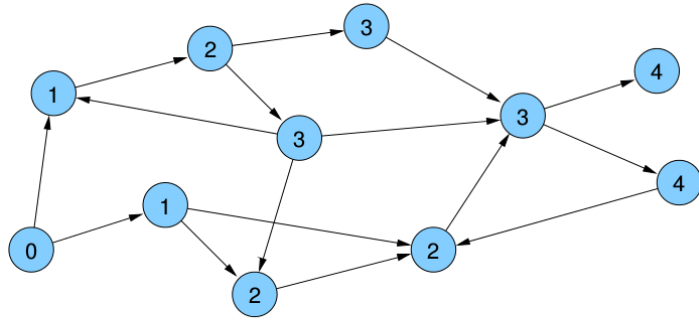
BFS(Node s) {
  d[s] = 0;
  parent[s] = s;
  List<Node> q = <s>;
  while ( !q.empty() ) {
    u = q.popFront();
    foreach ((u, v) ∈ E) {
      if (parent[v] == null) {
        q.pushBack(v);
        d[v] = d[u]+1;
        parent[v] = u;
      }
    }
  }
}

```



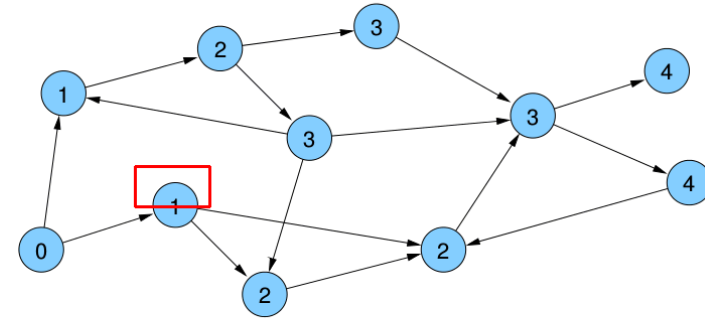
## Breitensuche

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



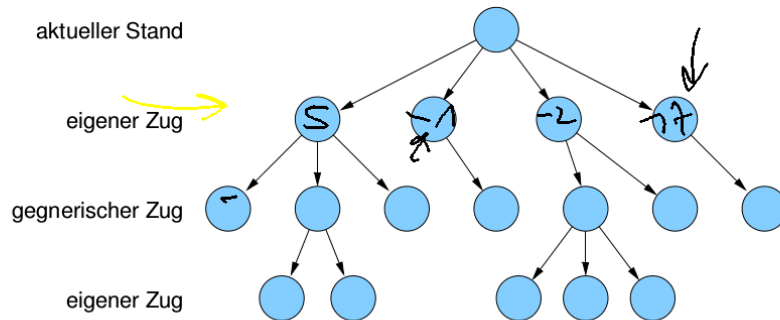
## Breitensuche

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



## Breitensuche

Anwendung: Bestimmung des nächsten Zugs bei Spielen  
Exploration des Spielbaums



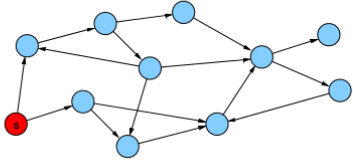
Problem: halte Aufwand zur Suche eines guten Zuges in Grenzen

## Breitensuche

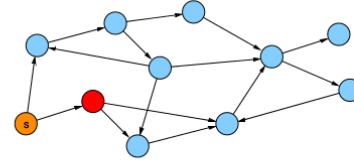
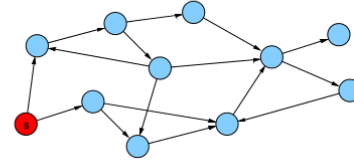
Anwendung: Bestimmung des nächsten Zugs bei Spielen

