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

18. Februar 2020

Betreuer:
Prof. Dr. Ulrich Meyer
Manuel Penschuck

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 **4**

2 Grundlagen **5**
2.1 Mathematische Definitionen 5
2.2 NetworKit 6
2.3 Datenstruktur 6
2.4 Global Curveball **(auf bipartiten Graphen)** 6
2.5 Parallelisierung 8

3 Beschreibung blabla? **9**
3.1 Bestimmung der gemeinsamen Nachbarschaft 9
3.2 Tauschen der Nachbarn 10
3.3 Globaler Curveball Tausch 11

4 Experimentelle Untersuchung **12**
4.1 Versuchsaufbau 12
4.2 Messung 13
4.3 Auswertung 14

5 Implementierung **15**
5.1 Networkit 15

6 Zusammenfassung? **16**

1 Einleitung

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

In diesem Kapitel gehen wir auf die wichtigsten theoretischen Grundlagen, die für diese Arbeit benötigt werden, ein.

2.1 Mathematische Definitionen

Zu Beginn definieren wir die grundlegenden mathematischen Begriffe. Als wichtigste Grundlage dient hierbei das Konstrukt des **ungerichteten** Graphen.

Definition 2.1 (Graph)**.**
Ein (ungerichteter) **Graph** $G = (V, E)$ ist ein Tupel bestehend aus einer Knotenmenge V und einer Kantenmenge E. Eine Kante verbindet zwei Knoten miteinander und ist damit eine Menge, aus zwei Knoten. Es gilt: $E \subseteq \{\{u, v\} |\ u, v \in V, u \neq v\}$.

In dieser Arbeit spielt eine spezielle Klasse von Graphen, bipartite Graphen, eine zentrale Rolle. Bei einem bipartiten Graphen kann man die Knoten in zwei Mengen teilen, sodass alle Kanten nur zwischen den beiden Mengen verlaufen und nicht innerhalb einer Menge. Formal bedeutet dies:

Definition 2.2 (bipartiter Graph)**.**
Ein Graph $G = (V, E)$ heißt **bipartit**, wenn es Teilmengen $V_1 \subset V$ und $V_2 \subset V$ gibt, für die $V_1 \cup V_2 = V$ und $V_1 \cap V_2 = \emptyset$ gilt, sodass für jede Kante $e \in E$ ein $u \in V_1$ und ein $v \in V_2$ existiert, sodass $e = \{u, v\}$ gilt. Die Knotenmengen V_1 und V_2 werden auch als Partitionen bezeichnet.

Ein Beispiel für einen bipartiten Graphen sieht man in Abbildung 2.1. Dabei gilt für die Partitionen: $V_1 = \{v_1, v_2, v_3, v_4\}$ und $V_2 = \{v_5, v_6, v_7, v_8\}$. Man sieht deutlich, dass alle Kanten die Partitionen V_1 und V_2 "kreuzen".

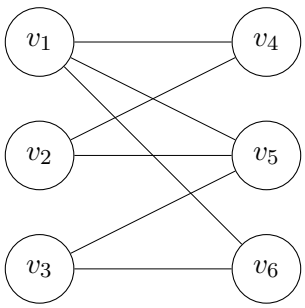

Abbildung 2.1: Beispiel eines bipartiten Graphen

überleitung Für einen Curveball-Tausch ist vor allem der Begriff der Nachbarschaft, genauer der gemeinsamen und disjunkten Nachbarschaft, entscheidend.

Definition 2.3 (Nachbarschaft)**.**
Ein Knoten $u \in V$ heißt **benachbart** (oder **adjazent**) zu einem anderen Knoten $v \in V$, wenn es eine Kante $\{u, v\} \in E$ gibt. Die Menge $N(u)$ aller adjazenten Knoten von u nennt man **Nachbarschaft**.

Definition 2.4 (gemeinsame und disjunkte Nachbarschaft)**.**
Die **gemeinsame** Nachbarschaft $N_c(u, v)$ zweier Knoten u und v ist die Menge aller Knoten, die sowohl zu u als auch zu v adjazent sind. In der **disjunkten** Nachbarschaft $N_d(u, v)$ von u und v sind dagegen alle Knoten die nur zu einem der beiden Knoten adjazent sind.
Es gilt also $N_c(u, v) = N(u) \cap N(v)$ und $N_d(u, v) = \big(N(u) \cup N(v)\big) \setminus \big(N(u) \cap N(v)\big)$

Dabei bemerken wir, dass in einem bipartiten Graphen zwei Knoten aus einer Partitionsklasse nie in der gegenseitigen Nachbarschaft liegen können. Dieser Fakt wird beim Bipartiten Global Curveball ausgenutzt. Zuletzt sind noch die Begriffe Knotengrad und Gradsequenz relevant.

