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

5. Februar 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, zufall
- Parallelität (OpenMP)?

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.
Pseudocode zu den einzelnen Varianten?
LAUFZEIT ??

3.2 Tauschen der Nachbarn

Im vorherigen Teil wurde beschrieben, wie man die gemeinsame und die disjunkte Nachbarschaft zweier Knoten u und v bestimmt. Nun beschäftigen wir uns damit, wie man diese Knoten zufällig tauscht. Als Eingabe stehen die Vektoren d, welcher alle Knoten aus der disjunkten Nachbarschaft enthält und c, der die gemeinsamen Nachbarn enthält, zur Verfügung. Weiterhin seien $\deg(u)$ und $\deg(v)$ die ursprünglichen Knotengrade. Wir betrachten hierfür zwei Möglichkeiten.

Die erste Idee besteht darin, den Vektor der disjunkten Nachbarn zufällig zu permutieren, sodass jedes Element an einer zufälligen Position steht. Um nun die beide "neuen" Nachbarschaften von u und v zu erstellen, werden zuerst die Knoten aus der gemeinsamen Nachbarschaft in die (leeren) Vektoren von u und v kopiert. Dann werden die ersten Elemente aus dem permutierten Vektor in den kleineren aus u und v kopiert. Dabei werden genau so viele Elemente kopiert, dass dieser wieder die gleiche Größe hat wie vor dem Beginn des Tausches. Die restlichen Elemente aus dem disjunkten Vektor werden schließlich in den anderen Vektor kopiert. Zur besseren Veranschaulichung ist in Abbildung 3.1 ein Beispiel zu sehen. Die Kästchen stellen die Elemente des permutierten Vektors

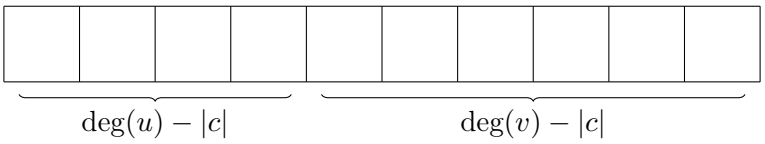

Abbildung 3.1: ist das Beispiel unnötig?

d dar. Ohne Beschränkung der Allgemeinheit betrachten wir den Fall, dass die Nachbarschaft von u die kleinere ist, also dass $\deg(u) \leq \deg(v)$ gilt. Somit werden die ersten $\deg(u) - |c|$ Elemente zum Vektor u hinzugefügt. Die restlichen Elemente werden folglich in v kopiert. Die Größen von u und v - also dementsprechend die Knotengrade der beiden Knoten - haben sich durch das zufällige Tauschen der disjunkten Nachbarn offensichtlich nicht verändert.
Bei die diesem Verfahren fällt jedoch auf, dass einige Elemente beim Permutieren unnötig vertauscht werden. Für jedes Element ist es eigentlich nur entscheidend, ob es unter den ersten $\deg(u)$ liegt (also zum Vektor u hinzugefügt wird) oder nicht. Auf welcher Position genau es in diesen Bereichen liegt, ist egal. Man kann also die Laufzeit dieser Variante verbessern, indem nicht der ganze Vektor zufällig permutiert wird, sondern nur die ersten $\deg(u)$ Elemente zufällig gewählt werden. Dies macht die Funktion *random_bipartition_shuffle*.
Ein Nachteil bei dieser Methode ist, dass durch das zufällige Vertauschen die beiden Vektoren u und v nicht mehr sortiert sind. Damit wird die im vorherigen kapitel? beschriebene Invariante eventuell verletzt. Um die Invariante aufrecht zu halten muss man also als letzten Schritt die beiden Vektoren nochmals sortieren. Wir nennen diese Variante **Permutation**.

