Seminar Algorithm Engineering

Externe Breitensuche

Die Nadel im Heuhaufen mit zu wenig Platz in der Scheune

Robert Jakob*

Sommersemester 2008
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Führt man eine Breitensuche in Graphen, welche nicht mehr komplett in den Speicher passen, mit dem normalen Algorithmus zur Breitensuche aus, dann führt das zu einem massiven Laufzeitanstieg durch die notwendigen Auslagerungen von Speicher zu Festplatte. Diese Laufzeitanstiege liegen hauptsächlich am unstrukturierten Zugriff auf die Adjazenzlisten (Kanten) und dem platzaufwändigen und somit zeitaufwändigen Speichern von schon besuchten Knoten. Hier wird ein Ansatz von Munagala und Ranade beschrieben, welche sich durch ein geschicktes Verfahren die schon besuchten Knoten implizit merken und somit keine Liste der besuchten Knoten führen müssen. Der Ansatz von Mehlhorn und Meyer, welcher auf dem von Munagala und Ranade aufsetzt, versucht das Adjazenzlistenproblem durch Partitionierung der Knoten zu lösen. Hierbei wird das Lokalitätskriterium ausgenutzt, was bedeutet, das Knoten welche nahe zusammenliegen, auch kurz darauf gelesen werden und somit auch schon geladen werden können.

1 Einführung

In vielen Anwendungsbereichen werden Graphen als Datenstruktur benutzt. Wir nutzen hier folgende Definition: Ein Graph G ist ein Tupel (V, E) mit Knoten V und einer Menge von Kanten E. Es ist $E \subseteq V \times V$ und $|V| = n$. Wir betrachten nur ungerichtete Graphen, d.h. $(v_i, v_j) \in E \Rightarrow (v_j, v_i) \in E$ mit $v_i, v_j \in V$. Weiterhin gilt: Ein Knoten v befindet sich in Ebene i bezüglich eines Startknotens s wenn gilt $\min\{\text{dist}(s, v)\} = i$.

Werden diese Graphen nun sehr groß, wie das beispielsweise bei Webgraphen [NW01], Abbildungen von sozialen Netzwerken oder Routenplanung auf Geräten mit wenig Speicher [GW05] auftritt, dann kann es sein, dass der Graph in seiner gesamten Größe nicht mehr in den Speicher passt. Die Suche in einem Graphen - hier wird die Breitensuche betrachtet - ist dann mit dem Algorithmus wie er normalerweise verwendet wird, nicht möglich. Dieser führt, bei über den Speicher hinausragenden Graphen, zu stark ansteigenden Laufzeiten, da die nicht in den Speicher passenden Teile des Graphen ausgelagert und später wieder geladen werden müssen. Da der Zugriff auf den Speicher (ca. 5ns) im Vergleich zur Festplatte (ca. 10ms) wesentlich schneller ist, müssen hier Algorithmen verwendet werden, welche an dieses Problem angepasst sind.

Der Aufbau dieser Arbeit ist wie folgt: In Abschnitt 2 werden die normale Breitensuche und die damit auftretenden Probleme betrachtet. In Abschnitt 3 wird ein Ein/Ausgabe-Modell, auf dessen Grundlage die verbesserten Algorithmen arbeiten, vorgestellt. In den Abschnitten 4 und 5 werden die verbesserten Algorithmen selbst betrachtet und schließlich in Abschnitt 6 Ergebnisse der Experimente mit den neuen und alten Algorithmen gezeigt.

2 Normale Breitensuche

Unter der normalen Breitensuche verstehen wir eine Breitensuche, welche die verschiedenen Speicherarten nicht berücksichtigt und implizit davon ausgeht, dass der komplette Graph in den Speicher passt. Wir bezeichnen sie, da sie davon ausgeht, dass der Graph in den internen Speicher passt, als internal memory breadth first search, kurz `IM_BFS`.