Definition 2.5 (Knotengrad)**.**
Der **Grad** eines Knotens $v \in V$ wird mit $\deg(v)$ bezeichnet und entspricht die Anzahl der adjazenten Knoten von v. Es gilt also $\deg(v) = |N(v)|$ für alle Knoten $v \in V$.

Definition 2.6 (Gradsequenz)**.**
Die **Gradsequenz** eines Graphen $G = (V, E)$ mit $|V| = n$ Knoten ist gegeben durch das Tupel $D = (d_1, \ldots, d_n)$, wobei $d_i = \deg(v_i)$ der Grad des Knotens v_i ist.

Im bipartiten Graph aus Abbildung 2.1 hat beispielsweise der Knoten v_1 dem Grad $\deg(v_1) = 3$. Für die Gradsequenz des Graphen gilt: $D = (3, 2, 2, 2, 3, 2)$. **kann man das so lassen?**

2.2 NetworKit

NetworKit [2] ist ein Open-Source Projekt, dass es zum Ziel hat, "Werkzeuge für die Analyse großer Netzwerke, in den Größenordnungen von Tausenden bis Milliarden von Kanten, zur Verfügung zu stellen". [1]
Innerhalb von NetworKit Gibt es einfache Graph Datenstrukturen **blabla**
Man kann es mit python nutzen **blabla** hmmm keine Ahnung

2.3 Datenstruktur

In NetworKit **[2] muss das immer hin, wenn NetworKit erwähnt wird?!** werden Graphen in einer eigenen Datenstruktur gespeichert. Um den Algorithmus zu vereinfachen, wird der Graph in eine einfachere Datenstruktur transformiert. Diese Datenstruktur muss so aufgebaut sein, dass man damit effizient zwei zufällige Knoten auswählen, die (disjunkten und gemeinsamen) Nachbarschaften berechnen und Knoten aus der disjunkten Nachbarschaft tauschen kann. Dafür eignet sich am Besten eine Art Adjazenzlistendarstellung des Graphen, wobei jedoch keine echten verketteten Listen verwendet werden, sondern lediglich Vektoren. Es wird also für einen Knoten $v \in V$ ein Vektor erstellt, indem alle adjazenten Knoten gespeichert sind. Da wir ausschließlich bipartite Graphen betrachten werden, speichern wir noch in einem weiteren Vektor die Knoten von einer der beiden Bipartitionsklassen **die größere? oder egal?** . Diesen werden wir als **Partitions-Vektor** bezeichnen.
Die Vektoren sind dabei vom C++ **Datentyp (falsches wort/was ist richtig? container?)** `std::vector`. **Dieser Container unterstützt random access in** $\mathcal{O}(1)$
KNOTEN ENTSPRECHEN INTS!?

2.4 Global Curveball (auf bipartiten Graphen)

Wie **in der Einleitung beschrieben**, ist Global Curveball ein Verfahren zum Randomisieren von Graphen. Dabei ist als Eingabe ein beliebiger bipartiter Graph gegeben, der in einen **anderen/-**

[1] aus [2] abgerufen am 10.2.2020

zufälligen Graph mit äquivalenter Gradsequenz transformiert werden soll.
Die Aufgabe ist also, bei einer gegebenen Gradsequenz D, eine uniform verteilte **Stichprobe???** aus der Menge aller Graphen mit Gradsequenz D zurückzugeben. Durch das Ausführen von Global Curveball bleibt also für jeden Knoten $v \in V$ sein Grad $\deg(v)$ erhalten. **(aus survey übernommen)**

Ein allgemeiner Ansatz wäre... Um dies zu erreichen kann man Kanten Tauschen.... Was soll da dazu??

