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

3. März 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

- 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 $E \subseteq \{\{u, v\} |\ u, v \in V, u \neq v\}$.

Definition 2.2 (Multigraph)**.**
Ein Multigraph G ist ein Graph, in dem zwischen zwei Knoten mehrere Kanten existieren können. Gibt es zwischen zwei Knoten mehrere Kanten, werden diese als Multikanten bezeichnet.

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

Definition 2.3 (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 Partitionsklassen bezeichnet.

Ein Beispiel für einen bipartiten Graphen sieht man in Abbildung 2.1. Dabei gilt für die Partitionsklassen: $V_1 = \{v_1, v_2, v_3, v_4\}$ und $V_2 = \{v_5, v_6, v_7, v_8\}$. Man sieht deutlich, dass alle Kanten die Partitionenklassen V_1 und V_2 „kreuzen“.
überleitung Für einen Curveball-Tausch ist vor allem der Begriff der Nachbarschaft, genauer der gemeinsamen und disjunkten Nachbarschaft, entscheidend.

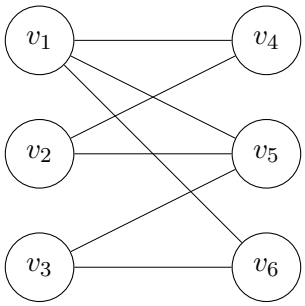

Abbildung 2.1: Beispiel eines bipartiten Graphen

Definition 2.4 (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 2.5 (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. Diesen Fakt werden wir beim Bipartiten Global Curveball ausnutzen. Zuletzt sind noch die Begriffe Knotengrad und Gradsequenz relevant.

Definition 2.6 (Knotengrad)**.**
Der **Grad** eines Knotens $v \in V$ wird mit $\deg(v)$ bezeichnet und entspricht der Anzahl der adjazenten Knoten von v. Es gilt also $\deg(v) = |N(v)|$ für alle Knoten $v \in V$.

Definition 2.7 (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 den Grad $\deg(v_1) = 3$. Für die Gradsequenz des Graphen gilt: $D = (3, 2, 2, 2, 3, 2)$.

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 hier fehlt noch was. . .

2.3 Datenstruktur

In NetworKit werden Graphen in einer eigenen Datenstruktur gespeichert. Dabei handelt es sich um eine Art Adjazenzlistendarstellung, bei der für jeden Knoten in einem Array die Nachbarn gespeichert sind. Die einzelnen Knoten sind ebenfalls in einem Array gespeichert. Da wir jedoch ausschließlich auf bipartiten ungerichteten Graphen arbeiten, können wir eine einfachere Datenstruktur nutzen. Dazu transformieren wird den den Graph, indem wir lediglich die Knoten aus einer der beiden Bipartitionsklassen **die größere? oder egal? soll ich der besser einen namen geben? welchen?** in einem Array speichern. Dieses bezeichnen wir als **Partitions-Array**. Für jeden dieser Knoten wird jeweils ein Array erstellt, indem alle adjazenten Knoten gespeichert sind. Wenn V_1 die ausgewählte Partitionsklasse ist, dann gibt es also für jeden Knoten $v \in V_1$ ein Array, welches die Elemente aus $N(v)$ enthält. Mit dieser Datenstruktur lassen sich effizient zwei zufällige Knoten der einen Partitionsklasse auswählen, die (disjunkten und gemeinsamen) Nachbarschaften berechnen und Knoten aus der disjunkten Nachbarschaft tauschen.**fehlt hier noch was?**

