Kürzeste-Wege-Berechnung bei sehr großen Datenmengen

Andreas Crauser, Kurt Mehlhorn und Ulrich Meyer

Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken
umeyer@mpi-sb.mpg.de

Betreuer: Prof. Dr. K. Mehlhorn

Kurzfassung: In diesem Report untersuchen wir die Ein/Ausgabe-Komplexität (I/O Komplexität) des Kürzesten-Wege-Problems mit einem Startknoten (single source shortest path) auf Graphen mit nicht-negativen Kantengewichten. Wir präsentieren einen Algorithmus, der für eine große Klasse zufälliger Graphen eine I/O Komplexität von $\mathcal{O}(\frac{n}{D} + \frac{m}{DB} \log_{S/B} \frac{m}{B})$ erreicht. Dabei bezeichnen n, m die Anzahl der Knoten bzw. Kanten im Graphen, S ist die Größe des verfügbaren Internspeichers, B bezeichnet die Größe eines Blocktransfers und D ist die Anzahl der unabhängigen parallelen Harddisks; D ist beschränkt auf $\mathcal{O}(\sqrt{n/d})$. Weiterhin präsentieren wir ein effizientes Phasen-Verfahren für Probleminstanzen, die so groß sind, daß selbst ein boolsches Feld für die Knotenmenge nicht mehr im Hauptspeicher gehalten werden kann.

Schlüsselwörter: I/O Komplexität, Kürzeste-Wege-Berechnung, Externspeicher, zufällige Graphen

1. Einleitung

Heutige Computer umfassen mehrere Speicherhierarchien. Die beiden wichtigsten Schichten dabei sind Hauptspeicher (oder auch Internspeicher) und Sekundärspeicher, oft auch Externspeicher genannt, der üblicherweise durch Festplatten oder Magnetbänder realisiert ist. Der Datenaustausch zwischen Intern- und Sekundärspeicher *(I/O-Operation)* vollzieht sich in Blöcken der Größe B. Die benötigte Zeit, um auf einen Block im Sekundärspeicher zuzugreifen und ihn in den Hauptspeicher zu übertragen, ist sehr groß: in heutigen Systemen dauert es etwa 100000 mal länger als der direkte Zugriff auf eine Hauptspeicher-Zelle. Es ist wahrscheinlich, daß dieses Verhältnis in der Zukunkt noch kritischer wird.

Folglich erfahren Internspeicher-Algorithmen mit geringer Lokalität der Zugriffsmuster auf den Speicher (und die meisten kombinatorischen Algorithmen sind von dieser Art) einen enormen Performance-Einbruch, sobald sie auf Probleminstanzen angewandt werden, die nicht mehr ganz in den Hauptspeicher passen. Diese Tatsache hat in den letzten Jahren zu einem großen Interesse an effizienten Algorithmen für Externspeicher geführt.

Wir präsentieren einen Externspeicher-Algorithmus für ein fundamentales Graph-Problen: der Berechnung kürzester Wege *(single source shortest path problem)* auf gerichteten Graphen mit nicht-negativen Kantengewichten.

Wir verwenden die folgende Notation, um die asymptotischen Laufzeiten auszudrücken: n bezeichnet die Anzahl der Knoten, m bezeichnet die Anzahl der Kanten, S gibt die Größe des Hauptspeichers wieder, B steht für die Blockgröße, und D bezeichnet die Anzahl der Festplatten, die parallel genutzt werden können. Unseren numerischen Berechnungen (= Abschätzungen der tatsächlichen Laufzeit) legen wir folgende Werte zugrunde: wir nehmen an, daß B 16 Kilobyte umfaßt, daß S etwa 500 Megabyte groß ist, daß die (durchschnittliche) Suchzeit für einen Block bei 8 msec liegt, daß die Transferrate zwischen Sekundär- und Internspeicher 2 Megabyte pro Sekunde beträgt; somit dauert es durchschnittlich 16 msec bis ein nachgefragter Block im Hauptspeicher verfügbar ist.

Kürzeste Wege: $G = (V, E)$ sei ein gerichteter Graph, s sei ein ausgezeichneter (Start-)Knoten von G und c sei eine Funktion, die jeder Kante $(v, w) \in E$ einen nicht-negativen reellen *Kosten*-Wert, $c(v, w)$, zuordnet. Das *Kürzeste-Wege-Problem (single source shortest path problem)* besteht nun darin, für jeden Knoten v, der von s aus erreichbar ist, die Kosten eines billigsten Pfades von s nach v zu ermitteln. (Unter den Kosten eines Pfades verstehen wir die Summe der Kosten seiner Kanten.)

Der theoretisch effizienteste Internspeicher-Algorithmus ist Dijkstra's Algorithmus [9]; auf Externspeicher angewandt, braucht er jedoch $\Omega(m)$ I/O. Wir entwickeln eine Variante von Dijkstra's Algorithmus, die auf zufälligen regulären gerichteten

Graphen[1,2] mit hoher Wahrscheinlichkeit nur $\mathcal{O}(\frac{n}{D} + \frac{m}{DB} \log_{\frac{S}{B}} \frac{m}{B})$ I/O benötigt, wobei bis zu $D = \mathcal{O}(\sqrt{n/d})$ Harddisks parallel genutzt werden können. Dabei nehmen wir an, daß der Hauptspeicher groß genug ist, um einen Bit-Vektor der Größe n zu halten.[3] Im Abschnitt 3 erweitern wir das Ergebnis auf Graph-Instanzen, bei denen wir diese Annahme nicht mehr garantieren können.

Unsere Verbesserungen basieren auf den folgenden Ideen: (1) Wenn wir erlauben, daß die Knoten in der Prioritäts-Warteschlange mehrfach auftreten dürfen, können m I/O-Operationen auf n I/Os reduziert werden. (2) Wir beschreiben ein einfaches Kriterium, das uns sehr häufig erlaubt, in einem Schritt mehr als einen Knoten aus der Prioritäts-Warteschlange zu entfernen. Wir zeigen, daß bei zufälligen gerichteten Graphen mit hoher Wahrscheinlichkeit alle Knoten in $O(\sqrt{nd})$ Phasen[4] entfernt werden können. (3) Der Zugriff auf die Adjazenz-Listen kann durch eine zufällige Verteilung über die D Festplatten "parallelisiert" werden; dabei werden Ideen aus [4] benutzt. (4) Wir beschreiben eine einfache externe Prioritäts-Warteschlange, die auf internen Radix–Heaps [1] beruht. Diese Implementierung unterstützt das Einfügen eines Elements mit $\mathcal{O}(1/(DB))$ I/O und das Löschen des kleinsten Elements mit amortisierten $\mathcal{O}(1/(DB) \cdot \log_{S/(DB \log C)} C)$ I/O, wobei C die maximal auftretende Distanz bezeichnet. Somit wird zwar nicht ganz die asymptotische Leistung von Buffer-Trees [2] erreicht, unsere Datenstruktur hat aber den Vorteil, daß sie viel einfacher ist und ihre Konstanten sehr leicht zu bestimmen sind.

Für unser numerisches Beispiel verwenden wir einen gerichteten Graphen mit $n = 10^9$ Knoten und $n = 10^{11}$ Kanten. In der gebräuchlichen Darstellung dünner Graphen muß für jede Kante zumindest der Index des Zielknotens und ihre Länge (entspricht den Kosten) gespeichert werden. Wir nehmen an, daß Knotenindizes und Kantenlängen jeweils 4 Bytes beanspruchen; bzgl. der Knotenindizes wird dies durch die Größe unseres Graphs diktiert, bzgl. der Kantenlängen erlaubt uns diese Wahl Werte aus den Bereichen *int* oder *float*. 8 Bytes pro Kante summieren sich in unserem Beispiel zu 0.8 Terabytes. Ein Feld für alle Knotendistanzen benötigt 4 Gigabytes und ein boolesches Feld für die Knoten braucht $(1/8)10^9 = 125$ Megabytes. Somit paßt bei $S = 500$ Megabytes ein boolesches Feld für die Knoten noch in den Hauptspeicher, ein Distanzfeld aber nicht mehr. Welche Leistung können wir erwarten?

---

[1] In einem zufälligen regulären Digraph hat jeder Knoten Ausgangsgrad $d = m/n$, die ausgehenden Kanten zeigen auf zufällige Knoten, und die Kantengewichte sind unabhängig und gleichverteilt im Einheitsintervall.

[2] Unser Resultat ist tatsächlich für eine größere Graphklasse gültig, wir verzichten jedoch an dieser Stelle auf eine exakte Definition.

[3] In Anbetracht des augenblicklichen Preisegefüges für Harddisks bzw. Hauptspeicher und der Tatsache, daß Rechner inzwischen mit mehreren Gigabytes RAM betrieben werden können, ist diese Annahme für die meisten Anwendungen realistisch: Festplattenspeicher ist z.Z. etwa um einen Faktor 30 bis 70 billiger als Hauptspeicher, andererseits ist der Speicherbedarf für eine Kante oftmals um einen Faktor 50 bis 100 höher als für ein Knoten-Bit. Schon für kleinen Knotengrad wird klar, daß der Gesamtplattenplatz weitaus teuerer ist, als das RAM für einen Bitvektor.