Curveball ist **nun** ein Prozess, der ähnlich zum Kanten Tauschen ist. Bei einem Curveball-Tausch werden zwei verschiedene Knoten u und v ($u \neq v$) zufällig uniform verteilt ausgewählt, deren Nachbarschaft zufällig **durchmischt (oder lieber shuffle?)** wird. Da bei der **bipartiten Version** von Curveball die Knoten u und v beide aus der gleichen Partitionsklasse gezogen werden, ist sichergestellt, dass es keine Kante zwischen u und v gibt. **Somit sind die beiden Knoten nicht in der jeweils anderen Nachbarschaft enthalten, es gilt folglich $u \notin N(v)$ und $v \notin N(u)$.** Wird die komplette Nachbarschaft $N(u) \cup N(v)$ durchmischt könnte es **passieren**, dass dadurch **Multikanten** (mehrere Kanten zwischen zwei Knoten) entstehen, nämlich genau dann, wenn ein Knoten aus der gemeinsamen Nachbarschaft getauscht wird, sodass er dann in einer der Nachbarschaften doppelt vorkommt. Um dies zu Vermeiden werden ausschließlich die Knoten aus der disjunkten Nachbarschaft $N_d(u, v)$ getauscht. Ein Beispiel für solch einen Tausch ist in Abbildung 2.2 gegeben. Dabei wird der Curveball-Tausch auf den Knoten v_1 und v_2 ausgeführt.

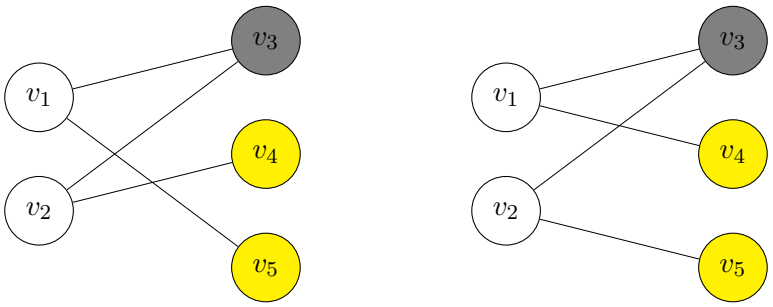

Abbildung 2.2: Beispiel eines Curveball-Tausches

Für die **(grau markierte)** gemeinsame Nachbarschaft gilt $N_c(v_1, v_2) = \{v_3\}$, die disjunkte Nachbarschaft $N_d(v_1, v_2) = \{v_4, v_5\}$ ist in gelber Farbe gekennzeichnet. In diesem Beispiel gibt es nur die zwei gegeben Graphen, die durch Tauschen der disjunkten Nachbarschaft entstehen können. **Ein Curveball-Tausch würde dann jeweils mit Wahrscheinlichkeit 0.5 einen der beiden Graphen zurückgeben.** In Abbildung 2.3 ist eine **Skizze** gegeben, wie man sich einen Curveball-

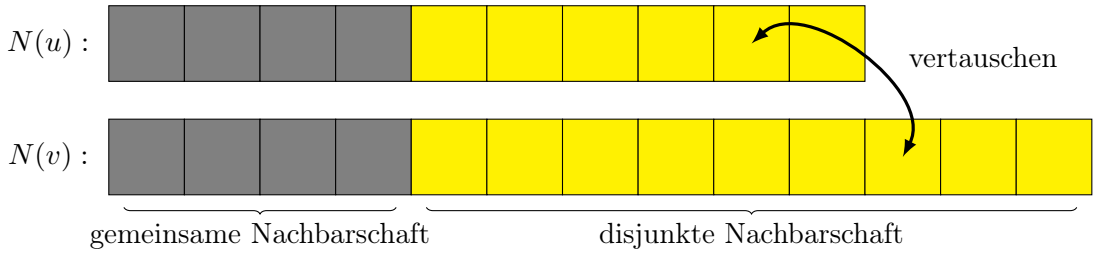

Abbildung 2.3: Curveball-Tausch auf den Vektoren

Tausch in der Datenstruktur, also den Vektoren vorstellen kann. Dabei werden zuerst die Elemente

der beiden Vektoren in gemeinsame und disjunkte Nachbarschaft aufgeteilt. Die gemeinsame Nachbarschaft ist wieder in grau gekennzeichnet, die disjunkte in gelb. Bei einem Curveball-Tausch bleiben dann die Elemente der gemeinsame Nachbarschaft unverändert, während die Elemente aus der disjunkten Nachbarschaft zufällig zwischen den beiden Vektoren getauscht werden.
Ein **Globaler Curveball-Tausch** besteht aus mehreren Curveball-Tauschen, wobei möglichst jeder Knoten aus der **Partition** Teil eines Curveball-Tausches sein soll. In Abbildung 2.4 ist eine