Die zweite Möglichkeit die wir betrachten heißt **Distribution**.
Die Idee besteht dabei, dass wir über jedes Element des Vektors d iterieren und eine Wahrscheinlichkeit berechnen, mit der das Element in den Vektor u (beziehungsweise v) eingefügt werden soll. Dann wird in einem Bernoulli Experiment mit genau dieser Wahrscheinlichkeit ein Zufallsbit gezogen. Je nachdem, welchen Wert das Zufallsbit hat, wird das Element dann entweder in u oder in v kopiert. Dies wird so lange wiederholt, bis einer der beiden Vektoren seine maximale Kapazität

erreicht hat.
Um die Wahrscheinlichkeit zu berechnen werden am Anfang zwei Variablen n_v und n_u initialisiert, welche den Kapazitäten der beiden Vektoren u und v entsprechen, wenn die Elemente aus der gemeinsamen Nachbarschaft nicht berücksichtigt werden. Es gilt also $n_v = \deg(v) - |c|$ und $n_u = \deg(u) - |c|$. Damit hat das erste Element des Vektors d eine Wahrscheinlichkeit von $p_u = \frac{n_u}{n_u+n_v}$, dem Vektor u hinzugefügt zu werden und analog eine Wahrscheinlichkeit $p_v = \frac{n_v}{n_u+n_v}$, um in v zu gelangen. Offensichtlich gilt $p_u + p_v = 1$. Dann wird mit einer der beiden Wahrscheinlichkeiten das Bernoulli Experiment durchgeführt, wobei es egal ist, welche Wahrscheinlichkeit man dazu wählt, da p_u genau die Gegenwahrscheinlichkeit von p_v ist und umgekehrt. Wählt man beispielsweise p_u und das Experiment liefert eine eins, dann wird das aktuelle Element in den Vektor u kopiert. Dabei hat sich aber offensichtlich die verbleibende Kapazität des Vektors u verringert. Also muss der Wert n_u dekrementiert werden. Analoges gilt, falls das Element in den Vektor v kopiert wird. Somit ändern sich nach jeder Iteration die Wahrscheinlichkeiten p_u beziehungsweise p_v. Gilt nach irgendeinem Zeitpunkt entweder $n_u = 0$ oder $n_v = 0$, ist offenbar einer der Vektoren keine Kapazität mehr frei. Somit werden die übrigen Elemente, die noch in d vorhanden sind, einfach dem anderen Vektor hinzugefügt.
Ein Vorteil dieser Methode ist, dass die beschriebene Invariante aufrecht erhalten werden kann. War der Vektor der disjunkten Nachbarschaft vor Beginn dieser Methode aufsteigend sortiert, dann sind auch die bisherigen Elemente der Vektoren u und v aufsteigend sortiert, da für jedes Element nacheinander entschieden wurde, ob es zu u oder zu v hinzugefügt wird und dabei die Reihenfolge der Elemente untereinander nicht verändert wurde.
Zum Schluss müssen noch die die gemeinsamen Nachbarn zu den Vektoren u und v hinzugefügt werden. Möchte man die Invariante aufrecht erhalten, dann sind die beiden Vektoren wie beschrieben schon aufsteigend sortiert. Da auch die Elemente aus c aufsteigend sortiert sind, erhält man die endgültigen Vektoren von u und v durch ein Mergen mit c. Soll die Invariante jedoch nicht aufrecht erhalten werden, reicht es aus, die Elemente aus c an das Ende der beiden Vektoren zu kopieren.

3.3 Globaler Curveball Tausch

Wie in den beiden vorherigen Abschnitten beschrieben, besteht ein Curveball Tausch auf zwei Knoten u und v daraus, die gemeinsame und disjunkte Nachbarschaft der beiden Vektoren zu bestimmen und schließlich die Knoten aus der disjunkten Nachbarschaft zufällig zu tauschen.
Bei einem Globalen Curveball Tausch, werden mehrere Curveball-Tausche gleichzeitig ausgeführt. Hierbei wird ausgenutzt, dass wir ausschließlich bipartite Graphen betrachten echt?

4 Implementierung

hier dann nur code ?!
welche latex umgebung für den code ?!

4.1 Networkit

5 experimentelle Untersuchung

- google Test/ google benchmark
- bestimmung der schnellsten Varianten..
  – disjoint neighbors
  – trade
  – auf welcher Maschine? gluten
  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 ?!