Zusammenfassung Platzeffiziente Alg.

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

Basierend auf dem Skript zur gleichnamigen Vorlesung von Prof. Dr. Torben Hagerup an der Universität Augsburg im WS15/16.

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

Erreichbarkeit in Graphen

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

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 Alg. löst Erreichbarkeit mit $\mathcal{O}((\log n)^2)$ Bits.

Bem. Die Laufzeit von Savitch's Alg. ist allerdings sehr schlecht, im schlechtesten Fall (z. B. bei einer verketteten Liste) $n^{\Theta(\log n)}$.

Die Repräsentation von Graphen

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

Tiefensuche

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 grauen 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 von allen 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 u weniger ausgehende Kanten besitzt, so speichern wir nur die $\mathcal{O}(\log\log n)$-Bit-Approximation

$$f_u := \lfloor \tfrac{k-1}{g_u} \rfloor, \quad \text{mit} \quad g_u := \lceil \tfrac{\deg(u)}{\ell} \rceil, \ \ell := \Theta(\log n).$$

Anwendungen von Tiefensuche

Topologisches Sortieren

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 gericht. Graph mit n Knoten und m Kanten. Ein Stream der Hauptaufrufe bei einer Tiefensuche von G in umgekehrter Reihenfolge 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.

Starke Zusammenhangskomponenten

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.

Zshgskomponenten und Breitensuche

Bem. Die Zshgskomponenten eines ungerichteten Graphen kann mit Tiefensuche berechnet werden. Mit anderen Explorierungsstrategien sind jedoch effizientere Algorithmen möglich.

Algorithmus (Connected Components mit Lücken)**.**

```
1:  function CC(G)
2:      k := 0
3:      for u := 1 to n do
4:          color[u] := white
5:      for u := 1 to n do
6:          if color[u] ≠ white then
7:              continue
8:          k := k + 1
9:          while at least one vertex is gray do
10:             Let v be a gray vertex
11:             OUTPUT(v, k)
12:             for all neighbors w of v do
13:                 if color[w] = white then
14:                     color[w] := gray
15:             color[v] := black
```

Bem. Um diesen Algorithmus zu implementieren, benötigt man eine Datenstruktur, die sich die grauen Knoten merkt:

Choice Dictionaries

Def. Ein Choice Dictionary der Größe n ist eine Datenstruktur, welche eine (initial leere) Teilmenge $I \subset [n] := \{1, \dots, n\}$ verwaltet und folgende Operationen unterstützt:

Aufruf	Resultat
INSERT(i)	Füge $i \in [n]$ zu I hinzu
DELETE(i)	Nehme $i \in I$ aus I heraus
ISMEMBER(i)	Prüfe, ob $i \in [n]$ in I enthalten ist
SOMEID( )	Gebe ein beliebiges Element von I zurück
ALLIDS( )	Gebe alle Elemente von I zurück (als Stream)

Lem (CD_1)**.** Man kann ein Choice-Dictionary der Größe n, welches die ersten vier Operationen in konstanter Zeit und ALLIDS in Zeit $\mathcal{O}(|I|+1)$ unterstützt, mit $n \cdot (1 + 2 \cdot \lceil \log_2 n \rceil)$ Bits realisieren.

Beweis. Mit einem n-Bit-Array merken wir uns für jede Zahl $i \in [n]$, ob sie in I enthalten ist. Wir verwenden außerdem zwei Arrays A und B bestehend aus je n Einträgen à $\lceil \log_2 n \rceil$ Bits und einen Zähler $k \in \{0, \dots, n\}$, welcher die Größe von I speichert. In A sind die ersten k Einträge beliebige Elemente aus I. Das Array B wird verwendet, um für jedes $i \in [n]$ zu merken, an welcher Stelle es in A vorkommt (falls überhaupt $i \in I$ gilt). Anders ausgedrückt:

$$\forall\, i \in I \,:\, A[B[i]] = i, \quad \forall\, 0 \le j < k = |I| \,:\, B[A[j]] = j.$$

Folgende Invariante ermöglicht zu prüfen, ob $i \in I$:

$$\forall\, i \in [n] \,:\quad i \in I \iff 0 \le B[i] < k \wedge A[B[i]] = i. \qquad \square$$