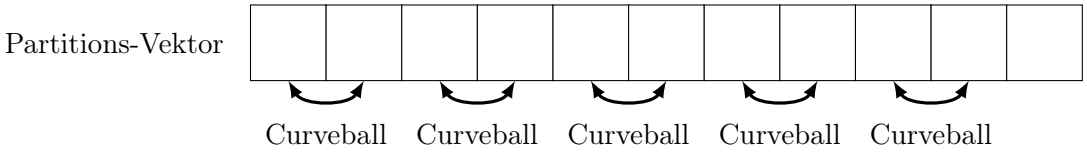

Abbildung 2.4: Global Curveball auf dem Partitions Vektor

Skizze des Partitionsvektors gegeben. Für einen Global Curveball Tausch wird dieser Vektor zuerst **geshuffelt (zufällig permutiert)**, sodass jedes Element an einer zufälligen Position steht. Dann wird jeweils parallel ein Curveball-Tausch auf den Elementen eins und zwei, drei und vier, usw. ausgeführt. Durch das zufällige Permutieren des Vektors zu Beginn, wird also jeder der Curveball-Tausche auf zwei zufälligen Knoten ausgeführt. Hat der Partitions-Vektor eine ungerade Anzahl an Elementen, bleibt am Ende ein Element übrig, welches nicht Teil von einem Curveball-Tausch ist. Bei der Parallelisierung an dieser Stelle, wird ausgenutzt, dass der zugrundeliegende Graph bipartit ist. Da es innerhalb der Partition keine zwei Knoten gibt, welche durch eine Kante miteinander verbunden sind, wird bei einem Curveball-Tausch auf beliebigen Knoten u und v die Nachbarschaft eines anderen Knotens x aus der gleichen Partition nicht geändert. Somit **"überschneiden" sich** die verschiedenen Curveball-Tausche nicht, man kann sie also **gleichzeitig laufen lassen**.

Im vollständigen Randomisierungs-Algorithmus werden schließlich mehrere solcher Global Curveball Tausche nacheinander durchgeführt.**Wie viele?!**

Man KONVERGIERT zu einem uniform sample...?

2.5 Parallelisierung

muss hier noch was dazu?!

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 **SeachSet**, 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 (**SeachUSet**).

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

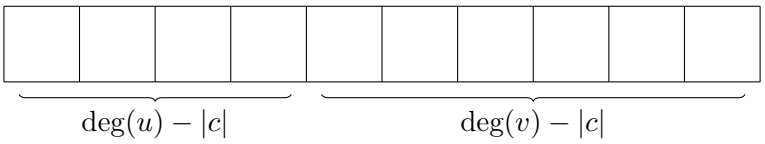

Abbildung 3.1: ist das Beispiel unnötig?

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?

INVRIANTE NAME?!

wie wir in kapitel xxx sehen, sind die varianten blblal am schnellsten. daher gehen wir an dieser stelle nochmals auf diese methoden ein indem wir sie im Pseudocode beschreiben

4 Experimentelle Untersuchung

- Versuche beschreiben. Aufbau, was warum wie wo?
- was wurde erwartet, wie passt das ergebnis dazu? was bedeutet es?
- google Test/ google benchmark
- bestimmung der schnellsten Varianten..
  – disjoint neighbors
  – trade
  – auf welcher Maschine? gluten
  plots

Wie im vorherigen Kapitel beschrieben, existieren verschiedene Varianten, einen Global Curveball Tausch durchzuführen. Für das Finden der gemeinsamen Nachbarschaft betrachten wir sieben verschiedene Methoden, für das Tauschen der Nachbarschaft zwei und weiterhin prüfen wir noch, ob es sinnvoll ist, die **vorsortiert** Invariante zu nutzen oder nicht, was ebenfalls zwei Möglichkeiten entspricht. Kombiniert man all diese Möglichkeiten erhält man also insgesamt 28 verschiedene Varianten für einen Global Curveball Tausch. In diesem Kapitel diskutieren wir, welche der Varianten ausgewählt wurde.

4.1 Versuchsaufbau

Um die einzelnen Varianten auf Ihre Laufzeit zu testen, wurde eine Art Versuch aufgebaut. Dazu wurden alle Methoden in C++ programmiert. Diese wurden dann auf unterschiedlichen Instanzen getestet und mittels Google Benchmark [3] wurde die Zeit gemessen, die für das Ausführen benötigt wurde.

Google Benchmark ist ein Framework ...?

