Zusammenfassung Platzeffiziente Alg.

© BY: 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) ∈ E then return true
5:     if k = 1 then return false
6:     for x ∈ V do
7:         if sReachable(u, x, ⌊k/2⌋) ∧ sReachable(x, v, ⌈k/2⌉) 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 $O(n \cdot \log n)$.

Bem. Eine Darstellung eines Graphen als Adjazenzmatrix benötigt $O(n^2)$, eine Darstellung als Adjazenzliste/-array $O(m \cdot \log n)$ 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
adjfirst : $V \to P$	ersten Eintrag in der Adjazenzliste
adjhead : $P \to V$	Knoten zum Eintrag in der Adjazenzliste
adjnext : $P \to P$	nächsten Eintrag in der Adjazenzliste
deg : $V \to \mathbb{N}$	Ausgrad eines Knoten
head : $A \to V$	den k-ten Nachbar eines Knoten
tail : $B \to V$	den k-ten In-Nachbar eines Knoten
mate : $A \to A$	den „Mate“ einer Kante (bei unger. Graphen)

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

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 ⇐ (u, adjfirst(u))
3:     while S ≠ ∅ do
4:         (u, p) ⇐ S
5:         if color[u] = white then
6:             color[u] := gray
7:             preprocess(u)
8:         if p ≠ null then
9:             S ⇐ (u, adjnext(p))
10:            v := adjhead(u, p)
11:            preexplore(u, v, color[v])
12:            if color[v] = white then
13:                S ⇐ (v, adjfirst(v))
14:            else
15:                postexplore(u, v)
16:        else
17:            postprocess(u)
18:            if S ≠ ∅ then
19:                (w, –) := peek(S)
20:                postexplore(w, u)
21:            color[u] := black
```

Thm. Für alle $\epsilon > 0$ gibt es ein $n_0 \in \mathbb{N}$, sodass eine Tiefensuche eines Graphen (repräsentiert mit Adjazenzlisten) mit $n \ge n_0$ 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 Adjazenz-*arrays*) 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 Adjazenzlisten 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 $\ge m/q$ ausgehenden Kanten. Falls ein Knoten weniger ausgehende Kanten besitzt, so speichern wir nur eine Approximation $\mathcal{O}(\log\log n)$ Bits.

Algorithmus (Topologisches Sortieren eines azyklischen Graphen)**.** Führe eine Tiefensuche durch. Lege in postProcess(u) den Knoten u auf einen Stack. Gebe nach Abschluss der Tiefensuche alle Knoten auf dem Stack aus, also in umgekehrter Reihenfolge, wie sie auf den Stack gelegt wurden.

Problem. Die Maximalgröße des Stacks beträgt $\mathcal{O}(n \log n)$.

Lem. Wenn eine Folge von n Elementen à $\mathcal{O}(\log n)$ Bits in $t(n)$ Zeit und mit $s(n)$ Bits berechnet werden kann, so kann die umgekehrte Folge in $\mathcal{O}(t(n)\log n)$ Zeit und $\mathcal{O}(s(n)+n)$ Bits berechnet werden.

Beweis. Wir berechnen die Folge mehrmals und speichern jeweils ein Segment mit $\mathcal{O}(n/\log n)$ Elementen der Ausgabe, angefangen beim letzten, drehen dieses um und geben es aus. □

Kor. Die topologische Sortierung eines azyklischen Graphen, der mit Adjazenzarrays repräsentiert wird, kann in $\mathcal{O}((n+m)\log n)$ Zeit und mit $\mathcal{O}(n \log\log n)$ Bits an Speicher berechnet werden.

Bezeichnung. Wir nennen die Aufrufe von preProcess() und postProcess() und Aufrufe, denen ein Aufruf einer dieser Prozeduren vorausgeht oder folgt, *Hauptaufrufe*.

Bem. Während der Tiefensuche eines Graphen mit n Knoten passieren maximal $6n$ Hauptaufrufe.

Lem. Sei G ein gerichteter Graph mit n Knoten und m Kanten. Eine umgekehrte Folge von (symbolischen Repräsentationen von) Hauptaufrufen während einer Tiefensuche von G kann mit $\mathcal{O}(n \log\log n)$ Bits an Speicher und in

$$\begin{cases} \mathcal{O}((n+m)\log n) & \text{falls } G \text{ mit Adjazenzlisten repräsentiert wird,} \\ \mathcal{O}(n \log\log n) & \text{falls } G \text{ mit Adjazenzarrays repräsentiert wird} \end{cases}$$

Zeit berechnet werden.

Beweis. Wir teilen die Ausführung einer Tiefensuche auf G in $\mathcal{O}(\log n)$ *Epochen* auf, in denen jeweils $\mathcal{O}(n/\log n)$ Hauptaufrufe stattfinden. Wir führen dann jede Epoche, beginnend bei der letzten erneut aus, wobei wir die Hauptaufrufe mitloggen und danach in umgekehrter Reihenfolge ausgeben. Es bleibt noch zu klären, wie wir den Zustand des Stacks zu Beginn einer Epoche wiederherstellen können. Dazu speichern wir für jede Epoche den obersten Knoten H auf dem Stack sowie den tiefsten Knoten $\hat{H}$, der während der Epoche verändert wird. Dann brauchen wir bloß den Teil des Stacks ab $\hat{H}$ bis H rekonstruieren. Dazu müssen wir noch wissen, welche Farbe ein jeder Knoten zu Beginn einer jeden Epoche besitzt. Dies können wir uns merken, indem wir für jeden Knoten speichern, in welcher Epoche er grau und in welcher er schwarz geworden ist. □

Kor. Eine topologische Sortierung eines gerichteten azyklischen Graphen mit n Knoten und m Kanten, der mit Adjazenzarrays repräsentiert wird, kann in $\mathcal{O}(n+m)$ Zeit und $\mathcal{O}(n \log\log n)$ Bits berechnet werden.

Algorithmus (Starke Zshgskomponenten durch TS)**.**
Führe eine Tiefensuche S auf G durch. Führe dann eine Tiefensuche $\overleftarrow{S}$ auf dem Graph $\overleftarrow{G}$ durch, wobei man neue Tiefensuchbäume bei dem Knoten beginnt, bei dem S als letztes fertig wurde, und gebe alle Knoten aus. Wenn $\overleftarrow{S}$ einen Tiefensuchbaum abschließt, so endet auch eine starke Zshgskomponente.

Thm. Die starken Zusammenhangskomponenten eines gerichteten Graphen mit n Knoten und m Kanten, der mit Aus- und In-Adjazenzarrays repräsentiert wird, kann in $\mathcal{O}(n+m)$ Zeit mit $\mathcal{O}(n \log\log n)$ Bits an Speicher berechnet werden.

Bem. Die Ausgabe ist dabei eine Liste von Knoten, in einer beliebigen Reihenfolge, gruppiert nach starken Zshgskomponenten. Es ist kein Algorithmus mit gleichen Platz- und Zeitanforderungen bekannt, der die Knoten sortiert zusammen mit der Nummer der Zshgskomponente ausgibt.

Folgerung. Im platzbeschränkten Setting ist es wichtig, genau zu spezifizieren, wie die Ausgabe eines Algorithmus auszusehen hat. Die Ausgabe ist dabei häufig ein *Stream*, eine Liste, die kontinuierlich produziert wird. Sie muss oft direkt weiterverarbeitet werden, weil man nicht den Platz hat, ihn zu speichern.