[4] Simulationen zeigen, daß für $d > 1$ etwa $2.5\sqrt{n}$ Phasen ausreichen.

- Die LEDA Implementierung [12, 13] von Dijkstra's Algorithmus hat eine asymptotische Laufzeit von $O(m + n \log n)$ und benötigt auf einer SPARC 10 etwa 2 Sekunden für einen Graph mit $n = 10^4$ Knoten und $m = 10^6$ Kanten. Wir schließen, daß die CPU Zeit für unser Beispiel mit $n = 10^9$ und $m = 10^{11}$ bei etwa $2\ 10^5$sec ≈ 2.2 Tage liegen wird.
- Der Algorithmus aus 1. macht nicht weniger als m Zugriffe auf den Sekundärspeicher, somit würde seine I/O Zeit gigantische 10^{11}16msec $= 1.6\ 10^9$sec, also etwa 50 Jahre (!!) betragen.
- Für $D = 32$ benötigt unser neuer Algorithmus höchstens 30 Tage.

Speicherplatz-effiziente Wörterbücher: Alle bis hierhin erwähnten Algorithmen gehen davon aus, daß der Hauptspeicher groß genug ist, um einen Bit-Vektor für die Knoten zu halten. Ist diese Annahme nicht erfüllt, folgen wir einem Ansatz von [6] und unterteilen die Berechnung in Runden. In jeder Runde löschen wir etwa S unterschiedliche Knoten aus der Prioritäts-Warteschlange. Sei V_{del} die Menge der verschiedenen Knoten in einer bestimmten Phase, die zum ersten Mal in dieser Phase gelöscht werden. (Die Tatsache, daß ein bestimmter Knoten mehrfach gelöscht werden kann, hängt damit zusammen, daß wir das Einfügen mehrerer Instanzen solcher Knoten in die Prioritäts-Warteschlange erlaubt haben.) Wir halten V_{del} als ein *Wörterbuch*. Am Ende jeder Phase markieren wir alle Kanten in E, die auf Knoten aus V_{del} zeigen. Weiterhin markieren wir alle Einträge in der Prioritäts-Warteschlange, die zu einem Knoten in V_{del} gehören.

Wir können nun erkennen, ob ein Knoten zum ersten Mal gelöscht wurde, indem wir einfach sein Markierungsbit bei seinem Eintrag in der Warteschlange testen. Die zusätzlichen I/O Kosten für jede Phase beschränken sich auf die Zeit für einen *Scan*, also das blockweisen Einlesen der externen Graphdarstellung. Die Anzahl der Phasen ist direkt davon abhängig, wieviele Einträge das Wörterbuch bei der durch S erzwungenen Platzbeschränkung in einer Phase speichern kann. Wir betrachten also das folgende Problem.

Problem: Seien S Bits Hauptspeicher für das Wörterbuch verfügbar; wir wollen eine sich dynamisch vergrößernde Teilmenge X aus $[0 .. n-1]$ verwalten und die Operationen *Einfügen* und *Test auf Mitgliedschaft* unterstützen. Sei r die maximale Größe der Teilmenge X, die wir mit S Bits effizient verwalten können. Das Ziel besteht also darin, r zu maximieren.

Falls $n \leq S$ kann ein Bit-Vektor benutzt werden, und somit ist $r = n$. Für $n > S$ ergibt sich bei *Hashing mit offener Adressierung* $r \geq c \cdot S/\lceil \log n \rceil$ für jede Konstante c kleiner als eins. Man beachte den signifikanten Rückgang bei r von S nach $S/\log n$, sobald n minimal von S auf $(1+\epsilon)S$ anwächst. Wir beschreiben eine Methode, die einen glatten Übergang liefert und dabei immer sehr nahe an der informationstheoretischen unteren Schranke bleibt. Unsere Wörterbuch-Implementierung garantiert $r = \Omega(S/(\log n/S))$. Verglichen mit der theoretischen unteren Schranke erreichen

wir schon Zwei-Optimalität für $n > 1.01 \cdot S$. Auf Kosten angestiegener interner Rechenzeiten, erreichen wir damit bessere I/O-Konstanten als dies unter Verwendung früherer Ansätze[5] möglich wäre.

2. Kürzeste-Wege-Berechnung

2.1. Dijkstra's Algorithmus und mehrfache vorläufige Distanzen.

Die Standard-Implementierung von Dijkstra's Algorithmus benutzt eine Prioritäts-Warteschlange PQ, die für jeden Knoten, dem noch keine endgültige Distanz zugeordnet wurde, einen Eintrag verwaltet. In jeder Iteration wird der Knoten v mit dem kleinsten Distanzwert aus der Warteschlange genommen, seine Distanz als endgültig erklärt, und alle von ihm ausgehenden Kanten werden geladen. Für jede Kante $e = (v, w)$ wird untersucht, ob der augenblickliche Distanzwert von w größer als $dist[v]+c(e)$ ist; falls ja, wird $dist[w]$ entsprechend reduziert. Auf diese Art verursacht jede Kante einen wahlfreien (Externspeicher-)Zugriff (nämlich auf $dist[w]$ und auf den Eintrag von w in PQ), also ergeben sich $\Omega(m)$ I/O-Operationen. Wir werden zeigen, wie wir durch eine Steigerung der CPU-Zeit auf $\mathcal{O}(m \log n)$ die I/O-Operationen auf $\mathcal{O}(n)$ reduzieren können (I/O für Operationen auf PQ klammern wir vorerst aus, sie werden später separat betrachtet.)

Wir speichern einen Bit-Vektor *tentative* für die Knoten aus G, wobei gilt $tentative[v] = true$ gdw. v noch keine endgültige Distanz zugewiesen wurde. Außerdem erlauben wir, daß sich zu einen Zeitpunkt mehrere Instanzen eines Knotens v in PQ befinden können, maximal ein Eintrag für jede auf v zeigende Kante. Wenn ein Knoten v mit Distanz d seine ausgehenden Kanten abarbeitet, wird für jede Kante (v, w) der Knoten w mit Distanzwert $d + c(v, w)$ in PQ eingefügt. Wird ein Paar (v, d) aus PQ entfernt und $tentative[v]$ ist *true*, dann wird $tentative[v]$ auf *false* gesetzt und die Kanten von v eingelesen; ist $tentative[v]$ hingegen *false*, wird nichts getan. Auf diese Weise reagieren wir nur auf das erste Löschen eines Knotens v aus PQ, die Korrektheit des Algorithmus bleibt gewahrt. Es ergibt sich der folgende Pseudo-Code:

```
while (PQ ist nicht leer)
{ v sei der Knoten mit kleinster vorläufiger Distanz d in PQ;
  lösche v aus PQ;
  if (tentative[v])
  { tentative[v]=false;
    für alle Kanten (v,w) :  PQ.insert(w,d+c(v,w));
  }
}
```

Welche Konsequenzen haben diese Änderungen? Wir führen nun m Einfüge- und Lösch-Operationen auf der Prioritäts-Warteschlange aus, aber keine unstrukturierten Zugriffe mehr auf einzelne Knoten, um ihre Distanzwerte zu verringern. Damit verhindern wir, für jede Kante eine I/O-Operation ausführen zu müssen. Es gibt immer

noch unstrukturierte Zugriffe auf das *tentative*-Feld (aber dieses paßt nach Voraussetzung in den Hauptspeicher), und wir müssen ebenfalls auf die Liste der von v ausgehenden Kanten zugreifen, wenn v zum ersten Mal aus PQ entfernt wird. Somit brauchen wir - die Operationen auf PQ ausgenommen - nur noch n wahlfreie Zugriffe auf den Sekundärspeicher.

2.2. Gleichzeitiges Löschen mehrerer Knoten aus PQ.

Wir beschreiben nun ein einfaches Kriterium, anhand dessen wir sehr häufig mehr als nur einen Knoten aus der Warteschlange entfernen können. Für jeden Knoten v in PQ, bezeichnen wir denjenigen Eintrag von v mit der kleinsten Distanz als *aktiv*, die restlichen Vorkommen von v in PQ nennen wir *inaktiv*. Wir interessieren uns in diesem Abschnitt nur für die aktiven Einträge der Knoten.

Seien $v_1, \ldots, v_q$ die aktiven Einträge in PQ, geordnet nach aufsteigenden Distanzwerten. Weiterhin sei

$$L = \min\{\mathit{dist}(v_i) + \mathit{cost}(e)\ ;\ 1 \le i \le q \text{ und } e \text{ ist eine ausgehende Kante von } v_i\}.$$

Mit Kenntnis von L können wir nun feststellen, welche Knoten aus PQ mit Sicherheit entfernt werden dürfen:

LEMMA 1 *Falls $\mathit{dist}(v_i) \le L$ dann ist $\mathit{dist}(v_i)$ die tatsächliche Distanz von s nach v; v kann somit aus PQ gelöscht werden.*

Sei N die Anzahl aktiver Knoten in PQ, deren Distanz kleiner als L ist; N ist mindestens eins und kann maximal q sein, also alle aktiven Einträge in PQ umfassen. Wir zeigen, daß der erwartete Wert von N für reguläre Graphen mit zufälligen Kantengewichten aus dem Bereich $[0 \,..\, 1]$ und Ausgangsgrad d bei $\Omega(\sqrt{q/d})$ liegt.