- 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

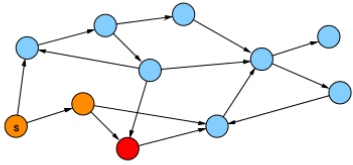
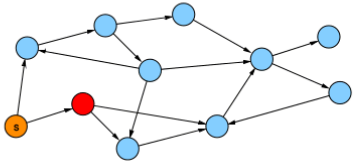
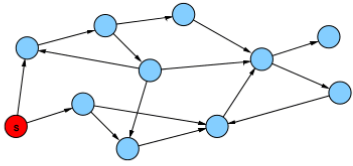
# Tiefensuche



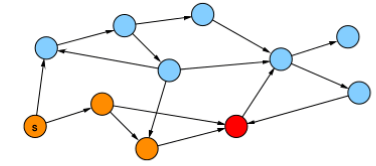
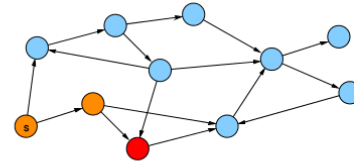
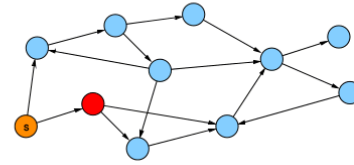
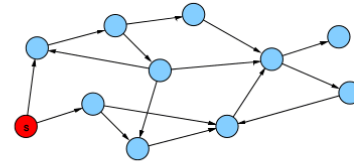
# Tiefensuche



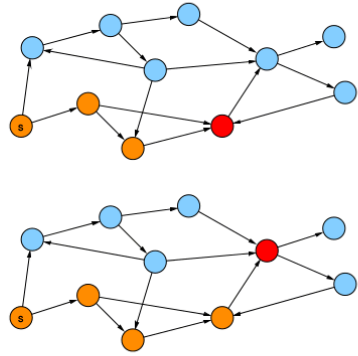
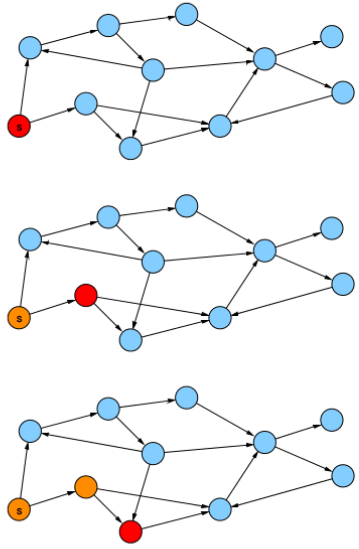
# Tiefensuche



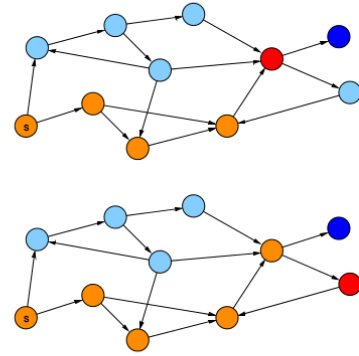
# Tiefensuche



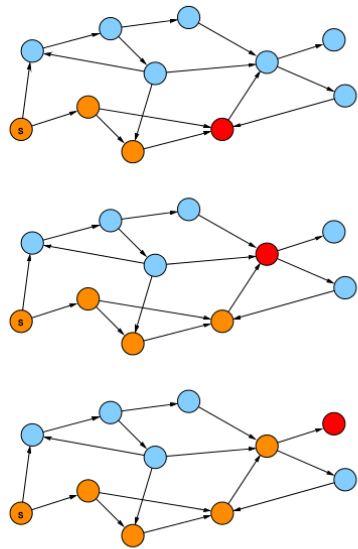
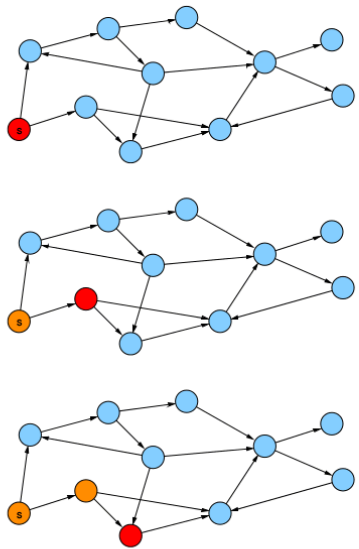
# Tiefensuche



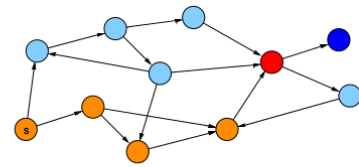
# Tiefensuche



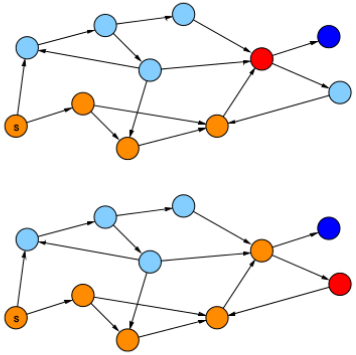
# Tiefensuche



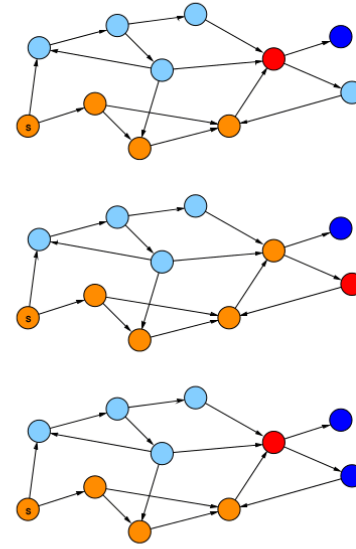
# Tiefensuche



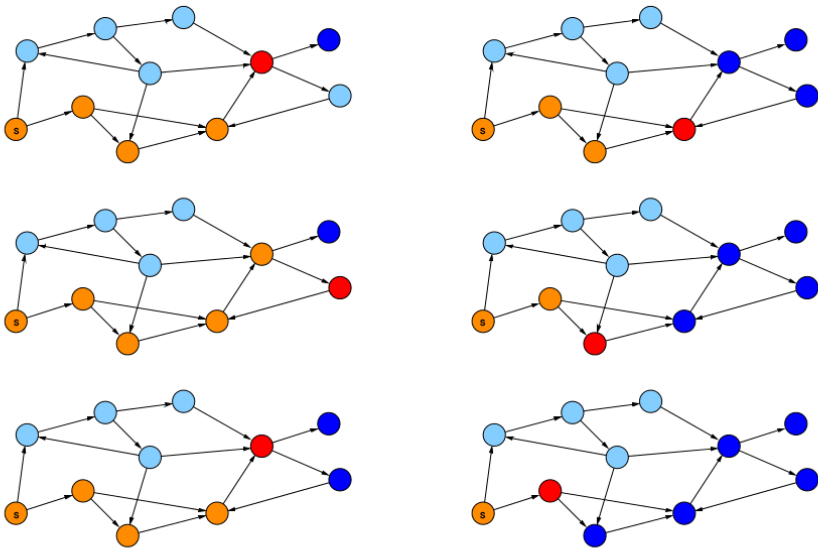
# Tiefensuche



# Tiefensuche



# Tiefensuche



# Tiefensuche

