

# Zusammenfassung Platzeffiziente Alg.

© Tim Baumann, <http://timbaumann.info/uni-spicker>

**Ziel.** Algorithmen entwerfen, die wenig Speicherplatz und Speicherzugriffe benötigen, aber trotzdem schnell sind.

**Problem (Erreichbarkeit).** Gegeben sei ein gerichteter oder ungerichteter Graph, ein Startknoten und ein Zielknoten darin. Frage: Ist der Zielknoten vom Startknoten erreichbar?

**Algorithmus.** Algorithmen, mit denen man Problem lösen kann, sind Breiten- und Tiefensuche.

**Lem.** Es sei ein Graph mit  $n$  Knoten und  $m$  Kanten gegeben. Tiefensuche benötigt  $\Theta(n + m)$  Zeit und  $\Theta(n \log n)$  Speicherplatz.

**Algorithmus (Savitch).**

```
1: function sREACHABLE( $u, v, k$ )
2:   if  $u = v$  then return true
3:   if  $k = 0$  then return false
4:   if  $(u, v) \in E$  then return true
5:   if  $k = 1$  then return false
6:   for  $x \in V$  do
7:     if sREACHABLE( $u, x, \lfloor \frac{k}{2} \rfloor$ )  $\wedge$  sREACHABLE( $x, v, \lceil \frac{k}{2} \rceil$ ) then
8:       return true
9:   return false

10: return sReachable(s,t,n-1)
```

**Lem.** Savitch's Algorithmus löst das Erreichbarkeits-Problem in  $\mathcal{O}((\log n)^2)$  Speicherplatz.

*Bem.* Die Laufzeit von Savitch's Alg. ist allerdings sehr schlecht, im schlechtesten Fall  $\mathcal{O}(n \cdot \log n)$ .

*Bem.* Eine Darstellung eines Graphen als Adjazenzmatrix benötigt  $\mathcal{O}(n^2)$ , eine Darstellung als Adjazenzzliste/-array  $\mathcal{O}(m \cdot \log n)$  Bits. Manchmal ist es nützlich, zusätzlich Rückwärtskanten oder Aus- und Ingrad von Knoten zu speichern, um diese Informationen nicht mehrmals berechnen zu müssen. Bei bestimmten Algorithmen werden sie auch als gegeben angenommen.

**Konvention.** Wir werden folgende Graphfunktionen benutzen:

Funktion	Ergebnis
$\text{adjfirst} : V \rightarrow P$	ersten Eintrag in der Adjazenzzliste
$\text{adjhead} : P \rightarrow V$	Knoten zum Eintrag in der Adjazenzzliste
$\text{adjnext} : P \rightarrow P$	nächsten Eintrag in der Adjazenzzliste
$\text{deg} : V \rightarrow \mathbb{N}$	Ausgrad eines Knoten
$\text{head} : A \rightarrow V$	den $k$ -ten Nachbar eines Knoten
$\text{tail} : B \rightarrow V$	den $k$ -ten In-Nachbar eines Knoten
$\text{mate} : A \rightarrow A$	den „Mate“ einer Kante (bei unger. Graphen)

wobei  $A := \{(v, k) \in V \times \mathbb{N} \mid 1 \leq k \leq \text{deg}(v)\}$   
 $B := \{(v, k) \in V \times \mathbb{N} \mid 1 \leq k \leq \text{indeg}(v)\}$

**Algorithmus.** Bei einer Tiefensuche in einem Graphen wird am meisten Platz für den Laufzeitstack verbraucht. Um diesen Platz zu optimieren, ist es geschickt, zunächst den Algorithmus mit explizitem Keller aufzuschreiben:

```
1: function PROCESS( $u$ )
2:    $S \leftarrow (u, \text{ADJFIRST}(u))$ 
3:   while  $S \neq \emptyset$  do
4:      $(u, p) \leftarrow S$ 
5:     if  $\text{color}[u] = \text{white}$  then
6:        $\text{color}[u] := \text{gray}$ 
7:       PREPROCESS( $u$ )
8:     if  $p \neq \text{null}$  then
9:        $S \leftarrow (u, \text{ADJNEXT}(p))$ 
10:       $v := \text{ADJHEAD}(u, p)$ 
11:      PREEXPLORE( $u, v, \text{color}[v]$ )
12:      if  $\text{color}[v] = \text{white}$  then
13:         $S \leftarrow (v, \text{ADJFIRST}(v))$ 
14:      else
15:        POSTEXPLORE( $u, v$ )
16:   else
17:     POSTPROCESS( $u$ )
18:   if  $S \neq \emptyset$  then
19:      $(w, -) := \text{PEEK}(S)$ 
20:     POSTEXPLORE( $w, u$ )
21:    $\text{color}[u] := \text{black}$ 
```

**Thm.** Für alle  $\epsilon > 0$  gibt es ein  $n_0 \in \mathbb{N}$ , sodass eine Tiefensuche eines Graphen (repräsentiert mit Adjazenzzlisten) mit  $n \geq n_0$  Ecken und  $m$  Kanten in  $\mathcal{O}((n + m) \log n)$  Zeit und  $(\log_2 3 + \epsilon)n$  Bits an Arbeitsspeicher durchgeführt werden kann.

**Algorithmus (DFS with logarithmic slowdown).**

Teile den Stack in Segmente der Größe  $q$  auf, wobei  $q = \Theta(n/\log n)$ . Wir behalten immer nur die obersten beiden Segmente des imaginären vollständigen Stacks  $S$  auf dem Stack  $S'$  unseres Algorithmus. Falls  $S'$  leerläuft, so müssen wir die obersten Segmente rekonstruieren: Dazu färben wir alle grauen Knoten wieder weiß und führen eine erneute Tiefensuche beginnend beim Startknoten aus.

**Thm.** Eine Tiefensuche eines Graphen (repräsentiert mit Adjazenzzarrays) mit  $n$  Ecken kann in  $\mathcal{O}(n + m)$  Zeit und  $\mathcal{O}(n \log \log n)$  Bits an Arbeitsspeicher durchgeführt werden.

**Algorithmus (log  $n$  shades of gray).**

Wir ändern den vorhergehenden Algorithmus folgendermaßen ab: Wir behalten nicht bloß die obersten beiden Segmente von  $S$  auf dem Stack  $S'$ , sondern auch zusätzlich von jedem Segment den obersten Knoten, den *Trailer*. Wir starten die Rekonstruktion nicht beim Startknoten, sondern beim Trailer des Segments unter dem Segment, das wir rekonstruieren wollen. Vor dem Rekonstruieren löschen wir die Farben nicht, da dies zu viel Laufzeit kosten würde. Während des Rekonstruierens müssen wir für jeden graue Knoten wissen, ob er weiter unten im Stack liegt, oder ob wir ihn rekonstruieren müssen. Dazu verwenden wir  $\mathcal{O}(n/q) = \mathcal{O}(\log n)$  Grautöne, einen für jede Segmenttiefe, wobei tiefere Segmente dunklere Grautöne bekommen. Zum Speichern der Grautöne für jeden Knoten benötigen wir  $\mathcal{O}(n \log \log n)$  Bits. Damit wir beim Rekonstruieren die Adjazenzzlisten der Knoten nicht wiederholt durchlaufen müssen, speichern wir für jeden Knoten (annähernd), wie weit wir in der Liste schon fortgeschritten sind. Genauer speichern wir die exakte Position für Knoten mit  $\geq m/q$  ausgehenden Kanten. Falls ein Knoten weniger ausgehende Kanten besitzt, so speichern wir nur eine Approximation  $\mathcal{O}(\log \log n)$  Bits.