Sei D der letzte aus PQ entfernte Distanzwert. Wir betrachten einen aktiven Eintrag v_i in PQ. Sei d_i der Distanzwert von v_i und sei weiterhin $X_i = d_i - D$; X_i ist eine Zufallsvariable aus dem Bereich $[0 \,..\, 1]$. Wir untersuchen zunächst die Verteilung von X_i. Genauer werden wir zeigen, daß die Dichtefunktion $f_{X_i}(x)$ eine nicht-wachsende Funktion in x ist. Seien $u_1, \ldots, u_k$ alle Vorkommen von $v = v_i$ in PQ und für alle j, $1 \le j \le k$, sei (w_j, u_j) die Kante, die u_j in die Warteschlange gebracht hat. Weiterhin definieren wir $U_j = \mathit{dist}(w_j) + c(w_j, v) - D$ für alle j, $1 \le j \le k$. Dann ist U_j gleichverteilt in $[0 \,..\, c_j]$, wobei $c_j = \mathit{dist}(w_j) - D + 1$. Das folgt sofort aus der Tatsache, daß die Verteilung von U_j gerade die Verteilung von $c(w_j, v) + (\mathit{dist}(w_j) - D)$ ist, unter der Voraussetzung, daß der Algorithmus den aktuellen Zustand erreicht. Daß der Algorithmus den aktuellen Zustand erreicht, impliziert wiederum $c(w_j, v) \ge \mathit{dist}(w_j) - D$, stellt aber sonst keine Bedingungen an $c(w_j, v)$. Somit ist an diesem Punkt der Ausführung $c(w_j, v)$ gleichverteilt in

$[1 - c_j \mathrel{..} 1]$. Wir sind interessiert an $d_i = \min\{dist(w_j) + c(w_j, v)\ ;\ 1 \le j \le k\}$, deshalb betrachten wir $X_i = \min\{U_j\ ;\ 1 \le j \le k\}$. Also

$$\int_0^x f_{X_i}(s)\ ds = \text{prob}(X_i \le x) = 1 - \prod_j \max(0, c_j - x)/c_j$$

und damit $f_{X_i}(x) = 0$ für $x \ge \min_j c_j$, sowie

$$f_{X_i}(x) = \sum_j (1/c_j) \prod_{l;l \ne j} (c_l - x)/c_l$$

für $x < \min_j c_j$. Wir schließen, daß $f_{X_i}(x)$ eine nicht-wachsende Funktion in x ist.

Die Zufallsvariablen X_i sind unabhängig. Beachte, daß ihre Verteilungen zwar verbunden sind, da der Wert von $dist(w_j) - D + 1$ eine Rolle in der Verteilung aller Knoten z mit $(w_j, z) \in E$ spielt, der tatsächliche Wert von X_i wird aber nur von den Zufallsvariablen $c(w_j, v_i)$ bestimmt. Somit ist die Menge der Variablen, die den Wert von X_i bestimmen unabhängig von der Menge der Variablen, die X_j für $j \ne i$ bestimmen.

Sei $Y_{i,k}$ eine Zufallsvariable, die die Länge der k-ten von v_i ausgehenden Kante angibt, $1 \le k \le d$. Dann gibt $N = |\{i\ ;\ X_i \le X_j + Y_{j,k}$ für alle $j, k\}|$ an, wieviele Knoten wir in einer Phase löschen können.

LEMMA 2 **(Profitable Löschphasen).** *Die Wahrscheinlichkeit, daß aus der Prioritäts-Warteschlange mit q aktiven Knoteneinträgen, jeder mit Ausgangsgrad $d = m/n$, mindestens $N \ge \sqrt{q/d}$ aktive Knoten gelöscht werden können, ist größer als* 3/40.

BEWEIS: Sei E_1 das Ereignis $N \ge \sqrt{q/d}$. Wir definieren eine lineare Ordnung $\prec$ auf den X_i's durch $X_i \prec X_j$ gdw. entweder $X_i < X_j$ oder $X_i = X_j$ und $i < j$. Außerdem sei Z das $\sqrt{q/d}$-kleinste der X_i's. Wir bezeichnen mit E_2 das Ereignis, daß für alle j und k entweder $X_j \ge Z$ oder $Y_{j,k} \ge Z$. Offensichtlich impliziert E_2, daß für alle j und k der Wert $Z \le X_j + Y_{j,k}$ ist und somit E_1 eingetreten ist. Deshalb genügt es, eine untere Schranke für die Wahrscheinlichkeit von E_2 zu finden.

Es gibt nach Definition $\sqrt{q/d}$ Indizes i mit $X_i \le Z$ und somit ist die Wahrscheinlichkeit, daß für diese i's und für alle k die Ungleichung $Y_{i,k} \ge Z$ gilt, gerade durch $(1 - Z)^{\sqrt{q/d} \cdot d}$ gegeben. Also gilt:

$$\text{prob}(E_2) \ge \int_0^1 (1 - s)^{\sqrt{qd}} f_Z(s)\ ds$$

und somit gilt für jede Konstante $c \ge 0$

$$\begin{aligned} \text{prob}(E_2) &\ge (1 - c/\sqrt{qd})^{\sqrt{qd}} \text{prob}(Z \le c/\sqrt{qd}) \\ &\approx e^{-c} \text{prob}(Z \le c/\sqrt{qd}). \end{aligned}$$

Wir müssen die Wahrscheinlichkeit abschätzen, daß $Z \le c/\sqrt{qd}$ ist. Wir erinnern uns, daß Z das $\sqrt{q/d}$-kleinste der X_i's ist, daß die X_i's unabhängig sind, und daß die

Dichtefunktion jedes X_i's eine nicht-wachsende Funktion auf $[0\ ..\ 1]$ darstellt. Wir schließen, daß $\text{prob}(Z \le c/\sqrt{qd})$ minimal ist, wenn alle X_i's gleichverteilt in $[0\ ..\ 1]$ sind. In diesem Falle ist

$$\mathrm{E}[Z] \approx \sqrt{q/d}/q = 1/\sqrt{qd}$$

da Z die $\sqrt{q/d}$-kleinste von q unabhängigen und gleichverteilten Zufallsvariablen ist. Indem wir die Markov-Ungleichung[5] benutzen, erhalten wir $\text{prob}(Z \le 1.55/\sqrt{qd}) \ge 1 - 1/1.55 = 11/31$. Durch Einsetzen ergibt sich schließlich $\text{prob}(N \ge \sqrt{q/d}) \ge \text{prob}(E_2) \ge 3/40$. □

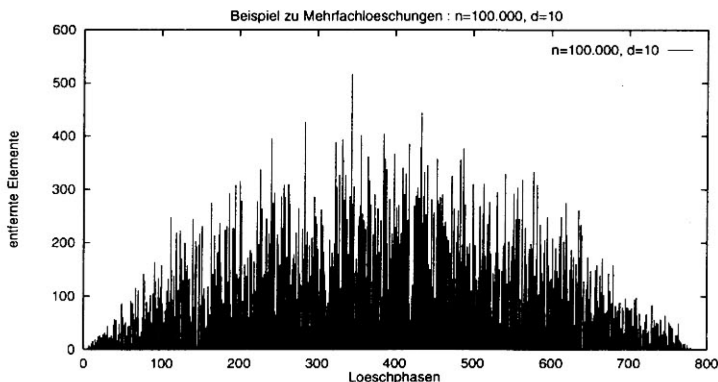

Fig. 1. Simulation des Kürzeste-Wege-Algorithmus mit Mehrfachlöschungen bei einem zufälligen Graphen mit $n = 10^5$ und $d = 10$. Die Ausbeute der einzelnen Löschphasen unterliegt starken lokalen Schwankungen.

Bemerkung : Dieses Lemma spielt eine Hauptrolle in der Analyse des Algorithmus: es wird benutzt um zu zeigen, daß in einer Folge von Löschphasen hinreichend viele von ihnen mindestens $\sqrt{q/d}$ aktive Knoten aus der Warteschlange entfernen, wobei q eine untere Schranke für die Anzahl aktiver Knoten in der Warteschlange zu jedem Zeitpunkt der Folge darstellt.

Besondere Beachtung verdient die Tatsache, daß die einzelnen Phasen als *unabhängige* Zufallsexperimente mit konstanter Erfolgswahrscheinlichkeit angesehen werden können. Vergleiche hierzu auch Abbildung 1. Dies scheint zunächst der Tatsache zu widersprechen, daß in Fällen, bei denen nur sehr wenige Elemente aus der Warteschlange gelöscht werden können, die meisten Distanzwerte der aktiven Knoten unverändert bleiben, ebenso wie die Gewichte der ausgehenden Kanten.