Die Breitensuche `IM_BFS`, wie sie beispielsweise in [MS04] beschrieben ist, hier als Pseudocode beschrieben:

IM_BFS(G, s)
for each vertex $x \in V\{s\}$ **do**
  $visited[x] \leftarrow 0$
end for
$TreeEdges \leftarrow null$
$Q \leftarrow \emptyset$
$visited[s] \leftarrow 1$
enqueue(Q, s)

*rj@cs-kb.net

while $Q \neq \emptyset$ **do**
  $v \leftarrow$ **dequeue**(Q)
  for each $w \in Adj[v]$ **do**
    if $visited[w] = 0$ **then**
      $visited[w] \leftarrow 1$
      $TreeEdges \leftarrow TreeEdges \cup \{(v, w)\}$
      enqueue(Q,w)
    end if
  end for
end while

Die Breitensuche `IM_BFS` hat eine Laufzeit $\mathcal{O}(|V| + |E|)$, da jeder Knoten ($|V|$) besucht und jede Kante ($|E|$) verfolgt werden muss.

Solange der Graph in den Speicher passt funktioniert dieser Algorithmus hervorragend. Bei überschreiten der Speichergröße, steigt die Laufzeit sprunghaft an, wie man in Abbildung 1 sehen kann (in etwa bei $n = 2^{21}$).

Der Laufzeitanstieg ist nach [AMO07] auf (1) die unstrukturierten Zugriffe auf die Adjazenzlisten und (2) das Problem des Speicherns von schon besuchten Knoten, zurückzuführen.

Um diese Probleme zu lösen muss man die verschiedenen Speicherarten im Algorithmus berücksichtigen. Hierzu wird die Speicherhierarchie mit einem Ein/Ausgabe-Modell (kurz E/A-Modell) abgebildet.

3 E/A-Modelle

Das Ein/Ausgabe-Modell, welches hier verwendet wird, ist das externe-Speicher Modell von Aggarwal und Vitter [AV88], welches allgemein akzeptiert ist [AMO07].

In diesem Modell wird ein Computer mit einem Prozessor, einem Speicher der Größe M und einer unbegrenzt großen Festplatte beschrieben. Die Problemgröße wird mit N identifiziert. Weiterhin können in einem einzelnen Vorgang B Datenelemente zwischen Festplatte und Speicher kopiert werden.

Es werden noch Bedingungen an die Werte gestellt: (1) Die Problemgröße sollte nicht in den Speicher passen, da ansonsten die Betrachtung des externen Speichers nicht notwendig ist; (2) Pro Vorgang sollen nur höchsten so viele Daten wie überhaupt in den Speicher passen kopiert werden können; (3) alle Werte sollten natürlich positiv sein. In einer Formel zusammengefasst erhält man: $1 \leq B \leq M \leq N$.

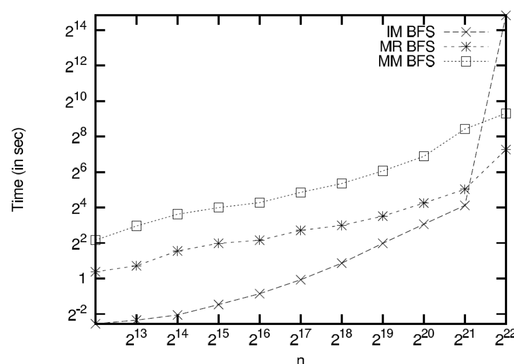

Abbildung 1: Laufzeitunterschiede zwischen den verschiedenen Algorithmen

Ein weiteres mögliches Ein/Ausgabe-Modell wäre das cache-blinde Modell wie es in [FLPR99] beschrieben ist. Es ist mit dem externen-Speicher Modell identisch, mit der Ausnahme, dass die Werte für B und M dem Algorithmus unbekannt sind. Dieses Modell wird jetzt allerdings nicht näher betrachtet.

Im externen-Speicher Modell werden zwei Grundoperationen definiert. Zum einen die Operation Scan(n') welche n' Datenelemente sequentiell zwischen externem und internem Speicher kopiert. Diese Operation hat Ein/Ausgabe-Operationen von $\Theta(\frac{n'}{B})$.