[1] aus [2] abgerufen am 10.2.2020

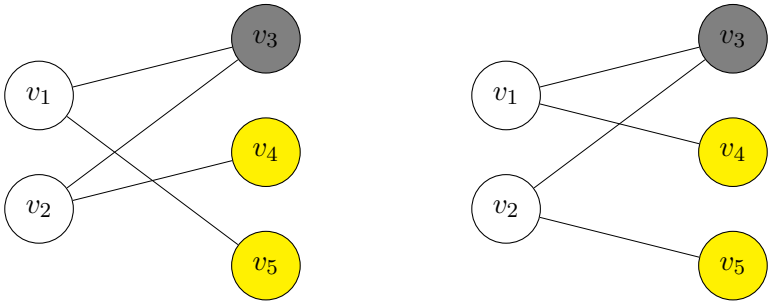

Abbildung 2.2: Es wird ein Curveball-Tausch auf den Knoten v_1 und v_2 ausgeführt. Für die grau markierte gemeinsame Nachbarschaft gilt $N_c(v_1, v_2) = \{v_3\}$, die disjunkte Nachbarschaft $N_d(v_1, v_2) = \{v_4, v_5\}$ ist in gelber Farbe gekennzeichnet. In diesem Beispiel gibt es nur die zwei gegeben Graphen, die durch Tauschen der disjunkten Nachbarschaft entstehen können. Ein Curveball-Tausch würde dann jeweils mit Wahrscheinlichkeit 0.5 einen der beiden Graphen zurückgeben.

2.4 Global Curveball (auf bipartiten Graphen)

Wie **in der Einleitung beschrieben**, ist Global Curveball ein Verfahren zum Randomisieren von Graphen. Dabei ist im Allgemeinen als Eingabe ein beliebiger Graph gegeben, der in einen zufälligen Graph mit identischer 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.
Bei einem bipartiten Global Curveball ist der Eingabegraph offensichtlich bipartit. Dadurch entstehen Vorteile, die ausgenutzt werden können. Zuerst behandeln wir jedoch einen einzelnen Curveball, welcher die Grundlage eines jeden Global Curveball darstellt.

Curveball ist ein Prozess, bei dem Kanten zufällig getauscht werden. Bei einem Curveball-Tausch werden zwei verschiedene Knoten u und v ($u \neq v$) zufällig uniform verteilt ausgewählt, deren Nachbarschaft zufällig durchmischt wird. Da bei der bipartiten Version von Curveball die Knoten u und v immer 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 und wieder auf $N(u)$ und $N(v)$ aufgeteilt, könnte es passieren, dass dadurch Multikanten entstehen, nämlich genau dann, wenn ein Knoten aus der gemeinsamen Nachbarschaft getauscht wird, sodass er danach 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. Wie ein Curveball-

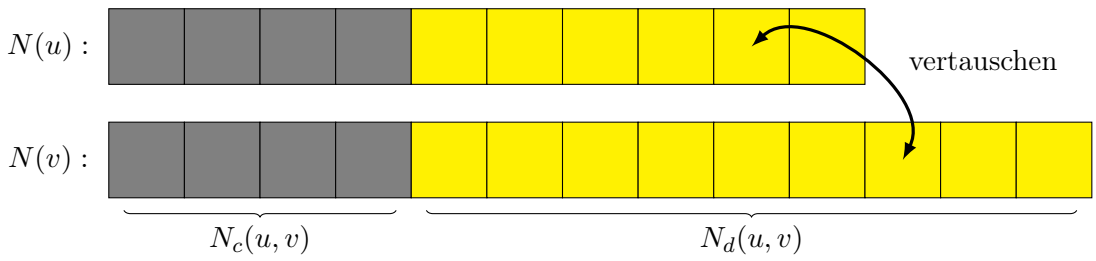

Abbildung 2.3: Skizze eines Curveball-Tausches auf den Arrays

Tausch in der Datenstruktur, also den beiden Arrays aussieht, ist in Abbildung 2.3 skizziert. 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 **Global Curveball-Tausch** besteht aus mehreren Curveball-Tauschen, wobei möglichst jeder Knoten aus der **Partitionsklasse** Teil eines Curveball-Tausches sein soll. In Abbildung 2.4 ist eine

