Bachelorarbeit mit dem Thema

Implementierung und experimentelle Untersuchung von Parallelem Global-Curveball zur Randomisierung Massiver Bipartiter Graphen

verfasst von:

Marius Hagemann

Matrikelnummer 5732742
s2486252@stud.uni-frankfurt.de

27. Januar 2020

Betreuer: Prof. Dr. Ulrich Meyer

Goethe-Universität Frankfurt am Main

Fachbereich Informatik

Bitte dieses Formular zusammen mit der Abschlussarbeit abgeben!

Erklärung zur Abschlussarbeit

gemäß § 25, Abs. 11 der Ordnung für den Bachelorstudiengang Informatik vom 06. Dezember 2010:

Hiermit erkläre ich Herr / Frau

_____

Die vorliegende Arbeit habe ich selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst.

Frankfurt am Main, den

_____

Unterschrift der Studentin / des Studenten

Inhaltsverzeichnis

1 Einleitung

wozu Randomisierung? – Als (zufällige) Eingabe um Algorithmen zu testen? – Zum Analysieren von Netzwerken?

es gibt auch andere Methoden um zufällige Graphen zu erstellen (zufällige Kanten zwischen Knoten) aber dann bleibt die gewollte Struktur nicht erhalten

also Global Curveball (, bei dem Kanten getauscht werden)

wir beschränken uns hier nur auf den Spezialfall der bipartiten Graphen wodurch es einfacher wird

wozu GlobalCurveball? warum macht man das?

– Zufällige Graphen, wobei die Grade aller Knoten erhalten bleiben

Was ist networkit? + Quelle

was heißt massiver graph ? hoher Knotengrad??

2 Grundlagen

Notation, Graph, bipartitheit, randomisierung, ...?
Kanten tauschen, globaler Tausch,
Parallelität?

3 Beschreibung blabla?

3.1 Bestimmung der gemeinsamen Nachbarschaft

Wie bereits erwähnt müssen um einen Curveball-Tausch auf den Knoten u und v auszuführen die Nachbarschaften der beiden Knoten bekannt sein. Dabei sind die Knoten gesucht, welche jeweils nur entweder u oder v als Nachbarn haben, also in der disjunkten Nachbarschaft (kann man das sagen?) liegen. Um diese Knoten zu finden, kann man jedoch einfach die gemeinsame Nachbarschaft bestimmen. Die Knoten in der disjunkten Nachbarschaft sind dann alle Nachbarn von u und v, welcheVORSORTIERT nicht in der gemeinsamen Nachbarschaft liegen.
Als Datenstruktur liegen uns die Nachbarschaften in einer Art Adjazenzliste (naja eine Liste ist es ja nicht wirklich) vor, sodass es für jeden Knoten des Graphen einen Vektor gibt, indem die Nachbarn gespeichert sind. Um nun gemeinsame Nachbarn zweier Knoten zu bestimmen, muss man also nur herausfinden, welche Einträge in beiden Vektoren gemeinsam vorkommen. Dafür gibt es verschiedene Varianten, die im Folgenden erklärt werden.

Als ersten "naiven" Ansatz könnte man für jedes Element des Vektors u den gesamten anderen Vektors v per linearer Suche nach diesem Element durchsuchen. Hierfür ergibt sich eine Laufzeit von $\mathcal{O}(|u| \cdot |v|)$, was aber natürlich nicht sehr sinnvoll ist, da der Vektor v ziemlich oft durchlaufen werden muss und wir im Falle von massiven Graphen davon ausgehen können, dass die Vektoren (also die Nachbarschaften) ziemlich groß werden.
Um dieses Problem zu verhindern kann man beide Vektoren aufsteigend sortieren. Um nun zu herauszufinden, welche Werte in beiden Vektoren vorkommen, muss man lediglich u und v gleichzeitig linear durchlaufen und testen, ob die Werte gleich sind. Somit muss man jedes Element der beiden Vektoren - nach dem Sortieren - nur einmal betrachten, was offensichtlich zu einer verbesserten Laufzeit im Vergleich zum "naiven" Ansatz führt. Man erhält damit eine Laufzeit von $\mathcal{O}(|u| \cdot \log(|u|) + |v| \cdot \log(|v|))$. Diese Variante wird im Folgenden als **SortSort** (darf ich das in englisch lassen?) bezeichnet.
Dies kann man leicht abwandeln zur Variante **SortSearch**. Dabei wird nur der größere Vektor (z.B. u) sortiert. Für jedes Element des kleineren Vektors wird nun per binärer Suche geprüft, ob das Element auch im größeren Vorhanden ist. Analog zu dieser Variante gibt es noch **SearchSort**, bei welcher der kleinere Vektor sortiert wird. LZ?
Eine weitere Methode um viele Werte schnell zu durchsuchen, bietet die Datenstruktur *Set*. Dabei wird jedes Element des einen Vektors (z.B. u) in das Set eingefügt. Die Datenstruktur baut aus diesen Elementen dann einen Binären Suchbaum. Für jedes Element aus v kann nun in logarithmischer Zeit bestimmt werden, ob es im Set und somit auch in u vorhanden ist. Auch hier gibt es zwei analoge Varianten, nämlich **SetSearch**, bei der der größere Vektor in das Set eingefügt wird und **SearchSet**, bei der der kleinere Vektor zum Set hinzugefügt wird.
Die letzte Methode, die wir (darf ich "wir" sagen?!) an dieser Stelle betrachten, ist die Verwendung der Datenstruktur *unordered_set*. Diese ist sehr ähnlich wie Set, mit dem Unterschied, dass die Werte nicht in geordneter Reihenfolge gespeichert werden, sondern in einer Hash-Tabelle?. Ebenfalls gibt es hierbei wieder die Varianten, in denen der größere Vektor in das unordered_set eingefügt wird (**USetSearch**) oder der kleinere (**SearchUSet**).

Wir haben also insgesamt sieben verschiedene Möglichkeiten, die alle eine ähnliche Laufzeit (wirklich?!) haben. Somit muss experimentell herausgefunden werden, welche dieser Varianten für welche

Instanzen am schnellsten sind. Weiterhin testen wir noch, ob die Invariante, dass die Vektoren bereits sortiert sind, zu einer besseren Laufzeit führt.

LAUFZEIT $\mathcal{O}\Big((|u| + |v|)\log(|u| + |v|)\Big)$??

3.2 Tauschen der Nachbarn

Im vorherigen Teil wurde beschrieben, wie man die gemeinsame und die disjunkte Nachbarschaft zweier Knoten bestimmt. Nun beschäftigen wir uns damit, wie man die Knoten aus der disjunkten Nachbarschaft zufällig auf tauscht. Dafür unterscheiden wir zwei Möglichkeiten.

Die erste Methode nennen wir **Permutation**. Dabei werden

3.3 Globaler Tausch

4 Implementierung

hier dann nur code ?!

4.1 Networkit

5 experimentelle Untersuchung

google Test/ google benchmark
bestimmung der schnellsten Varianten.. – disjoint neighbors – trade plots
WO WIRD DIE BIPARTITHEIT AUSGENUTZT? –> parallele Trades

6 Zusammenfassung?

was hat das alles gebracht? ausblick? was könnte man verbessern?

Literaturverzeichnis

[1] https://arxiv.org/pdf/1804.08487.pdf, abgerufen ?!