Als zweite Operation wird Sort(n') definiert, welche eine Menge von n' Datenelementen sortiert. Man erhält hier Ein/Ausgabe-Operationen von $\Theta(\frac{n'}{B} \log_{\frac{M}{B}} \frac{n'}{B})$.

4 Algorithmus von Munagala und Ranade

Der Algorithmus von Munagala und Ranade [MR99] versucht das Problem des Speicherns von schon besuchten Knoten zu lösen.

Hierzu wird eine elegante Eigenschaft angewendet: Sei n_i ein Knoten, welcher sich in der i. Ebene des Graphen befindet. Alle Nachbarn von n_i liegen in den Ebenen $i + 1$ und $i - 1$. Mithilfe dieser Eigenschaft kann man die Knoten der Ebene $i + 1$ aus den Knoten der Ebenen i und $i - 1$ berechnen: Man bestimmt alle Nachbarn der Knoten aus der Ebene i. Von dieser Menge entfernt man die Knoten der Ebene i und alle Knoten der Ebene $i - 1$. Man erhält alle Knoten aus der Ebene $i + 1$.

Durch diese einfache Eigenschaft ist es nun nicht mehr notwendig die schon besuchten Knoten zu speichern, da sie implizit gegeben sind.

Sei $N(V)$ die Funktion, die für eine Menge von Knoten V alle Nachbarn ermittelt und sei $L(t)$ die Menge der Knoten, welche in der Ebene t liegen.

Man kann nun den folgenden Algorithmus `MR_BFS` (etwas vereinfacht aus [MR99] und [AMO07]), zur Breitensuche verwenden:

MR_BFS(G, s)
for each vertex $v \in V[G]$ **do**
  $C[V] = 0$
end for
$L(-1) = L(-2) = \emptyset$
$t = c = 0$
while $L(t-1) \neq \emptyset$ or unvisited vertices exist **do**
  if $L(t-1) \neq \emptyset$**then**
    $L(t-1) =$ next unvisited vertex
    $c = c + 1$
  else
    $A(t) = N(L(t-1))$
    Remove duplicates from $A(t)$
    $L(t) = A(t) \setminus (L(t-1) \cup L(t-2))$

```
      for each vertex v ∈ L(t) do C[u] = c
    end if
    t = t + 1
end while
```

Wir erhalten nach [MM02] mit diesem Algorithmus eine Laufzeit von $\mathcal{O}(|V| + \text{sort}(|V| + |E|))$ Ein/Ausgabe-Operationen bzw. Zugriffe.

Beweis Die Zugriffe auf den externen Speicher erfolgen beim Ermitteln der Knoten aus der nächsten Ebene, d.h. von Ebene i aus Ebene $i-1$. Diese Operation benötigt für das Bestimmen der Nachbarn Zugriffe von $\sum_t |N(L(t))| = \mathcal{O}(|E|)$ und für das Lesen der Knoten Zugriffe von $\sum_t |L(t)| = \mathcal{O}(|V|)$. Für $A(t) = N(L(t-1))$ müssen also $\mathcal{O}(|V| + \text{scan}(|E|))$ Zugriffe durchgeführt werden, davon $|V|$ unstrukturierte Adjazenzlistenzugriffe und $\text{scan}(|E|)$ Zugriffe auf die Knoten der Nachbarn, da diese sequentiell im Speicher liegen (siehe Abschnitt 3).

Jetzt müssen nach Ermittlung der Nachbarn $A(t)$ die Duplikate entfernt werden. Das erfordert pro Durchlauf $\mathcal{O}(\text{sort}(|A(t)|))$ Zugriffe, da die Mengen sortiert werden müssen um die Duplikate zu entfernen (für sort siehe Abschnitt 3). Da für die gesamte Breitensuche dies für jede Ebene gemacht werden muss, erhält man für den Gesamtaufwand (alle Schleifendurchläufe) $\mathcal{O}(\text{sort}(|E|))$ Zugriffe.

