Grundlagen der Programmierung

Algorithmen auf Graphen:

Breitensuche ♦ Tiefensuche

Kleine-Welt-Problem

- Soziale Netzwerke
- Computernetzwerke
  - schneller Informationsfluss zwischen zwei Computern im Netzwerk zu erwarten
- Netzwerk repräsentiert als ungerichteter Graph
- Berechnung der Abstandsverteilung

➢ Berechnung des Abstands zweier Knoten *u* und *v* (Länge des kürzesten Pfades zwischen *u* und *v*)

Abstand von Knoten

Name: Abstand von Knoten (**Brute Force**)
Eingabe: ungerichteter, schlingenfreier Graph $G = (V,E)$,
$u, v \in V, u \neq v$
Ausgabe: $D(u,v)$

für $k \longleftarrow 1$ **bis** $|V| - 1$
    für alle Teilmengen $V' \subseteq V$ der Größe k-1 existiert
        für jede Permutation der Elemente von V'
            falls Pfad von u nach v über diese
                gib k **aus**
                STOP
gib ∞ **aus**

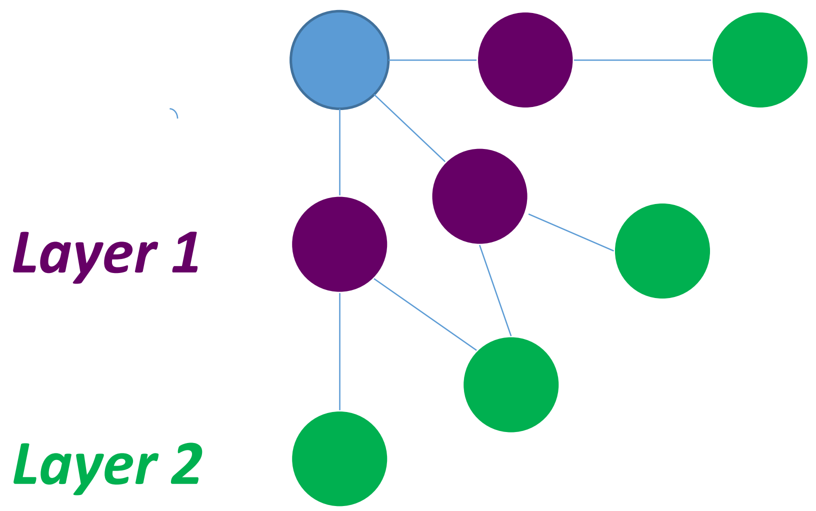
$$t(n) \in \Omega(2^n)$$

Neue Algorithmische Idee?!

- Berechnung des Abstands von u zu allen Knoten
- d_v : Abstand von Knoten u zu Knoten v
- $d_u = 0$
- Sei d_j berechnet, d_k noch nicht.
  Falls $\{j,k\} \in E$, dann $d_k = d_j + 1$.

Breitensuche (BFS) in Graphen

- zuerst alle Nachbarn des Knotens *u* bestimmen (*Layer 1*)
- dann für alle Knoten aus *Layer 1* alle (neuen) Nachbarn bestimmen (*Layer 2*)
- usw.

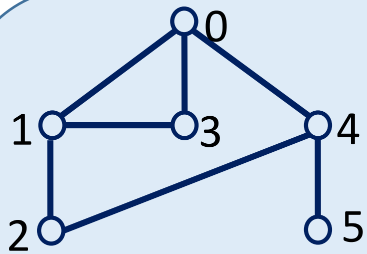


BFS und Warteschlangen

- BFS muss Buch führen über
  - Knoten, für die Abstand bereits berechnet ist
  - Knoten, deren Nachbarn noch zu betrachten sind
- BFS soll dabei
  - Nachbarn der Knoten in derselben Reihenfolge behandeln, wie die Knoten selbst aufgesucht wurden
- Warteschlange (Queue):

  Eine **Queue (Warteschlange)** ist eine Kollektion, die die Daten nach dem **First-In-First-Out (FIFO)**-Prinzip verwaltet.