Technik. Sei $f : A \to B$ eine Funktion, A endlich. Wenn f in einem Programm häufig aufgerufen wird, so kann es günstiger sein, die Werte von f im Voraus (beim Kompilieren oder Initialisieren) zu berechnen und im einem Array, der **Lookup-Table**, zu speichern. Werte können dann einfach nachgeschlagen werden.
Die Voraussetzung dafür ist, dass $|A|$ relativ klein ist und die Werte von f wenig Speicher benötigen.
Genauer: Angenommen, für einen Parameter $n \in \mathbb{N}$ gilt, dass

- Elemente von A binär mit $m \le 1/2 \log_2 n$ Bits repräsentiert werden können (folglich $|A| \le 2^{1/2 \log_2 n} = \sqrt{n}$),
- f in polynomieller Zeit in m auf binär repräsentierten Elementen von A ausgewertet werden kann,
- die Werte von f polynomiell in m viel Platz benötigen.

Dann kann eine Lookup-Table in $\sqrt{n}(\log_2 n)^C$ Zeit berechnet und in $\sqrt{n}(\log_2 n)^D$ Bit gespeichert werden.
Durch Verwendung dieser Technik kann man Funktionen mit kleinen Argumenten als in konstanter Zeit auswertbar betrachten.

Bspe. Lookup-Tables können verwendet werden, um folgende Funktionen zu berechnen:

- Sinus (approx.)
- Anzahl/Position der 1-Bits in einem Byte

Lem (CD_2)**.** Es gibt ein Choice-Dictionary, welches $\mathcal{O}(n)$ Platz benötigt, die ersten vier Operationen in konstanter Zeit und ALLIDS in Zeit $\mathcal{O}(|I|+1)$ unterstützt.

Beweis. Wir merken uns in einem Array V von n Bit, welche Elemente in I enthalten sind. Wir teilen die Menge $[n]$ in $\mathcal{O}(n/\log n)$ Gruppen, d. h. disjunkte zusammenhängende Teilmengen, zu je $\le 1/2 \log_2 n$ ganzen Zahlen auf. Sei I^* die Menge aller Gruppen, die Elemente aus I enthalten. Da die Zahlen jeder Gruppe nur höchstens zwei Maschinenworte im Bitarray V umfassen, können wir in konstanter Zeit prüfen, ob eine Gruppe Zahlen in I enthält, also ob die Gruppe in I^* liegt. Um schnell Gruppen aus I^* zu finden, verwenden wir zwei Arrays A und B zu je $\mathcal{O}(n/\log n \cdot \log(n/\log n)) = \mathcal{O}(n)$ Bits wie im Beweis von CD_1. Wir implementieren SOMEID wie folgt: Zunächst ist $G := A[0]$ eine nichtleere Gruppe. Sei q das zugehörige Wort in V. Sei f eine Funktion, welche für ein Maschinenwort den Index eines 1-Bits im Wort zurückgibt. Dann ist $(\min G) + f(q) \in I$. Wir benutzen eine Lookup-Table, um f effizient auszuwerten. □

Thm (CD_3)**.** Es gibt ein Choice-Dictionary, welches $n + \mathcal{O}(n/\log n)$ Bits an Platz benötigt, die ersten vier Operationen in konstanter Zeit und ALLIDS in Zeit $\mathcal{O}(|I|+1)$ unterstützt.

Beweis. Wir verwenden diesselbe Konstr. wie im Beweis von CD_2, speziell das Bitarray V mit n Bits und die Einteilung in Gruppen. Um $I^* \subseteq [m]$ mit $m = \mathcal{O}(n/\log n)$ zu verwalten, gebrauchen wir das Choice-Dictionary von CD_2, welches nur $\mathcal{O}(m)$ Bits benötigt. □

Kor. Man kann die Zshgskomponenten eines ungerichteten Graphen in Zeit $\mathcal{O}(n+m)$ mit $2n + \mathcal{O}(n/\log n)$ Bits berechnen.

Bem. Man benötigt sogar nur $cn + \mathcal{O}(1)$ Bits für belieb. $c > \log_2 3$.