Wie beschrieben benötigen die Methoden als Eingabe keinen Graph, sondern lediglich zwei Vektoren, welche jeweils die Nachbarschaft zweier Knoten repräsentieren. Ohne Beschränkung der Allgemeinheit nennen wir den größeren (sofern einer der beiden Vektoren größer ist) u und den kleineren v. Um möglichst gut zu erkennen, wie sich die verschiedenen Methoden bei unterschiedlichen Eingaben verhalten, messen wir die Laufzeiten für eine ganze Reihe an Instanzen. Um ein gutes Bild zu erhalten, sollten folgende Fälle auf jeden Fall abgedeckt sein:

- Beide Vektoren liegen in der gleichen Größenordnung
- Einer der Vektoren ist wesentlich größer als der andere
- Der Anteil an gemeinsamen Nachbarn ist groß
- Der Anteil an gemeinsame Nachbarn ist klein

Um dies zu erreichen, **erstellen** wir mehrere Runden, in denen der Vektor u von anfänglich 128 Elementen auf bis zu 4.000.000 Elementen vergrößert wird. Innerhalb jeder Runde werden mehrere Durchläufe durchgeführt, bei denen der Vektor u eine Größe zwischen 32 Elementen und der jeweiligen Größe von v hat. Für jeden dieser Durchgänge werden die beiden Vektoren mit zufälligen, aber paarweise verschiedenen, Werten befüllt, bis sie die entsprechende Größe haben. Dabei haben die Vektoren aber offensichtlich keine Elemente gemeinsam, was dazu führen würde, dass ein Global Curveball Tausch nichts verändern würde. Um sicherzugehen, dass die gemeinsamen Nachbarschaft nicht leer ist, müssen somit Elemente des einen Vektors in den anderen hineinkopiert werden. Damit die Größe der gemeinsamen Nachbarschaft variiert wird, werden zuerst 10, dann 25, 50 und 75 Prozent der Elemente kopiert.
Eine einzelne Test Messung sich also durch das Tupel **(large, small, fraction)** beschreiben, wobei **large** die Größe von u ist, **small** die Größe von v und **fraction** der Anteil der gemeinsamen Elemente. Um eventuelle Messfehler zu minimieren, wird jeder Durchlauf durch Google Benchmark 5 mal wiederholt.

Alle Messungen wurden **auf einem Rechner** mit **63GB** Arbeitsspeicher und 16 Prozessoren vom Typ Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz, welche jeweils 8 Kerne und einen Cache von 20 MB haben, ausgeführt.

benchmark mintime..?
WIE WERDEN DIE Varianten BEZEICHNET..?

mit hilfe von Jupyter notebook ausgewertet... daten aus json file

4.2 Messung

Insgesamt wurde für 672 Instanzen die Laufzeit der einzelnen Methoden gemessen. Die gesamte Dauer hat dabei ungefähr 19 Stunden betragen. Die Varianten werden mit einem Tripel bezeichnet, wobei In Abbildung 4.1 ist ein Balkendiagramm gegeben, was anzeigt, welche Methoden pro In-