[5] Die Markov Ungleichung ist sehr schwach. Nutzt man stattdessen Chernoff-Schranken, ergibt sich $\text{prob}(Z \le c/\sqrt{qd}) \ge 1 - e^{-\frac{(c-1)^2}{2c}\sqrt{q/d}}$. Ist also q/d hinreichend groß, können wir die Abschätzung verbessern, indem wir $c = 1 + \delta$ setzen, wobei δ eine kleine Konstante darstellt. Ist z.B. $q/d \ge 10^5$, so können wir $c = 1.15$ wählen und erhalten $\text{prob}(N \ge \sqrt{q/d}) \ge 3/10$, also eine Verbesserung um den Faktor vier.

Für die Distanzwerte der aktiven Knoten ist dies unkritisch, da ihre Werte nicht ansteigen können, im Gegenteil werden die Schwankungsbreiten im Beweis sogar aufgebläht, indem alle c_i auf eins gesetzt werden. Eine Lösung für die Kantengewichte ergibt sich in der Form, daß ihre tatsächlichen Werte solange "versteckt" werden, bis man sie wirklich benötigt, um festzustellen, ob das Zufallsexperiment erfolgreich war oder nicht (*Prinzip der verschobenen Entscheidungen*). Der Algorithmus bleibt korrekt, wenn die X_i's in aufsteigender Reihenfolge bzgl. der Relation $\prec$ betrachtet werden bis $X_i > \min_j\{X_j + Y_{j,k}\}$ ist, wobei das j nur die i's umfaßt, die bis dahin gesehen wurden. Auf diese Art und Weise wird jedes Kantengewicht nur in genau einer Löschphase betrachtet.

Im Abschnitt 2.5 zeigen wir, wie das Kriterium für Mehrfachlöschungen aus der Warteschlange effizient im Sekundärspeicher implementiert werden kann.

2.3. Verhalten der Warteschlange bei Mehrfachlöschungen.

Wir haben in Lemma 2 gesehen, daß bei zufälligen regulären gerichteten Graphen aus $G(n,m)$, $d = m/n$, mit konstanter Wahrscheinlichkeit mindestens $N \geq \sqrt{q/d}$ aktive Knoten in einer Phase aus PQ gelöscht werden können, wenn PQ mindestens q aktiven Knoten enthält. Damit läßt sich die Anzahl erforderlicher Löschphasen wie folgt beschränken:

LEMMA 3 *Gegeben sei ein zufälliger gerichteter Graph aus* $G(n,p=d/n)$[6] *mit gleichverteilten Kantengewichten aus dem Bereich* $[0\,..\,1]$. *Dann genügen für das Kürzeste-Wege-Problem mit hoher Wahrscheinlichkeit* $\mathcal{O}(\sqrt{nd})$ *Löschphasen.*

BEWEIS: (Skizze) Im wesentlichen verwenden wir Lemma 2 in Verbindung mit einem Resultat aus [3] über die Entwicklung der Warteschlange PQ bei Explorations-Algorithmen: *Gegeben ist ein zufälliger gerichteter Graph aus* $G(n,p=d/n)$. *Nachdem bei seiner Durchmusterung schon t Knoten besucht wurden, ist die Anzahl der Elemente in PQ, Q_t, binomial verteilt gemäß* $Q_t \sim B\,[n-1; 1-(1-p)^t] + 1 - t$. Wir müssen also untersuchen, nach wievielen Löschphasen mit hoher Wahrscheinlichkeit alle $t_f \leq n$ erreichbaren Knoten besucht wurden. Man geht wie folgt vor: die Binomialverteilung von Q_t sichert, daß ab einer gewissen Mindestgröße von PQ die tatsächliche Größe der Warteschlange mit hoher Wahrscheinlichkeit nur um einen kleinen konstanten Faktor von ihrem Erwartungswert $E\,[Q_t]$ abweicht:
mit $B\,[n-1; 1-(1-d/n)^t] \approx B\,[n; 1-e^{-dt/n}]$ suchen wir also ein t_s, so daß für $t > t_s$ gilt: $\text{prob}(X \leq (1-\epsilon)n(1-e^{-dt/n})) \leq n^{-c}$ für ein $c \geq 1$. Mit Chernoff-Schranken findet man $t_s = \mathcal{O}(\frac{c \ln n}{\epsilon^2 d})$.

Lemma 2 benötigt nicht zwingend einem konstanten Ausgangsgrad der Knoten, es ist nur wichtig, daß für jede Teilmenge von $\sqrt{q/d}$ Knoten die Summe der Aus-

[6] $G(n,p=d/n)$ ist ein Graphmodell, bei dem jede gerichtete Kante mit Wahrscheinlichkeit p vorhanden ist. Die Modelle $G(n,m)$ mit $d = m/n$ und $G(n,p=d/n)$ haben asymptotisch äquivalente Eigenschaften.

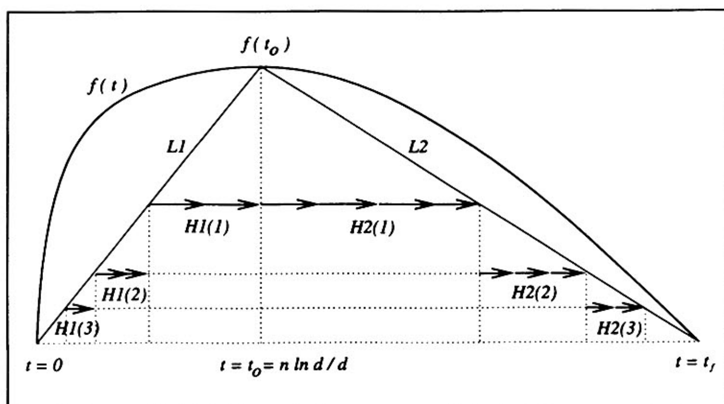

Fig. 2. Erwartete Anzahl aktiver Elemente $f(t)$ in der Warteschlange nachdem schon t Knoten besucht wurden. $L1$ und $L2$ werden als einfache untere Schranke von $f(t)$ benutzt, um eine Schranke für die Anzahl der Löschphasen herzuleiten.

gangskanten beschränkt ist. Wir fordern also $\text{prob}(X \leq (1+\epsilon)(d/n)\sqrt{q/d}\, n) \leq n^{-c}$. Erneute Anwendung von Chernoff liefert $q \geq \mathcal{O}(\frac{c^2 \ln^2 n}{\epsilon^4 d})$. Unter diesen Bedingungen gilt dann für alle Werte von t "in der Mitte des Algorithmus" $\text{prob}(Q_t \geq \sqrt{\frac{1-\epsilon}{1+\epsilon}} E\,[Q_t]) \geq 1 - 2n^{-c+1}$. An den Rändern nehmen wir an, daß der Algorithmus keine Mehrfachlöschungen vornimmt.

Im nächsten Schritt wird der relativ schwierig zu handhabende Term $E\,[Q_t]$ durch zwei Geradengleichungen $L1$ und $L2$ ersetzt, so daß gilt $E\,[Q_t] \geq L1(t)$ für $t \leq t_0$ und $E\,[Q_t] \geq L2(t)$ für $t \geq t_0$, wobei $t_0 = (n \ln d)/d$. Wir nutzen nun $L1$ bzw. $L2$ als untere Schranke für q in Lemma 2. Als eine weitere Vereinfachung unterteilen wir die Entwicklung von $t = 0$ nach $t = t_f \leq n$ in *Strecken* der folgenden Längen: $|H1(1) + H2(1)| = t_f/2 \leq n/2$, $|H1(2) + H2(2)| = t_f/4 \leq n/4$, und allgemein $|H1(i) + H2(i)| = t_f/2^i \leq n/2^i$. Vergleiche auch Abb. 2. Für die "inneren" t's, die zu einer Strecke $H1(i)$ oder $H2(i)$ gehören, gilt dann mit hoher Wahrscheinlichkeit $Q_t \geq \sqrt{\frac{1-\epsilon}{1+\epsilon}} f(t_0)/2^i = \Theta(n/2^i)$.

Eine profitable Löschphase entfernt mindestens $\sqrt{Q_t/d}$ Elemente, die Strecken $H1(i)$ und $H2(i)$ umfassen aber höchstens $n/2^i$ Elemente, also werden für den "inneren Teil" mit hoher Wahrscheinlichkeit weniger als $\sum_{i \geq 1} \frac{n/2^i 2^{i/2}}{k\sqrt{n/d}} = \frac{\sqrt{nd}}{k} \sum_{i \geq 1} 2^{-i/2} = (1+\sqrt{2})\frac{\sqrt{nd}}{k}$ profitable Löschphasen benötigt, k ist dabei eine Konstante.