Jetzt werden aus den Nachbarn die Knoten der vorhergehenden Ebene entfernt: $L(t) = A(t) \setminus (L(t-1) \cup L(t-2))$. Da hier eine Sortierung vorliegt erhalten wir für jeden Durchlauf Zugriffe von $\mathcal{O}(\text{scan}(|A(t)| + |L(t-1)| + |L(t-2)|))$.

Alles zusammengenommen erhalten wir somit als obere Grenze Zugriffe von $\mathcal{O}(|V| + \text{sort}(|E|))$. Zusätzlich muss nun noch der BFS-Baum für die Suche erstellt werden, was $\mathcal{O}(\text{sort}(|V| + |E|))$ Zugriffe kostet.

Als Gesamtaufwand erhalten wir damit $\mathcal{O}(|V| + \text{sort}(|V| + |E|))$ Ein/Ausgabe-Operationen.

5 Algorithmus von Mehlhorn und Meyer

Der im vorigen Abschnitt beschriebene Algorithmus von Munagala und Ranade versucht das Problem des Speicherns schon besuchter Knoten zu lösen, allerdings besteht das Problem der unstrukturierten Adjazenzlistenzugriffe noch immer.

Mehlhorn und Meyer [AMO07, MM02] versuchen nun dieses Problem zu lösen, indem die Knoten zu Gruppen zusammengefasst werden. Die Knoten werden so gruppiert, dass folgende Eigenschaft genutzt werden kann: Wenn ein Knoten einer Gruppe besucht wird, dann werden die anderen Knoten derselben Gruppe in wenigen Schritten darauf auch besucht. Aus diesem Grund kann man die Informationen, welche die anderen Knoten in der Gruppe betreffen, beim Zugriff auf den externen Speicher auch laden.

Der Algorithmus von Mehlhorn und Meyer besteht aus zwei Teilen: (1) einem Vorverarbeitungsschritt, der die Knoten gruppiert und (2) der tatsächlichen Breitensuche.

Bei der Gruppierung wird der Graph in Teilgraphen $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$ unterteilt. Für die Bestimmung der Teilgraphen S_i wird von [MM02] ein randomisierter und ein deterministischer Ansatz vorgeschlagen.

5.1 Randomisierter Ansatz

In der randomisierten Variante werden auf zufällige Weise k Hauptknoten s_i gewählt, welche zur Gruppierung benötigt werden. Nach [MM02] wird ein Knoten mit einer Wahrscheinlichkeit von $\mu = \min\{1, \sqrt{\frac{|V|+|E|}{|E|B}}\}$ zum Hauptknoten. B ist die Blockgröße einer Übertragung (siehe Abschnitt 3). μ ist so gewählt, dass die Gesamtkosten des Algorithmus minimiert werden. (Für genaueres siehe [MM02]). Für die Anzahl der Hauptknoten ergibt sich ein Erwartungswert von $\mathbb{E}[k] = \mathcal{O}(\mu n)$.

Nun werden parallel, ausgehend von jedem Hauptknoten s_i, k Breitensuchen gestartet: In jeder Runde versucht jeder Hauptknoten s_i alle noch nicht besuchten Nachbarknoten seines Teilgraphen S_i zu besuchen. Versuchen mehrere Hauptknoten denselben Knoten v zu besuchen, und damit zu ihrem Teilgraphen hinzuzufügen, dann entscheidet der Zufall, wer von ihnen den Knoten v erhält. Die Breitensuchen enden, wenn es keine unbesuchten Knoten mehr gibt.

Damit ergeben sich Gruppen, welche den Graphen in disjunkte Teilgraphen $\mathcal{S}_i$ zerlegen.

5.2 Deterministischer Ansatz

Die deterministische Variante wird hier nur skizziert, zumal sie in [AMO07, MM02] nur als Alternative aufgeführt wird und der randomisierten Variante mehr Gewicht gegeben wird.

Es wird ein Spannbaum $\mathcal{T}_s$ über die Knoten des Graphen inklusive Startknoten s ermittelt. Jetzt wird eine Eulertour entlang des Spannbaumes $\mathcal{T}_s$ berechnet und die doppelten Knoten entfernt. Die Knoten werden in Gruppen zerlegt und ergeben dann die Knoten der Teilgraphen $\mathcal{S}_i$.