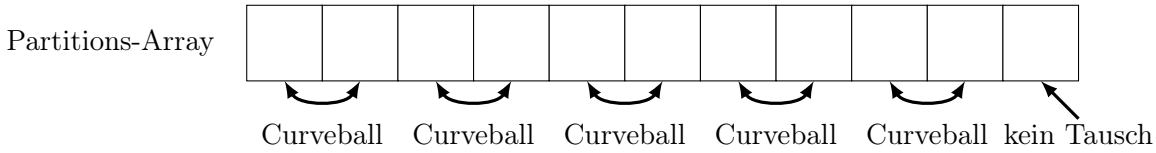

Abbildung 2.4: Global Curveball auf dem Partitions-Array

Skizze des Partitions-Arrays gegeben. Für einen Global Curveball Tausch wird das Array zuerst zufällig permutiert, sodass jedes Element an einer zufälligen Position steht. Dann wird jeweils unabhängig ein Curveball-Tausch auf den Elementen eins und zwei, drei und vier, usw. ausgeführt. Durch das zufällige Permutieren des Arrays zu Beginn, wird also jeder der Curveball-Tausche auf zwei zufälligen Knoten ausgeführt. Hat das Partitions-Array eine ungerade Anzahl an Elementen, bleibt am Ende ein Element übrig, welches nicht Teil von einem Curveball-Tausch ist. Hierbei können wir ausnutzen, dass der Eingabegraph bipartit ist. Da es innerhalb des **Partitions-Arrays** 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 Partitionklasse nicht verändert. Somit „überschneiden“ sich die einzelnen Curveball-Tausche nicht. Man kann sie daher zeitgleich, also parallel, ausführen. **Diese Parallelität führt natürlich zu einem Laufzeitvorteil.**
Der hauptsächliche Unterschied zwischen Global Curveball auf allgemeinen und bipartiten Graphen liegt also darin, dass die einzelnen Curveball-Tausche sich gegenseitig nicht beeinflussen und daher parallel behandelt werden können.

Im vollständigen Randomisierungs-Algorithmus werden schließlich mehrere solcher Global Curveball Tausche nacheinander durchgeführt. Die genaue Anzahl lässt sich beim Aufrufen des Algorithmus durch einen Parameter festlegen.

Um zu beweisen, dass das mehrfache Anwenden vom Bipartiten Global Curveball eine uniform verteilte Stichprobe aller Graphen mit gleicher Gradsequenz erzeugt, kann man das Verfahren als Markov-Kette simulieren. Dabei entsprechen die Zustände allen Graphen, welche eine identische Gradsequenz wie der Urspungsgraph haben. Zwischen zwei Zuständen gibt es genau dann einen Übergang, wenn die beiden Graphen durch einen Global Curveball-Tausch ineinander überführbar sind. **Es lässt sich zeigen,** dass diese Markov-Kette aperiodisch, irreduzibel und symmetrisch ist [**?**] **welche quelle?**. Ebenso ist die Markov-Kette endlich, da die Anzahl an Graphen mit gegebener Gradsequenz beschränkt ist. In [**?**]**welche quelle?** wird bewiesen, dass solche Markov-Ketten **zu einer uniformen Verteilung auf den Zuständen konvergiert.**

3 Beschreibung eines Curveball-Tausches

3.1 Bestimmung der gemeinsamen Nachbarschaft

Wie bereits in Kapitel 2.4 erwähnt muss die disjunkte Nachbarschaft der beiden Knoten u und v bekannt sein, um einen Curveball-Tausch auf diesen Knoten u und v auszuführen. Der Curveball-Tausch besteht dann darin, diese Knoten aus $N_d(u, v)$ zu durchmischen.

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

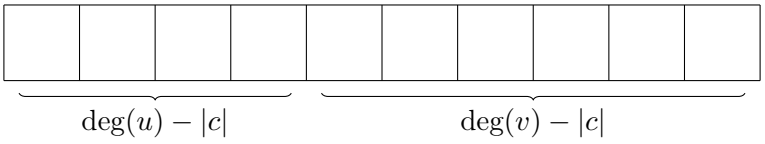

Abbildung 3.1: ist das Beispiel unnötig?

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?