Da eine Löschphase mit konstanter Wahrscheinlichkeit profitabel, d.h. erfolgreich ist, und da wir die Erfolge aufeinanderfolgender Löschphasen als unabhängige Zufallsexperimente ansehen können, reicht eine weitere Anwendung von Chernoff, um zu zeigen, daß in einer Reihe von $\Theta(\sqrt{nd})$ Phasen hinreichend viele erfolgreich sind.

Darin eingeschlossen ist auch das langsamere Vorankommen am Anfang und gegen Ende des Algorithmus. □

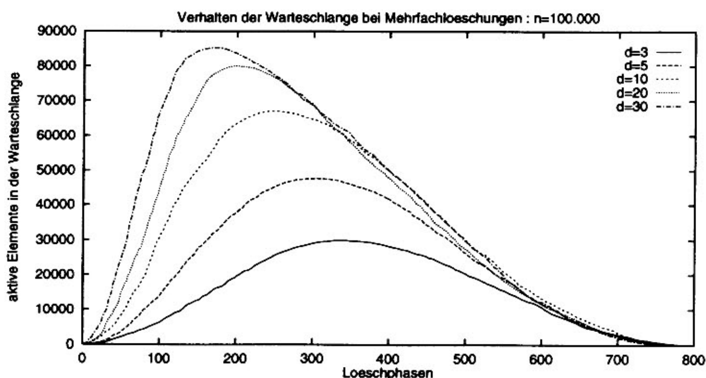

Fig. 3. Inhalt der Warteschlange zum Beginn der Löschphasen. Für $n = 10^5$ wurden verschiedene Ausgangsgrade getestet. Mit wachsendem Grad füllt sich die Warteschlange schneller und erreicht eine größere Maximallänge. Eine Steigerung der Anzahl von Löschphasen kann jedoch nicht beobachtet werden.

Bemerkung: Für $n = 10^9$, $d = 100$ ergibt die Analyse, daß mit Wahrscheinlichkeit größer als $1 - n^{-2}$ höchstens $12.35\sqrt{nd}$ Löschphasen benötigt werden. Unsere Simulationen zeigen sogar mit großer Stabilität, daß $2.5\sqrt{n}$ Phasen auszureichen scheinen, *unabhängig* vom durchschnittlichen Knotengrad. Vgl. auch Abbildung 3.

2.4. Parallele Zugriffe auf die Spitzen der Adjazenzlisten.

Wir betrachten die i-te Löschphase: sei N_i die Anzahl der Knoten, die in Löschphase i zusammen aus PQ entfernt werden können. Für jeden Knoten, den wir aus der Warteschlange löschen, müssen wir auf die Spitze seiner Adjazenzliste zugreifen (, und von dort aus dann blockweise lesen). *Wir setzen voraus, daß die Adjazenzlisten zufällig über die D verfügbaren Festplatten verteilt stehen.* Die Gesamt-Zugriffszeit auf die Spitzen der Adjazenzlisten für die N_i Knoten der Phase i wird nun durch diejenige Festplatte bestimmt, die die meisten dieser Spitzen speichert.

LEMMA 4 *Sind die Adjazenzlisten eines Graphs aus $G(n,m)$, $d = m/n$, zufällig über D Festplatten verteilt, und es gilt $D \leq \sqrt{n/d}$, sowie $D \ln^3 D \leq n$, dann genügen mit hoher Wahrscheinlichkeit $\mathcal{O}(n/D)$ I/Os, um die Spitzen aller Adjazenzlisten im Verlauf des Algorithmus zu lesen.*

BEWEIS: Wegen der zufälligen Verteilung der Adjazenzlisten über die Festplatten können wir die Löschphase i als ein Belegungsproblem auffassen, bei dem N_i Bälle zufällig in D Urnen geworfen werden.

Parallele Zugriffe auf die Spitzen der Ajazenzlisten.						
$n = 2.5\ 10^5$ ($n = 10^6$)						
Disks	Löschphasen		parallele Zugriffe		Balancierung	
2	1253	(2541)	130790	(514926)	1.05	(1.03)
5	1259	(2499)	58554	(222576)	1.17	(1.11)
10	1247	(2490)	33143	(122147)	1.32	(1.22)
20	1245	(2530)	19739	(70062)	1.58	(1.40)

Tab. 1. Lastbalancierung beim parallelen Zugriff auf die Spitzen der Adjazenzlisten. Die Balancierung ist i.a. deutlich besser als $\ln D$ oder $(1+\epsilon)\cdot e$. Die Auslastung ist umso homogener, je größer der Quotient n/D wird.

Es ist bekannt [14], daß die erwartete maximale Anzahl von Bällen in einer beliebigen Urne durch $O(N_i/D)$ beschränkt ist, falls $N_i \geq D \ln D$ ist, und durch $O(\ln D)$, für $N_i \leq D \ln D$. Die (erwartete) Gesamtzahl der I/O-Operationen ist also beschränkt durch $O(\sqrt{nd} + \ln D\sqrt{(D \ln D)d} + n/D)$, wobei der erste Term für die Anzahl der Löschphasen steht, der zweite Term ist verantwortlich für die Löschphasen, bei denen der Erwartungswert von N_i kleiner als $D \ln D$ ist (davon gibt es $\sqrt{(D \ln D)d}$ Stück), und der letzte Term steht für diejenigen Löschphasen, bei denen der Erwartungswert von N_i größer als $D \ln D$ ist. Unter den gemachten Annahmen bzgl. D dominiert der letzte Term.

Unter Benutzung der Markov-Ungleichung kann man leicht beweisen, daß eine Löschphase mit konstanter Wahrscheinlichkeit eine hinreichend gute Balancierung der Zugriffe über die vorhandenen Festplatten zeigt. Die Aussage der hohen Wahrscheinlichkeit folgt dann sofort mit Chernoff-Schranken. Die Unabhängigkeit der einzelnen Zufallsexperimente wird wieder über das Prinzip verzögerter Entscheidungen garantiert: basierend auf der Kenntnis von N_i werden die Plattenzugriffe sukzessive generiert, bis eine Harddisk mehr als cN_i/D Zugriffe erhält (Fehlerfall, eine neue Phase beginnt) oder alle Zugriffe dieser Phase generiert wurden (Erfolgsfall). Dabei ist c eine positive Konstante. □

Bemerkung: Simulationen zeigen, daß die Lastbalancierung in der Praxis oft noch deutlich besser ist, als die Konstanten in obigem Lemma angeben, vgl. Tabelle 2.4. Sogar wenn die Listen einfach zyklisch (anstatt zufällig) über die Festplatten verteilt werden, konnten keine nennenswerten Unterschiede festgestellt werden.

2.5. Modifizierung der externen Graph-Darstellung.

In unserem Algorithmus entfernen wir alle aktiven Knoten aus der Warteschlange deren Distanzwerte kleiner sind als
$L = \min\{dist(v_i)+cost(e);\ 1 \leq i \leq q$ und e ist eine Kante von v_i ausgehende Kante$\}$.
Wie kann der Wert von L effizient bestimmt werden? Der naheliegende Ansatz

besteht darin, L on-line zu berechnen; d.h., wir starten mit $L = \infty$ und verringern L, wenn aktive Knoten aus der Queue entfernt werden. Immer dann, wenn ein aktiver Knoten aus der Warteschlange gelöscht wird, erfolgt ein Zugriff auf seine Adjazenzliste, um das kleinste Kantengewicht der ausgehenden Kanten zu bestimmen, und L wird angepaßt. Dieser Ansatz ist jedoch unbrauchbar, da er die Zugriffe auf die Spitzen der Adjazenzlisten serialisiert. Wir schlagen stattdessen eine Änderung der externen Graph-Darstellung vor. In einem Vorberechnungsschritt speichern wir mit jeder Kante (v, w) das minimale Kantengewicht unter allen von w ausgehenden Kanten und ebenso einen Zeiger auf die Spitze der Adjazenzliste von w. Letztere Information erlaubt uns den Zugriff auf den Anfang der Adjazenzliste eines Zielknotens in *einer* I/O-Operation anstatt in zwei (, indem zuerst auf w zugegriffen wird, um zu erfahren wo seine Adjazenzliste steht und dann erst auf den Anfang der Liste). Die modifizierte Graph-Darstellung erfordert den doppelten Plattenplatz gegenüber der Standard-Representation und kann mit zwei globalen *Scan-* und *Sortier*-Operationen aus der Standard-Darstellung erzeugt werden. Das Sortieren geht sehr schnell und der erhöhte Platzbedarf ist kein ernsthaftes Problem (da selbst in der erweiterten Darstellung für moderate Werte von m/n eine Adjazenzliste in einen einzigen Block paßt).