5.3 Modifizierte Breitensuche

Nachdem die Teilgraphen $\mathcal{S}_i$ ermittelt wurden, werden die Adjazenzlisten entsprechend der Teilgraphen gruppiert: Es ist $\mathcal{F}_i$ die Menge aller Adjazenzlisten für die Knoten in $\mathcal{S}_i$. Entsprechend der Gruppen werden die Adjazenzlisten in $\mathcal{F} = \mathcal{F}_1\mathcal{F}_2\mathcal{F}_3 \ldots \mathcal{F}_k$ unterteilt.

Als Breitensuchalgorithmus wird eine modifizierte Variante des Algorithmus von Munagala und Ranade (Abschnitt 4) genutzt. Die Ebenen werden wie in MR_BFS beschrieben erstellt.

Zusätzlich wird, zur Ausnutzung der Gruppierungen, eine Hot Adjacency List $\mathcal{H}$ vorgehalten, welche die Adjazenzlisten der aktuellen Ebene enthält. Somit liegen die

benötigten Adjazenzlisten (meistens) vor und müssen nur bei einem Ebenenwechsel geladen werden.

Als Laufzeiten erhält man nun für die randomisierte Variante von MM_BFS Ein/Ausgabe-Operationen von

$$\mathcal{O}\left(\sqrt{\frac{|V|\cdot(|V|+|E|)\cdot\log|V|}{B}}+\mathrm{sort}(|V|+|E|)\right)$$

und für die deterministische Variante

$$\mathcal{O}\left(\sqrt{\frac{|V|\cdot(|V|+|E|)}{B}}+\mathrm{sort}(|V|+|E|)+\mathrm{ST}(|V|,|E|)\right)$$

Hierbei ist $\mathrm{ST}(|V|,|E|)$ die Laufzeit um einen Spannbaum aufzubauen (siehe [ABT00]). Für einen Beweis dieser Laufzeiten sei auf [MM02] verwiesen.

6 Experimentelle Ergebnisse

Wie man in Abb. 1 sieht, sind die Laufzeiten von MR_BFS und MM_BFS nach Überschreiten der internen Speichergröße deutlich besser. Passt der Graph noch in den Speicher, ist IM_BFS natürlich die erste Wahl, da kein Mehraufwand betrieben wird.

In Tabelle 1 ist der direkte Vergleich von MR_BFS und MM_BFS abgebildet. In fast allen Fällen ergibt sich ein Laufzeitvorteil für MM_BFS. Zu Beachten sind vor allen Dingen die Laufzeiten von MR_BFS: Eine Laufzeit von 4000 Stunden entspricht ca. 166 Tagen!

Betrachtet man Tabelle 2, welche die 2 Phasen (Gruppierung und Breitensuche) des MM_BFS vergleicht, so sieht man, dass die tatsächliche Breitensuche selbst deutlich mehr Aufwand mit sich bringt.

Verwendet man, als Beispiel aus der Praxis, einen Webgraphen, und vergleicht die Ergebnisse von MR_BFS und MM_BFS, so ergeben sich die in Tabelle 3 angegebenen Warte-/Laufzeiten.

7 Ergebnis

Die Algorithmen MR_BFS und MM_BFS brachten eine deutliche Verbesserungen der Laufzeiten (Stunden im Vergleich zu Monaten) und sind damit ein schönes Beispiel für angewandten Algorithmenentwurf.

Quellen

[ABT00] Arge, Lars, Gerth Brodal und Laura Toma: *On External-Memory MST, SSSP, and Multi-way Planar Graph Separation.* In: *SWAT '00: Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, Seiten 433–447, London, UK, 2000. Springer-Verlag.

[AMO07] Ajwani, Deepak, Ulrich Meyer und Vitaly Osipov: *Improved external memory BFS implementations.* In: *ALENEX/ANALCO*, Seiten 3–12. SIAM, 2007.

