Traversierung von Graphen

Die Algorithmen für eine Breiten- und Tiefensuche auf Bäumen lassen sich auf Graphen verallgemeinern.

Tiefensuche

`DFSvisit(v)` besucht alle (noch nicht besuchten) Knoten, die von `v` aus erreichbar sind.

`v->dfsNum` codiert hinterher die Reihenfolge in der Knoten besucht wurden.

Zusätzlich speichert `v->finNum` die Reihenfolge, in der die rekursiven Aufrufe beendet wurden.

Tiefensuche

```
DFS()
    dfsCount = 1;
    finCount = 1;
    foreach v ∈ V
        v->state  = initial;
        v->parent = NULL;
    foreach (s ∈ V)
        if s->state == initial
            DFSvisit(s)
```

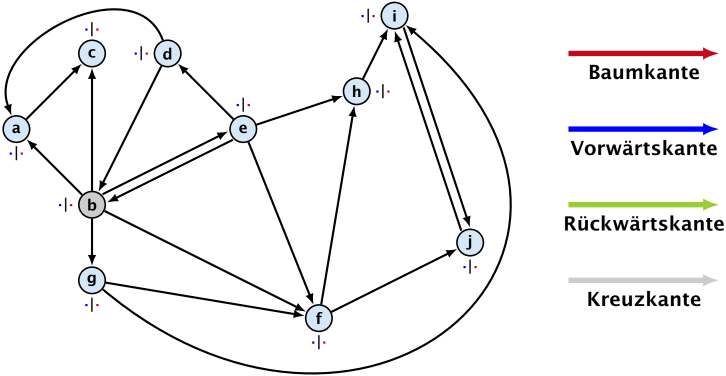
Tiefensuche – rekursiv

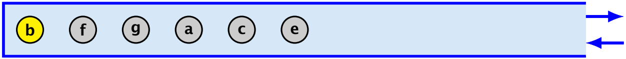
```
DFSvisit(v)
    v->state  = active;
    v->dfsNum = dfsCount++;
    foreach x ∈ N[v] // iteriere ueber Nachbarn
        if (x->state == initial)
            x->parent = v;
            DFSvisit(x);
    v->state  = finished;
    v->finNum = finCount++;
```

Tiefensuche – mit Stack

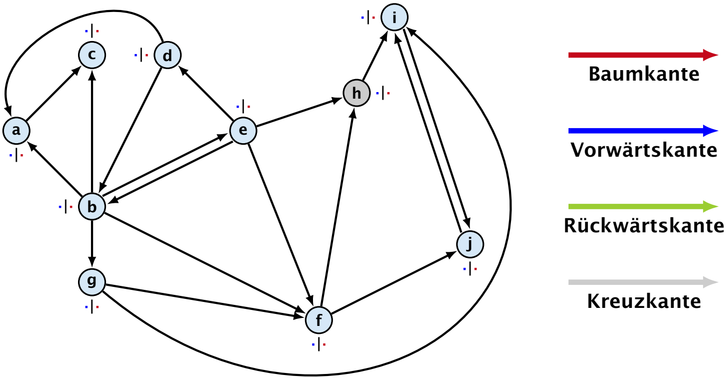
```
DFSvisit(s)
    Stack S; S.push(s);
    while (!S.empty()) {
        v = S.top();
        if (v->state == initial)
            v->state  = active;
            v->dfsNum = dfsCount++;
            foreach x ∈ N(v)
                if (x->state == active)      //(v,x) back edge
                else if (x->state == finished)//(v,x) cross edge
                else // (x->parent,x) forward edge
                    x->parent = v;
                    S.push(x);
        else
            v->state  = finished;
            v->finNum = finCount++;
            S.pop();
    }
```

DFS


a
b
c
d
e
f
g
h
i
j
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e

DFS



a
b
c
d
e
f
g
h
i
j
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

h
i
d
b
a

Tiefensuche

Kantentypen

- **Baumkanten**
  Für jeden Nichtwurzelknoten v, speichert man den Vorgänger über den v „betreten" wurde. Diese Kanten erzeugen einen Wald.
  Die Bäume sind von der Wurzel zu den Blättern gerichtet.
- **Vorwärtskanten**
  Falls für Kante (x, y) ein Pfad von x nach y in einem Baum existiert (und (x, y) nicht Baumkante).
- **Rückwärtskanten**
  Falls für Kante (x, y) ein Pfad von y nach x in einem Baum existiert.
- **Kreuzkanten**
  Alle anderen Kanten.

Tiefensuche

Eigenschaften der DFS-Nummerierung einer Kante `(x,y)`:

Baumkante oder **Vorwärtskante**

`x->dfsNum < y->dfsNum` und `x->finNum > y->finNum`

Rückwärtskante

`x->dfsNum > y->dfsNum` und `x->finNum < y->finNum`

Kreuzkante

`x->dfsNum > y->dfsNum` und `x->finNum > y->finNum`

Es kann keine Kanten mit `x->dfsNum < y->dfsNum` und `x->finNum < y->finNum` geben.

Komplexität der Tiefensuche

Rekursive Implementierung

- erste Schleife in DFS wird genau $|V|$ mal aufgerufen
- `DFSvisit(v)` wird für jeden Knoten genau einmal aufgerufen
- Schleife in `DFSvisit(v)` wird $|N[v]|$ mal aufgerufen;

$$\sum_{v \in V} |N[v]| \leq 2|E| = \Theta(|E|)$$

- Gesamtlaufzeit: $\Theta(|V| + |E|)$

Komplexität der Stackimplementierung ist asymptotisch gleich.