2.6. R–Heaps als externe Prioritäts-Warteschlangen.

Wir beschreiben hier eine praxistaugliche Alternative zu einem Ansatz von Arge [2], der eine asymptotisch optimale Realisierung für externe Prioritäts-Warteschlangen liefert:

LEMMA 5 ([2]) *Im D-Festplatten Modell unterstützt die Buffer-Tree Prioritäts-Warteschlange das Einfügen eines Elementes und das Löschen des kleinsten Elementes mit amortisierten* $\mathcal{O}(\frac{1}{DB} \log_{\frac{S}{B}} \frac{n}{B})$ *I/O-Operationen, wobei S die Größe des verfügbaren Hauptspeichers angibt.*

Unsere Alternative zu den Buffer-Trees basiert auf R–Heaps[1]. Sie erreicht zwar nicht die asymptotisch optimale Leistung, hat aber den Vorteil, daß sie sehr einfach zu implementieren ist und wir deshalb leicht die Konstanten bestimmen können. Für R–Heaps nutzen wir die D Festplatten in *striped*-Form: jeweils D sich entsprechende Blöcke der verschiedenen Platten werden logisch zu einem Superblock der Größe DB verschmolzen und gleichzeitig bearbeitet.

Sei C eine natürliche Zahl. Wir nehmen an, daß alle Kantengewichte zwischen 0 und C liegen. Daraus folgt, daß für die Kürzeste-Wege-Berechnung alle endgültigen Distanzwerte zwischen 0 und nC liegen. Die vorläufigen Distanzwerte für Elemente in der Warteschlange liegen zwischen min und $min + C$, wobei min der Distanzwert desjenigen Elementes darstellt, das als letztes aus der Warteschlange entfernt wurde.

Es sei nun k eine beliebige positive Zahl und h eine ganze Zahl, so daß $k^h > C$. Weiterhin sei $d_h d_{h-1} \dots d_0$ die k–näre Darstellung des Distanzwertes d, sowie

$m_h m_{h-1} \dots m_0$ die k–näre Darstellung des Minimalwertes min. Ein Distanzwert d kann sich nur in der Warteschlange befinden, wenn $d - min < k^h$. Das heißt aber $d_h = m_h$ oder $d_h = m_h + 1$. Ein Radix Heap besteht aus:

- einem zweidimensionalen Feld B der Größe $h \times k$; jeder Feldeintrag $B(i,j)$ ist eine lineare Liste, die wir auch als einen *Bucket* bezeichnen. $B(i,j)$ enthält alle Paare (d,v), für die $i \geq 0$ minimal ist mit $d_l = m_l$ für $l > i$ und $j = d_i$. Insbesondere enthält $B(0, m_0)$ alle Paare (d,v) mit $d = min$. Den ersten Block eines jeden (unsortierten) Buckets halten wir im Hauptspeicher. Deshalb kostet das Einfügen eines Elementes in einen Bucket amortisiert $\mathcal{O}(1/DB)$ I/O und beschränkt k gemäß $hkDB \leq S$.
- einem weiteren Bucket N, der alle Paare (d,v) mit $d_h = m_h + 1 \bmod k$ enthält.
- einer (internen) Prioritäts-Warteschlange IQ, die alle Indizes nicht-leerer Buckets enthält. Die Indizes sind lexikographisch geordnet, also $(i,j) < (i',j')$ falls entweder $i < i'$ oder $i = i'$ und $j < j'$. IQ speichert nie mehr als $h \cdot k$ Indizes.

Einfügen: Um ein Paar (d,v) einzufügen, berechnen wir zuerst die letzten $h+1$ Stellen der k-nären Darstellung von d in Zeit $\mathcal{O}(h)$ und ohne I/O, dann fügen wir das Paar in den entsprechenden Bucket ein. War der Bucket davor noch leer, wird sein Index auch in IQ gespeichert.

Löschen des Minimums: Falls der Bucket $B(0, m_h)$ nicht leer ist, wird ein beliebiges Element daraus entnommen. Das braucht $\mathcal{O}(1/B)$ I/Os und $\mathcal{O}(\log(hk))$ Zeit; man beachte, daß immer dann, wenn ein Bucket leer wird, auch sein Eintrag in IQ gelöscht werden muß (Löschen des Minimums von IQ). Ist der Bucket $B(0, m_h)$ leer, verwenden wir IQ, um den ersten gefüllten Bucket zu finden. Das ist entweder ein Bucket $B(i,j)$ für irgendwelche i und j oder der Bucket N. Im letzten Fall sei i gleich h. Für beide Fälle bezeichne nun K den nicht-leeren Bucket. Wir bestimmen den neuen Wert von min in K, der gleichzeitig die Basis für die Umverteilung der restlichen Elemente aus K liefert. Es ist wichtig zu beobachten, daß alle Distanzwerte im Bucket K in allen Stellen d_l für $l \geq i$ mit dem neuen Minimum übereinstimmen. Falls $i > 0$ werden nun die restlichen Elemente aus Kneu verteilt. Sie gelangen dabei ausnahmslos in Buckets mit kleinerem ersten Index, da sie bezüglich der k-nären Darstellung des alten Minimums an der i-ten Stelle größer waren und nun an dieser Stelle mit dem neuen Minimum übereinstimmen. Schließlich aktualisieren wir IQ entsprechend.

LEMMA 6 *Sei* $s = S/B$, $k = (s/D \log C) \log (s/D \log C)$ *und* $h = \log_k C$. *Eine Einfüge-Operation in den externen Radix-Heap dauert amortisiert* $\mathcal{O}(1/DB)$ *I/Os, das Löschen des Minimums benötigt amortisiert* $\mathcal{O}(h/DB)$ *I/Os.*

BEWEIS: Für Einfüge-Operationen folgt der Beweis direkt aus der obigen Beschreibung. Beim Löschen des Minimums ist im wesentlichen zu beachten, daß jedes Element höchstens h mal umverteilt werden kann, nämlich jeweils einmal pro Stufe. Daraus folgt die I/O-Schranke. □

Bemerkung: Die interne Darstellung von Fließkommazahlen [10] zieht den Exponenten vor die Mantisse. Somit ändert die binäre Interpretation von Fließkommazahlen als Integer-Werte nicht die Ordnungsrelation, d.h. R-Heaps können auch für Fließkommazahlen benutzt werden.

2.7. Zusammenfassung und Voraussage der I/O–Zeit.

Bauen wir alle Teilergebnisse zusammen, so läßt sich das Kürzeste-Wege-Problem für eine große Klasse zufälliger Graphen mit I/O Komplexität $\mathcal{O}(\frac{n}{D} + \frac{m}{DB} \log_{S/B} \frac{m}{B})$ lösen, wobei maximal $D = \mathcal{O}(\sqrt{n/d})$ Harddisks sinnvoll genutzt werden können.

Ist es nun möglich, unser Beispiel mit $n = 10^9$ Knoten und $m = 10^{11}$ Kanten in vertretbarer Zeit auszurechnen? Nehmen wir $D = 32$ Platten an, jede mit Blockgröße $B = 16$ KBytes und einer durchschnittlichen Zugriffszeit von 16 msec pro Block. Bei $S = 512$ MBytes Hauptspeicher nutzen wir 256 MBytes für den R-Heap (das erlaubt Parameter $h = 5$), 128 MB RAM für den Bit-Vektor, sowie den Rest für das Betriebssystem und augenblicklich behandelte Kanten. Die erweiterte Graph-Darstellung braucht 16 Byte pro Kante, also passen $b = 1000$ Kanten in einen Block, keine Adjazenzliste erstreckt sich über mehr als einen Block. Alle Heap-Operationen zusammen können mit $4hm/(bD)$ I/Os durchgeführt werden, was insgesamt etwa 12 Tage dauert.

In diesem Beispiel sind $12.35\sqrt{nd}$ Löschphasen mit hoher Wahrscheinlichkeit ausreichend. Der Straffaktor für unbalancierte Platten-Zugriffe ist mit hoher Wahrscheinlichkeit kleiner als $\ln 32 < 3.5$ für Phasen mit kleiner Ausbeute und $(1+\epsilon)e < 3$ sonst. Also erwarten wir weniger als $12.35(\sqrt{nd} + \ln D\sqrt{(D \ln D)d}\ ) + 3n/D$ I/Os zum Lesen der Adjazenzlisten. Das entspricht etwa 18 Tagen.

Damit erreichen wir in unserem Beispiel eine I/O-Zeit von etwa 30 Tagen. Unsere Simulationen zeigen, daß ein Straffaktor drei für unbalancierte Zugriffe bei sehr großem Quotienten n/D stark überschätzt ist: Werte von weniger als 1.25 werden schon früh erreicht, vgl. Tabelle 2.4. Angesichts des enormen Plattenbedarfs ist sicher auch eine Hauptspeichervergrößerung auf 1.5 GBytes realistisch. Damit kann der interne Platz für den R-Heap verfünffacht werden, wodurch sich der Parameter h von fünf auf drei reduziert. Diese beiden Verbesserungen zusammengenommen, verringern die I/O-Zeit auf 15 Tage. Ohne die Möglichkeit der Mehrfachlöschungen würde man trotz externer R-Heaps mit parallelem Plattenzugriff mehr als 350 Tage brauchen.