BFS und Warteschlangen



0	1, 3, 4
1	0, 2, 3
2	1, 4
3	0, 1
4	0, 2, 5
5	4

0, 1, 3, 4, 2, 5

[0] **dequeue:** j = 0

[] **enqueue:** 1, dann 3, dann 4

[1] → [3, 1] → [4, 3, 1]

[4, 3, 1] **dequeue:** j = 1

[4, 3] **enqueue:** 2

[2, 4, 3] **dequeue:** j = 3

[2, 4] **dequeue:** j = 4

[2] **enqueue:** 5 2

[5, 2] → [5] → [] 5

Eine **Queue (Warteschlange)** ist eine Kollektion, die die Daten nach dem **First-In-First-Out (FIFO)**-Prinzip verwaltet.

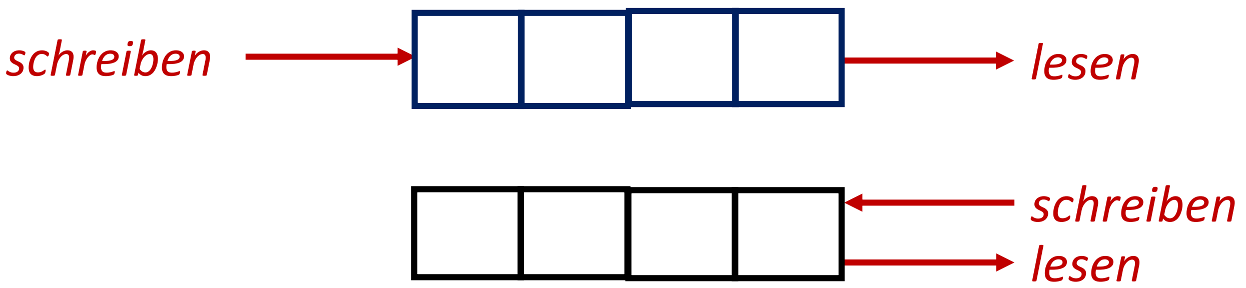
Queue

Liste, bei der neue Elemente immer nur an einem Ende eingefügt und immer nur am anderen Ende gelöscht werden können:

- Einfügen:
```
qu.insert(0,x)      # enqueue
qu[:0] = [x]
```
- Löschen:
```
qu.pop()            # dequeue
del qu[len(qu)-1]
# und Rückgabe von qu[len(qu)-1]
```

Queue:

Stack:



BFS Algorithmus

Name: Abstand von Knoten (BFS)

Eingabe: ungerichteteter, schlingenfreier Graph $G = (V,E)$,

$u \in V$

Ausgabe: Liste mit Abständen aller Knoten j in V zu u

also Liste D mit $D[j] = D(u,j) = d_j$

BFS Algorithmus

$Q \longleftarrow$ leere Warteschlange
für alle $i \in V$
    $D[i] \longleftarrow \infty$
$D[u] \longleftarrow 0$
enqueue(Q,u)
solange Q nicht leer ist
    $j \longleftarrow$ dequeue(Q)
    für alle Nachbarn k von j
        falls $D[k] = \infty$
            $D[k] \longleftarrow D[j] + 1$
            enqueue (Q,k)
gib D **aus**

korrekt:
auch wenn G nicht zusammenhängend

terminiert:
auch wenn G nicht zusammenhängend

BFS – grobe Laufzeitanalyse

- Annahmen:
  - G gegeben in Adjazenzlisten-Repräsentation
  - n Knoten, m Kanten
  - G ist zusammenhängend $\rightarrow$ **worst case**

BFS Algorithmus

O(1) $Q \longleftarrow$ leere Warteschlange — **O(1)**

O(*n*) **für alle** $i \in V$

  O(1) $D[i] \longleftarrow \infty$ — **O(*n*)**

O(1) $D[u] \longleftarrow 0$

O(1) enqueue(Q,u) — **O(1)**

O(*n*) **solange** Q nicht leer ist

  O(1) $j \longleftarrow$ dequeue(Q)

  O(*n*) **für alle** k in adj[j]

    O(1) **falls** $D[k] = \infty$

      O(1) $D[k] \longleftarrow D[j] + 1$

      O(1) enqueue (Q,k) — $\boxed{O(n^2)}$

O(1) **gib** D **aus** — **O(1)**

BFS – genauere Laufzeitanalyse

- Beobachtungen
  - Anzahl der Kanten spielt (noch) keine Rolle: $O(n^2)$
  - G mit $n = 1.000.000$ und $m = 1$ **?!**