Eine untere Schranke für Choice Dictionaries

Def. Ein Choice-Dictionary D heißt **systematisch**, falls es (nach der Initialisierung von D) n von D verwaltete Bits $z_1, \dots, z_n$ gibt, sodass $z_i = 1 \iff i \in I$ für alle $i = 1, \dots, n$ ab dem ersten Schreiben von z_i durch D gilt.

Def.
- Ein **Rot-Blau-Baum** ist ein Binärbaum, dessen inneren Knoten alle entweder rot oder blau gefärbt sind.
- Die **blaue Tiefe** eines Blattes ist die Zahl der blauen Vorfahren.
- Die **rechts-rote (rr) Tiefe** eines Blattes ist die Anzahl der Vorfahren, die rechtes Kind eines roten Knotens sind.
- Ein (d, r, s)**-Baum** ist ein Rot-Blau-Baum, in dem jedes Blatt Tiefe $= d$, blaue Tiefe $\leq s$ und rechts-rote Tiefe $\leq r$ besitzt.

Notation. $N(d, r, s) :=$ max. Anzahl Blätter eines (d, r, s)-Baums

Lem. $N(d, r, s) =$

$$\begin{cases} 0 & \text{wenn } \min\{d, r, s\} < 0, \\ 1 & \text{wenn } d = 0 \text{ und } r, s \geq 0, \\ \max\left\{ \begin{array}{l} 2N(d-1, r, s-1), \\ N(d-1, r, s) + N(d-1, r-1, s) \end{array} \right\} & \text{wenn } d > 0 \text{ und } r, s \geq 0. \end{cases}$$

Lem. $N(d, r, s) \leq 2^{s'} \binom{d - s' + r}{r}$ wobei $s' := \min\{d, s\}$

Thm. Seien $n, s, t \in \mathbb{N}$. Sei D ein systematisches Choice-Dictionary, das nach dem Initialisieren neben $z_1, \dots, z_n$ weitere s Bits benutzt. Angenommen, für $0 \leq r \leq n$ gibt es eine Berechnung C_r, die alle Teilmengen $A \subset I$ mit $|A| = r$ unterscheiden kann und dabei $\leq rt$ Bits der internen Repräsentation von D liest. Dann gilt $st \geq \frac{n}{2}$.

Kor. Seien $n, s \in \mathbb{N}$ und D ein systematisches Choice-Dictionary, welches nach dem Initialisieren $n + s$ Bits an Speicher belegt. Seien t_{choice} und t_{delete} die maximalen Anzahlen an Bits, die bei einer *choice*- bzw. *delete*-Operation gelesen werden. Dann gilt $s(t_{\text{choice}} + t_{\text{delete}}) \leq \frac{n}{2}$.

Breitensuche

Def. Sei ein Graph gegeben. Eine allgemeine Suche durch den Graphen geht wie folgt: Es gibt weiße, graue und schwarze Knoten, explorierte und unexplorierte Kanten. Zunächst sind alle Knoten weiß, bis auf einen grauen Startknoten und alle Kanten unexploriert. Solange es noch einen grauen Knoten u gibt, wählen wir einen solchen aus. Hat u keine unexplorierten Kanten, so färben wir u schwarz. Wenn u noch eine unexplorierte Kante (u, v) hat, so *explorieren* wir (u, v), d. h. falls v weiß ist, so färben wir v grau und rufen eine Benutzerfunktion mit (u, v) auf.

Bem. Verschiedene Suchstrategien unterscheiden sich darin, wie u gewählt wird.

Def (Breitensuche). Drei Varianten:

A. u ist der älteste graue Knoten

B. Der graue Knoten u aus der vorherigen Iteration wird weiterverwendet, falls er noch unexplorierte Kanten besitzt. Andernfalls ist u wie in C.

C. u ist ein grauer Knoten mit minimaler Distanz vom Startknoten.

Thm. Sei ein Graph mit n Knoten und m Kanten gegeben, r ein Startknoten, von dem aus alle anderen Knoten erreichbar sind. Die Variante B der Breitensuche kann in $O(n + m)$ Zeit mit $O(n)$ Bits an Speicher durchgeführt werden.

Idee. Verwende zwei Choice-Dictionaries, um graue Knoten zu speichern: Eines für alle Knoten in Entfernung d, eines für alle Knoten in Entfernung $d + 1$ vom Startknoten.