Tiefensuche – Anwendungen

Test auf (starken) Zusammenhang in Graphen.

- rufe `DFSvisit(v)` nur für einen Startknoten auf
- falls danach nicht alle Knoten besucht, ist Graph nicht (stark) zusammenhängend

Test auf Zyklenfreiheit.

- Graph hat Zyklus genau dann wenn die Tiefensuche eine Rückwärtskante enthält
- $\Leftarrow$ wenn man eine Rückwärtskante hat wir mit dieser Rückwärtskante über die Baumkanten ein Zyklus geschlossen
- $\Longrightarrow$ angenommen man hat einen Zyklus und die Tiefensuche liefert keine Rückwärtskante; dann ist `finNum` entlang des Zyklus steigend, da dieser nur aus Vorwärtskanten, Baumkanten, und Kreuzkanten besteht (Widerspruch)

Breitensuche

Anwendung:

Bestimmung der Entfernung zum Startknoten in einem ungewichteten Graphen.

Breitensuche

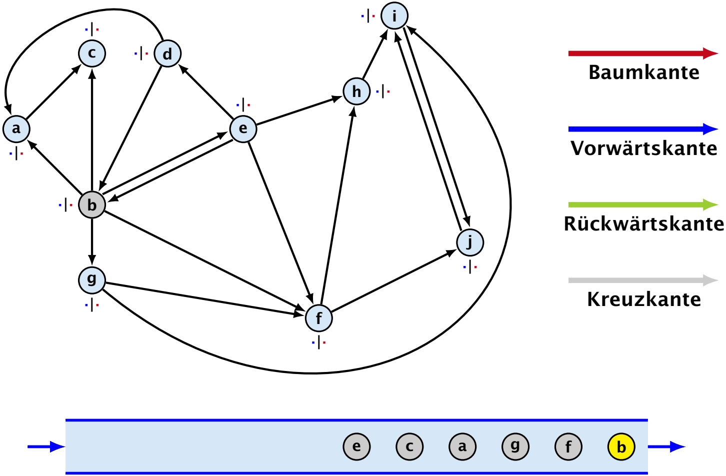
```
BFS()
    bfsCount = 1;
    finCount = 1;
    foreach s ∈ V
        v->state  = initial;
        v->parent = NULL;
    foreach (s ∈ V)
        if s->state == initial
            BFSvisit(s)
```

Breitensuche – mit Queue

```
BFSvisit(s)
    Queue Q; Q.enqueue(s);
    while (!Q.empty()) {
        v = Q.front();
        if (v->state == initial)
            v->state  = active;
            v->bfsNum = bfsCount++;
            foreach x ∈ N(v)
                // v->parent = x; removed statement
                Q.enqueue(x);
        else
            v->state  = finished;
            v->finNum = finCount++;
            Q.dequeue();
    }
```

Wir haben im wesentlichen nur den Stack durch eine Queue ersetzt!

BFS



a
b
c
d
e
f
g
h
i
j
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
e
c
a
g
f
b

Beobachtungen

- `finNum` ist immer gleich `bfsNum` (also überflüssig)
- keine Vorwärtskanten
- Klassifizierung von Kreuzkanten und Rückwärtskanten ist nicht so einfach möglich; wird normalerweise bei BFS auch nicht gemacht
- Komplexität die gleiche wie DFS, da nur ein Stack gegen eine Queue getauscht wurde.

Breitensuche – mit Queue

```
BFSvisit(s)
    Queue Q; Q.enqueue(s); s->dist = 0;
    while (!Q.empty()) {
        v = Q.front();
        if (v->parent) v->dist = v->parent->dist+1;
        if (v->state == initial)
            v->state  = active;
            v->bfsNum = bfsCount++;
            foreach x ∈ N(v)
                Q.enqueue(x);
        else
            v->state  = finished;
            v->finNum = finCount++;
            Q.dequeue();
    }
```

Berechnet die Distanz zu s im Baum. Für einen BFS-Baum ist dies auch die Distanz zu s im ungewichteten Graphen.

Beweis:

- Sei $\mathrm{dist}(x)$ der berechnete Wert, und $\mathrm{rdist}(x)$ die wirkliche Distanz zu s.
- $\mathrm{rdist}(x) \le \mathrm{dist}(x)$ folgt, da der Baum einen Pfad zu s der Länge dist codiert.
- Angenommen es gibt einen Knoten v mit $\mathrm{rdist}(v) < \mathrm{dist}(v)$; sei v ein solcher Knoten mit kleinstem Wert von $\mathrm{dist}(v)$

  Die Knoten werden gemäß der Größe ihrer dist-Werte besucht. (warum?)

  Sei p_v, der vom Algorithmus gefundene Vorgänger von v (`v->parent`) und sei p'_v der Vorgänger von v auf einem kürzesten s-v Pfad.

  - $\mathrm{dist}(p_v) = \mathrm{rdist}(p_v) = \mathrm{dist}(v) - 1$
  - $\mathrm{dist}(p'_v) = \mathrm{rdist}(p'_v) - 1 < \mathrm{dist}(v) - 1$
  - dann wird aber p'_v zuerst besucht; und dabei würden wir `v->parent` auf p'_v setzen. Widerspruch.