3. Effiziente Wörterbücher unter Platzbeschränkungen.

Hier betrachten wir den Fall, daß es nicht genügend Hauptspeicher gibt, um einen Bit-Vektor für alle n Knoten intern zu halten. Dann gilt es das Problem zu lösen, wie wir bei der Kürzesten-Wege-Berechnung effizient bestimmen können, ob wir einen bestimmten Knoten schon einmal aus der Warteschlange gelöscht haben.

Formal betrachtet haben wir das folgende

Problem: Seien S Bits Hauptspeicher für das Wörterbuch verfügbar; wir wollen eine sich dynamisch vergrößernde Teilmenge X der Knotenmenge V verwalten und die Operationen *Einfügen* und *Test auf Mitgliedschaft* unterstützen. Sei r die maximale Größe der Teilmenge X, die wir mit S Bits effizient verwalten können. Das Ziel besteht also darin, r zu maximieren. Wir beschreiben eine leicht zu implementierende Datenstruktur, die wir *Suffix-Listen* nennen.

3.1. Suffix-Listen.

Für eine in Bits gegebene Hauptspeichergröße S sei n eine ganze Zahl mit $n > S$. *Weiterhin sei $bin(x)$ die binäre Darstellung einer Zahl $x \leq n$. Wir spalten $bin(x)$ in* p hohe Bits und s niedrige Bits auf, wobei $s + p = \lceil \log n \rceil$ gelten muß. $x_{s+p-1}, \ldots, x_s$ bezeichnet den Präfix von $bin(x)$, und $x_{s-1}, \ldots, x_0$ steht für den Suffix von $bin(x)$. Die Suffix-Listen Datenstruktur besteht aus einem linearen Feld P der Größe 2^p Bits und aus einem zwei-dimensionalen Feld L der Größe $a \times (s+1)$ Bits. Die Hauptidee der Suffix-Listen besteht darin, einen gemeinsamen Präfix mehrerer Einträge als ein einziges Bit in P zu speichern, während die unterschiedlichen Suffixe in L eine Gruppe bilden.

P wird als ein Bit-Vektor gespeichert. L kann mehrere Gruppen speichern, wobei jede Gruppe aus einem Vielfachen von $s+1$ Bits besteht. Das erste Bit jeder $(s+1)$-Bit Zeile in L dient als *Gruppenbit.* Der erste s-Bit Suffix-Eintrag einer Gruppe hat als Gruppenbit Eins, die restlichen Gruppenmitglieder haben Gruppenbit Null. Innerhalb einer Gruppe stehen die Elemente lexikographisch aufsteigend sortiert.

Suchen. Bisher können wir zwar gespeicherte Suffix-Gruppen erkennen, aber wie findet man die spezifische Gruppe, die zu einem Eintrag x gehört? Zuerst berechnen wir $k := \sum_{i=0}^{p-1} x_{s+i}\, 2^i$, womit wir die Suchposition im Präfixfeld P erhalten. Dann zählen wir einfach die Anzahl der Einsen im Intervall von $P[0]$ bis $P[k]$. Sei z diese Zahl. Schließlich laufen wir durch L bis wir den z-ten Suffix in L mit Gruppenbit Eins gefunden haben. Wollen wir testen, ob x in X enthalten ist, suchen wir nun linear in dieser Gruppe. In dieser einfachen Version kann das Aufsuchen einzelner Werte jedoch sehr lange dauern, da u.U. große Speicherbereiche sukzessive durchsucht werden müssen.

Einfügen. Um das Element x einzufügen, suchen wir zunächst wie oben beschrieben seine entsprechende Gruppe. Falls x in L eine neue Gruppe hervorruft, müssen in P und L die entsprechenden Gruppenbits gesetzt werden. Der Suffix von x wird so in seine Gruppe eingefügt, daß diese weiterhin sortiert bleibt. Ohne zusätzliche Modifikationen kann so das Einfügen große Verschiebungen im Speicher erfordern, um an der gewünschten Stelle in L Platz zu schaffen.

Die maximale Anzahl von Elementen, r, die auf diese Art mit S Bits abgespeichert werden kann, ist wie folgt beschränkt: wir brauchen 2^p Bits für P und $s + 1 = \lceil \log n \rceil - p + 1$ Bits für jeden Eintrag von L. Deshalb wählen wir p so, daß r unter der Bedingung

$$r \leq (S - 2^p)/(\lceil \log n \rceil - p + 1).$$

maximiert wird. Für $p = \Theta(\log(S) - \log\log(n/S))$ ist der Platzverbrauch sowohl für P als auch für die Suffixe in L klein genug um $r = \Omega(S/\log(n/S))$ zu garantieren.

Sicherungspunkte. Wir zeigen nun, wie die Hauptspeicher-Operationen beschleunigt werden können, ohne die asymptotische Ausbeute $r = \Omega(S/\log(n/S))$ zu gefährden. Zum Suchen oder Einfügen eines Elementes x müssen wir z berechnen, um die richtige Gruppe in x zu finden. Anstatt für jede einzelne Operation potentiell große Speicherbereiche in P und L immer wieder neu zu durchsuchen, legen wir Sicherungspunkte (*Eins–Zähler*) an, die die Anzahl der bis dahin aufgetretenen Einsen speichern. Diese Sicherungspunkte müssen hinreichend dicht liegen, um eine schnelle Suche zu unterstützen, andererseits dürfen sie jedoch nicht mehr als einen kleinen Teil des Hauptspeichers belegen. Mit $2^p \leq r$ gilt für beide Felder $z \leq r$, so daß $\lceil \log r \rceil$ Bits ausreichen, um einen Eins-Zähler zu realisieren. Wenn wir nach jeweils $1/c_1 \lfloor \log r \rfloor$ Einträgen einen Eins-Zähler halten, benötigen wir dafür insgesamt $2\, c_1\, r$ Bits, was akzeptabel ist, solange c_1 eine sehr kleine Konstante ist.

Zur Berechnung von z muß nun nur noch ein Abschnitt mit höchstens $1/c_1 \lfloor \log r \rfloor$ Bits betrachtet werden. z ergibt sich als die Summe der Einsen in diesem Bereich plus dem Wert des vorangehenden Eins–Zählers.

Zum Auffinden der z-ten Gruppe in L wird eine binäre Suche auf den Eins-Zählern von L durchgeführt. Es ist dabei zu beachten, daß die Gruppen bis zu 2^s Einträge umfassen können, wozu dann mehrere Eins-Zähler mit identischen Werten gehören. Trotzdem ist das Finden des Anfangs und des Endes von derart großen Gruppen mit binärer Suche effizient realisierbar. Da die Elemente innerhalb einer Gruppe aber sortiert stehen, genügt eine weitere binäre Suche auf den Gruppenmitgliedern, um die gewünschte Position in L zu finden.

Die Handhabung von Einträgen verschiedener Längen verlangt in unserer Analyse der Internspeicher-Operationen die Betrachtung im Bit-Komplexitätsmaß.

Für den Test eines Elementes auf Mitgliedschaft in X führen wir binäres Suchen auf Zahlen der Größe $\lceil \log r \rceil$ Bits bzw. s Bits aus. Um nach einem Element x zu suchen, brauchen wir also $\mathcal{O}(\log^2 r + s^2) = \mathcal{O}(\log^2 S + \log^2 n/S)$ Bit-Operationen.

Wir betrachten das Einfügen, bei den noch zwei Probleme bestehen: die Aufnahme eines neuen Elementes in eine Gruppe verursacht u.U. das Verschieben großer Datenmengen, und außerdem müssen nach jedem Einfügen die Eins-Zähler aktualisiert werden.

Puffer. Eine einfache Lösung des obigen Problems besteht in der Verwendung einer zweiten Datenstruktur B, die zwar weniger platzsparend ist, dafür aber das schnelle Einfügen und Nachschlagen gestattet. B wird als Puffer für neue Einträge genutzt. Sobald die Anzahl der Elemente in B einen bestimmten Grenzwert übersteigt,

wird B mit den alten Suffix-Listen gemischt, um dann wieder eine aktuelle und platzsparende Datenstruktur zu erhalten. Wenn wir B nicht zu groß wählen, erreichen wir amortisiert deutlich bessere Laufzeiten für die interne Rechnenzeit, ohne substantielle Einbußen bei der I/O-Schranke in Kauf nehmen zu müssen. Man beachte daß die Tests auf Mitgliedschaft in X nun ebenfalls auf B ausgedehnt werden müssen.

Der Puffer B kann entweder deterministisch über balancierte Bäume, oder noch platzsparender, randomisiert als ein Feld für Hashing mit offener Adressierung realisiert werden: B speichert höchstens $c_2 \frac{r}{\lceil \log n \rceil}$ Elemente der Größe $p + s = \lceil \log n \rceil$, wobei c_2 eine kleine Konstante darstellt. Solange es noch etwa 10 % Platz in B gibt, fügen wir weiter Elemente in B ein, ansonsten wird B sortiert und die Suffixe wandern von B in die entsprechenden Gruppen in L. Die Größe der Hashtabelle wird nicht voll ausgenutzt, um die erwartete Anzahl von Kollisionen beim Hashen konstant zu halten. Wir erwarten also für Tests und Einfügungen auf der Hashtabelle je $\mathcal{O}(\log n)$ Bit-Operationen.