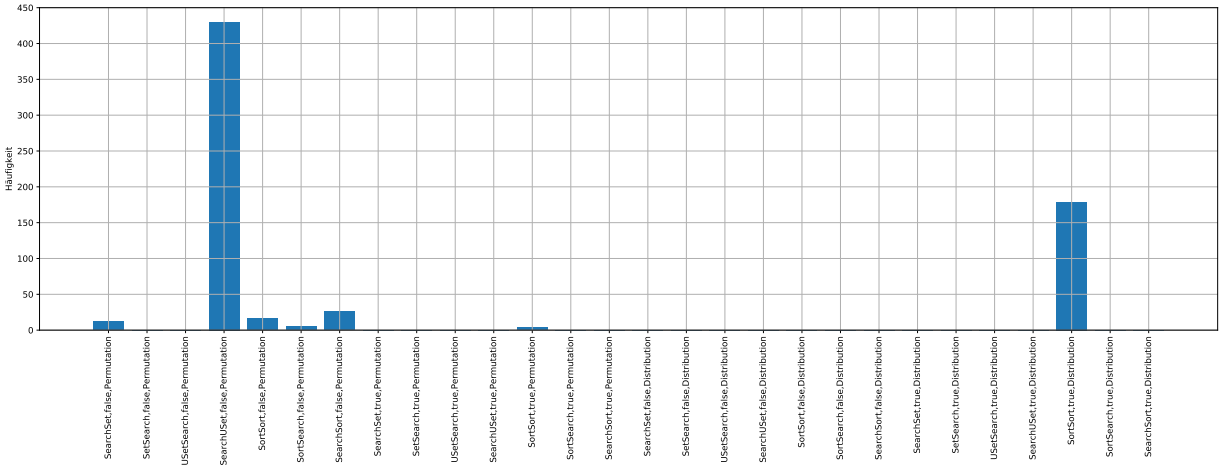

Abbildung 4.1: **Häufigkeit**

stanz am häufigsten die schnellsten sind. Dabei sieht man eindeutig, dass die Variante (**SeachUSet**, false,**Permutation**) mit Abstand auf den meisten Instanzen die schnellste Laufzeit aller Methoden hat. Die 430 Instanzen, auf denen (**SeachUSet**,false,**Permutation**) die schnellste Methode ist, entsprechen einem Anteil von rund 64%. Mit 179 ”gewonnenen” Instanzen folgt die Variante

(**SortSort**,true,**Distribution**), was einem Anteil von 27% entspricht. Zusammen ist somit in etwa 91% aller getesteter Instanzen eine dieser beiden Methoden die schnellste gewesen.

wie ist die laufzeit von sortsort, in den instanzen wo das andere gewinnt?? vielleicht ist es ja nicht viel langsamer....

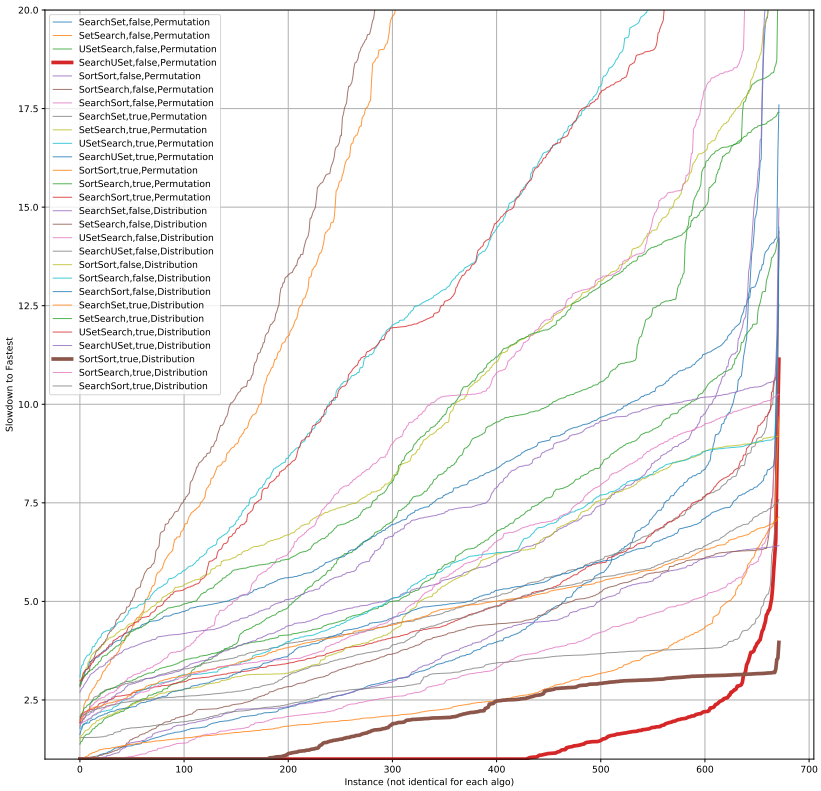

Abbildung 4.2: Slowdown

4.3 Auswertung

5 Implementierung

hier dann nur code ?!
pseudocode für alles so
welche latex umgebung für den code ?!

5.1 Networkit

google test?!

6 Zusammenfassung?

was hat das alles gebracht? ausblick? was könnte man verbessern? mehre tests, andere maschinen
was ist die Laufzeit von einem Global Curveball tausch im mittel? für große graphen?
Curveball bezeichnungen undso einheitlich?!

Literaturverzeichnis

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

[2] https://networkit.github.io/

[3] https://github.com/google/benchmark

Abbildungsverzeichnis

2.1 Beispiel eines bipartiten Graphen 5
2.2 Beispiel eines Curveball-Tausches 7
2.3 Curveball-Tausch auf den Vektoren 7
2.4 Global Curveball auf dem Partitions Vektor 8

3.1 ist das Beispiel unnötig? 10

4.1 **Häufigkeit** 13
4.2 Slowdown 14