$Q \longleftarrow$ leere Warteschlange
für alle $i \in V$
    $D[i] \longleftarrow \infty$
$D[u] \longleftarrow 0$
enqueue(Q,u)
solange Q nicht leer ist
    $j \longleftarrow$ dequeue(Q)
    für alle k in adj[j]
        falls $D[k] = \infty$
            $D[k] \longleftarrow D[j] + 1$
            enqueue (Q,k)
gib D aus

O(*n*)

O(*m*)

$\boxed{\mathbf{O}(n+m)}$

Diskussion

- O(n+m) ist im Vergleich zu O(n^2)
  - kein Widerspruch
  - aber genauer

  da $m \in O(n^2)$

- In wirklichen Anwendungsszenarien: $m > n$
  - ➢ O(m)

BFS – Zusammenhangskomponente

- BFS ist vielfältig einsetzbar; andere Anwendung:

Name: Zusammenhangskomponente (BFS)

Eingabe: ungerichteter, schlingenfreier Graph $G = (V,E)$,

$u \in V$

Ausgabe: Anzahl der mit *u* zusammenhängenden Knoten

Idee:

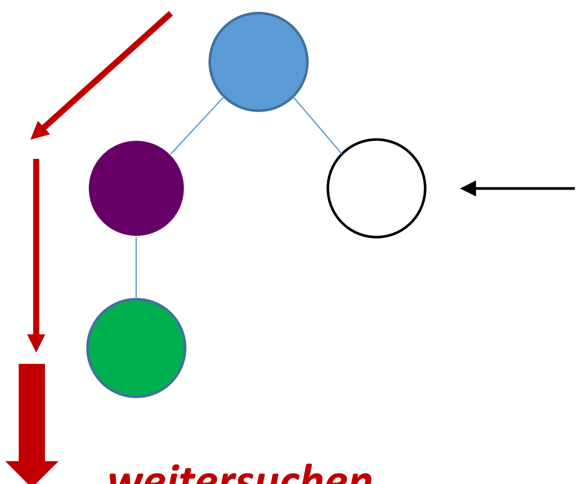
- alle Knoten aufsuchen (BFS)
- neu angetroffene Knoten mitzählen
- Liste *mark* zum Speichern, ob Knoten bereits aufgesucht oder nicht

BFS – Zusammenhangskomponente

```
Q ⟵ leere Warteschlange
für alle  i ∈ V
          mark[i] ⟵ 0
mark[u] ⟵ 1
z ⟵ 1    # Arbeit auf u
enqueue(Q,u)
solange Q nicht leer ist
          j ⟵ dequeue(Q)
          für alle k in adj[j]
                    falls mark[k] = 0
                              z ⟵ z + 1            # Arbeit auf k
                              mark[k] ⟵ 1
                              enqueue (Q,k)
gib z aus
```

Tiefensuche (DFS) in Graphen

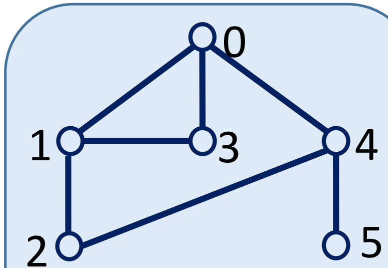
- von jedem gefundenen Knoten sofort einen neuen Nachbarn suchen (*sofort weiter „in die Tiefe“ gehen*)
- erst, wenn so kein neuer Knoten gefunden werden kann, **zurückgehen** zum zuletzt gefundenen Knoten, der noch weitere Nachbarn haben kann: **Backtracking**



Realisierung mit **Stack**!

DFS – Algorithmus

```
für alle i ∈ V
        mark[i] ⟵ 0
S ⟵ leerer Stack
mark[u] ⟵ 1
z ⟵ 1
push(S,u)
solange S nicht leer ist
        akt ⟵ S[|S|]    # Top-Element top(S) im Stack S
        falls k ∈ adj[akt] existiert, für das mark[k] = 0
                mark[k] ⟵ 1
                z ⟵ z + 1
                push(S,k)
        sonst
                pop(S)
gib z aus
```



0	1, 3, 4
1	0, 2, 3
2	1, 4
3	0, 1
4	0, 2, 5
5	4