Das Sortieren von B geht mit $\mathcal{O}(\log n \frac{r}{\log n} \log \frac{r}{\log n}) = \mathcal{O}(r \log r)$ Bit-Operationen. Das Zusammenmischen der Datenstrukturen (beginnend mit den größten auftretenden Schlüsseln) kann mit $\mathcal{O}(1)$ Speicherdurchläufen, also $\mathcal{O}(S)$ Operationen erreicht werden. Darin eingeschlossen ist das Updaten aller Eins-Zähler.

Zusammengenommen brauchen wir also $\mathcal{O}(r/(r/\log n)) = \mathcal{O}(\log n)$ Sortier- und Misch-Phasen pro I/O-Runde. Trotz der zusätzlichen Daten-Strukturen gilt immer noch $r = \Theta(\frac{S}{\log(n/S)})$. Somit ist die gesamte Bit-Komplexität der internen Operationen für n Einfüge- und Test-Operationen gegeben durch

$$\begin{aligned} & \mathcal{O}(n/r \ \log n \ (r \ \log r + S) + n \ \log n + n \ (\log^2 S + \log^2 n/S)) \\ = \ & \mathcal{O}(n \ \log n \ (\log S + \log n/S) + n \ \log n + n \ (\log^2 S + \log^2 n/S)) \\ = \ & \mathcal{O}(n \ \log n \ (\log S + \log n/S)). \end{aligned}$$

THEOREM 1 *Das Suchen und Einfügen von n Knoten in die Suffix-Listen Struktur bei Platzrestriktion S geht in $\mathcal{O}(n/S \ \log(n/S))$ I/O-Runden mit $\mathcal{O}(n \ \log n \ (\log S + \log n/S))$ internen Bit-Operationen.*

Bei der Benutzung von Suffix-Listen im Kürzeste-Wege-Algorithmus ist zu beachten, daß die benötigte Anzahl von I/O-Runden nicht von den konkreten Werten von n und S abhängt; nur das Verhältnis zwischen n und S ist wichtig. Wir brauchen zur Berechnung der kürzesten Wege keine Lösch-Operation auf den Suffix-Listen, sie kann jedoch für allgemeine Wörterbücher analog zur Einfüge-Operation implementiert werden.

Die informationstheoretische Schranke. Wir betrachten noch einmal unser formales Problem. Wir wollen die Größe r der Teilmenge X maximieren. Für den statischen Fall ist die untere Schranke leicht: wir beobachten $\lceil \log \binom{n}{r} \rceil \leq S$. Im dynamischen Fall müssen wir jedoch alle früheren Konfigurationen von X betrachten. Daraus folgt $\lceil \log (\sum_{i=0}^{r} \binom{n}{i}) \rceil \leq S$. Wir wollen r maximal bzgl. dieser Ungleichung

wählen. Dann ist die minimale Anzahl der I/O-Runden offensichtlich gegeben durch $I = \lceil \frac{n}{r} \rceil$.

Für $r \leq (n-2)/3$ können wir verwenden: $\binom{n}{r} \leq \sum_{i=0}^{r} \binom{n}{i} \leq 2 \binom{n}{r}$. Die Korrektheit folgt direkt aus $\frac{\binom{n}{i}}{\binom{n}{i+1}} \leq 1/2$ für $i \leq (n-2)/3$. Wir sind nur interessiert am Logarithmus, also betrachten wir: $\log \binom{n}{r} \leq \log(\sum_{i=0}^{r} \binom{n}{i}) \leq \log(2 \binom{n}{r}) = \log \binom{n}{r} + 1$.
Aber nun ist offensichtlich, daß es in diesem beschränkten Bereich genügt, den letzten Binomialkoeffizienten zu betrachten.

Der Fehler in unserer Abschätzung ist höchstens ein Bit. Weiterhin werden wir sehen, daß die Beschränkung $r \leq (n-2)/3$ für alle interessanten Kombinationen von n und S unerheblich ist. Mit $\log \binom{n}{r} = \log \frac{n\,(n-1)\cdots(n-r+1)}{r!} = \sum_{j=n-r+1}^{n} \log j - \sum_{j=1}^{r} \log j$ können wir den Logarithmus durch zwei entsprechende Integrale approximieren. Durch geeignetes Verschieben der Integralgrenzen können wir sicher sein, eine untere Schranke zu berechnen: $\log \binom{n}{r} \geq \int_{n-r+1}^{n} \log(x)\,dx - \int_{2}^{r+1} \log(x)\,dx$.

Wir vergleichen nun die Suffix-Listen mit der unteren Schranke, sowie mit Hashing bei offener Adressierung. Die Konstanten für Suffix-Listen wurden so gewählt, daß gilt $2\,c_1 + c_2 \leq 1/10$, was bedeutet, daß wenn r Bits pro Runde gespeichert werden könnten, $r/10$ Bits für die Zusatzstrukturen zu schnelleren Berechnung genutzt werden. Beim Hashing haben wir ebenfalls 10 % des Speichers freigelassen. In Tabelle 3.1. vergleichen wir die Anzahl erforderlicher I/O-Runden für verschiedene Paare von n und S.

n/S	Anzahl I/O – Runden.				
	Untere Schranke		Suffix-	Hashing	
	$n{=}2^{20}$	$n{=}2^{30}$	Listen	$n{=}2^{20}$	$n{=}2^{30}$
$1{+}\epsilon$	2	2	6	24	35
1.01	3	3	6	24	35
1.10	4	4	7	26	38
1.25	5	5	7	29	43
1.50	7	7	9	35	52
2.00	10	10	11	47	69
3.00	17	17	21	73	106
4.00	24	24	28	97	141
8.00	59	59	65	203	291
16.00	137	137	149	384	599
32.00	312	312	333	880	1232

Tab. 2. Anzahl der I/O - Runden für verschiedene Datenstrukturen: Suffix-Listen und Hashing mit offener Adressierung. n/S gibt das Verhältnis zwischen Knotenzahl und Speichergröße an.

Ergebnis. Suffix-Listen können die Lücke in der I/O-Performanz zwischen der unteren Schranke und einfachen Ansätzen wie Hashing mit offener Adressierung schließen. Für $n = S + \epsilon$ ist die Anzahl der I/O-Runden höchstens das Dreifache der unteren Schranke. Schon für $n \geq 1.01\ S$ erreichen wir Zwei-Optimalität. Die Leistung wird mit wachsendem Verhältnis von n und S besser. For $n = 2^i, i \in I\!N$ zeigt sich ein besonders gutes Verhalten, weil in diesen Fällen P tatsächlich in seiner vollen Länge ausgenutzt wird. Für $n = S + \epsilon$ gibt es an dieser Stelle ein weiteres Optimierungspotential: Wir wissen, daß dann fast die Hälfte der Einträge in P immer Null sein werden, und zwar genau diejenigen, die lexikographisch größer sind als der Suffix von n selbst. Wird das P-Feld an dieser Stelle abgeschnitten, bleibt mehr Raum für L, wodurch wiederum mehr Elemente pro Runde gespeichert werden können.

Literaturverzeichnis

1. R. Ahuja, K. Mehlhorn, J.B. Orlin, and R.E. Tarjan. Faster algorithms for the shortest path problem. *JACM*, 3(2):213–223, 1990.
2. L. Arge. *Efficient external-memory data structures and applications*. PhD thesis, University of Aarhus, 1996.
3. Noga Alon, Joel H. Spencer, and Paul Erdös. *The Probabilistic Method.* Wiley, 1992.
4. R.D. Barve, E.F. Grove, and J.S Vitter. Simple randomized merge sort on parallel disks. *Parallel Computing*, 23(4):601–?, 1997.
5. A. Brodnik and J. I. Munro. Membership in constant time and minimum space. *European Symposium on Algorithms*, pages 72–81, 1994.
6. Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, and J.S. Vitter. External-memory graph algorithms. In *SODA*, pages 139–149, 1995.
7. Y.-J. Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Algorithms*. PhD thesis, Brown University, 1995.
8. F.Y. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *CACM*, 25(9):659–665, 1982.
9. E.W. Dijkstra. A note on two problems in connection with graphs. *Num. Math.*, 1:269–271, 1959.
10. IEEE standard 754-1985 for binary floating-point arithmetic. *SIGPLAN Notices*, 2:9-25, 1987.
11. David Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *J. Assoc. Comput. Mach.*, 42:321–329, 1995.
12. K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.
13. Kurt Mehlhorn, S. Näher, and Ch. Uhrig. The LEDA User Manual (Version R 3.5). Technical report, Max-Planck-Institut für Informatik, 1997. http://www.mpi-sb.mpg.de/LEDA/leda.html.
14. Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
15. R.E. Tarjan and U.Vishkin. Finding biconnected components and computing tree functions in logarithmic time. *SICOMP*, 14(4):862–874, 1985.