[AV88] Aggarwal, Alok und Jeffrey S. Vitter: *The input/output complexity of sorting and related problems.* Commun. ACM, 31(9):1116–1127, 1988.

[FLPR99] Frigo, Matteo, Charles E. Leiserson, Harald Prokop und Sridhar Ramachandran: *Cache-Oblivious Algorithms.* In: *FOCS*, Seiten 285–298, 1999.

[GW05] Goldberg, Andrew V. und Renato Fonseca F. Werneck: *Computing Point-to-Point Shortest Paths from External Memory.* In: Demetrescu, Camil, Robert Sedgewick und Roberto Tamassia (Herausgeber): *ALENEX/ANALCO*, Seiten 26–40. SIAM, 2005.

[MM02] Mehlhorn, Kurt und Ulrich Meyer: *External-Memory Breadth-First Search with Sublinear I/O.* In: *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, Seiten 723–735, London, UK, 2002. Springer-Verlag.

[MR99] Munagala, Kameshwar und Abhiram Ranade: *I/O-complexity of graph algorithms.* In: *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, Seiten 687–694, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[MS04] Mehta, Dinesh P. und Sartaj Sahni: *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.).* Chapman & Hall/CRC, 2004.

[NW01] Najork, Marc und Janet L. Wiener: *Breadth-first search crawling yields high-quality pages*, 2001.

Graph class	n	m	MR_BFS I/O wait time	MR_BFS Total time	MM_BFS I/O wait time	MM_BFS Total time
Random	2^{28}	2^{30}	2.4	3.4	7.3	9.6
Grid	2^{28}	$\sim 2^{29}$	4733.0	4736.0	52.1	53.9
B-level random	2^{28}	2^{30}	3989.8	3994.8	42.2	49.7
B-level spider web	2^{28}	$\sim 2^{29}$	3364.2	3366.5	36.5	39.8
MM Worst	2^{25}	$\sim 2^{25}$	25.2	25.4	19.5	32.4
Simple line	2^{28}	2^{28}	0.6	10.2	84.8	275.9
Rand line	2^{28}	2^{28}	4156.2	4167.7	100.2	283.3
B-interleaved line	2^{28}	2^{28}	4210.3	4222.6	97.5	280.8

Tabelle 1: Wartezeiten für Festplattenzugriffe und Laufzeiten (in Stunden) von MR_BFS und MM_BFS

Graph class	n	m	MM_BFS Phase 1 I/O wait time	MM_BFS Phase 1 Total time	MM_BFS Phase 2 I/O wait time	MM_BFS Phase 2 Total time
Random	2^{28}	2^{30}	3.7	5.1	3.6	4.5
Grid	2^{28}	$\sim 2^{29}$	6.6	7.3	45.5	46.6
B-level random	2^{28}	2^{30}	3.6	5.1	38.6	44.6
B-level Spider Web	2^{28}	$\sim 2^{29}$	6.6	7.3	29.9	32.5
MM Worst	2^{25}	$\sim 2^{25}$	6.5	6.7	13.0	25.7
Simple line	2^{28}	2^{28}	84.3	85.1	0.5	190.8
Rand line	2^{28}	2^{28}	79.4	80.6	20.8	202.7
B-interleaved line	2^{28}	2^{28}	79.3	80.4	18.1	200.4

Tabelle 2: Wartezeiten und Laufzeiten der 2 Phasen von MM_BFS (Gruppierung und Breitensuche) in Stunden

	MR_BFS I/O wait time	MR_BFS Total time	MM_BFS - common μ I/O wait time	MM_BFS - common μ Total time	MM_BFS - Graph dep μ I/O wait time	MM_BFS - Graph dep μ Total time
Single disk	3.7	4.0	7.4	9.4	6.3	8.4
Multiple disk	2.0	2.3	2.7	4.8	2.3	4.5

Tabelle 3: Warte- und Laufzeit (in Stunden) von MR_BFS und MM_BFS. MM_BFS einmal mit allgemeinem μ und graphabhängigem μ.