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

	Distribution		Permutation	
	true	**false**	**true**	**false**
SortSort				
SearchSort				
SortSearch				
SearchSet				
SetSearch				
SearchUSet				
USetSearch				

Tabelle 3.1: Jedes Feld in der Tabelle entspricht einer Variante für einen Curveball-Tausch

Algorithm 1 SortSort

```
1:  procedure SortSort(u, v)
2:      U ← N(u)                                   ▷ Nachbarschaft von u
3:      V ← N(v)                                   ▷ Nachbarschaft von v
4:      sort(U)                                    ▷ Sortiere beide Vektoren
5:      sort(V)
6:      nu ← 0                                     ▷ Zähler für U
7:      nv ← 0                                     ▷ Zähler für V
8:      while (nu < U.size()) and (nv < V.size()) do
9:          if U[nu] < V[nv] then
10:             disjoint.append(U[nu])
11:             nu ++
12:         else if U[nu] > V[nv] then
13:             disjoint.append(V[nv])
14:             nv ++
15:         else if U[nu] == V[nv] then
16:             common.append(U[nu])
17:             nu ++
18:             nv ++
19:     if nu ≠ U.size() then
20:         disjoint.append( U[nu], U[nu+1], ... )
21:     else
22:         disjoint.append( V[nv], V[nv+1], ... )
23:     return common, disjoint
```

Algorithm 2 Distribution

```
1:  procedure DISTRIBUTION(common, disjoint)
2:      nu ← U.size() - common.size()
3:      nv ← V.size() - common.size()
4:      i = 0
5:      while i < disjoint.size() do
6:          X ∼ B(nu/(nu+nv))      ▷ ziehe ein Zufallsbit Bernoulli verteilt mit Wahrscheinlichkeit nu/(nu+nv)
7:          if X==1 then
8:              U.append(disjoint[i])
9:              i++
10:             nu- -
11:         else
12:             V.append(disjoint[i])
13:             i++
14:             nv- -
15:     U.insert(common)
16:     V.insert(common)
17:     return U, V
```

4 Experimentelle Untersuchung

- **was wurde erwartet, wie passt das ergebnis dazu? was bedeutet es?**

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 [**?**] 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.

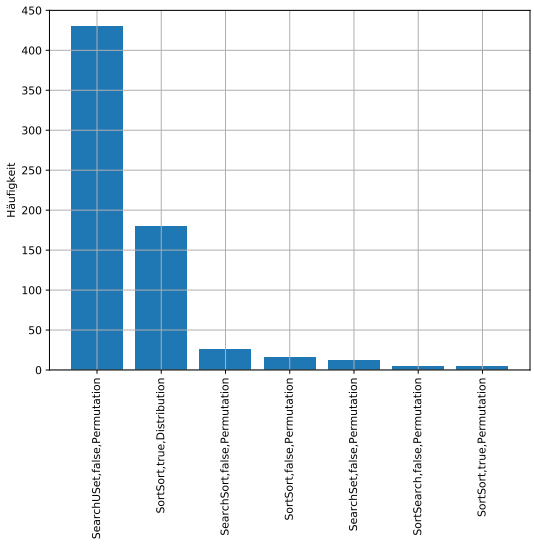

Abbildung 4.1: Welche Variante ist am häufigsten die schnellste?

Eine einzelne Test Messung lässt sich somit durch ein Tripel **(large, small, fraction)** beschreiben, wobei **large** die Größe von u ist, **small** die Größe von v und **fraction** der Anteil der gemeinsamen Elemente.

4.2 Messung

Auf die im vorherigen Abschnitt beschriebene Weise, werden die verschiedenen Instanzen erstellt und mittels Google Benchmark die Zeit gemessen. Aus Zeitgründen werden jedoch nicht alle Werte für large und small erstellt. Deshalb verdoppeln wir in jedem Schritt die Werte von large und small anstatt sie um eins zu inkrementieren. Somit ergeben sich insgesamt 672 Instanzen, auf denen die Laufzeiten der einzelnen Methoden gemessen werden. Um eventuelle Messfehler zu minimieren, wird jeder Durchlauf durch den Google Benchmark Parameter `benchmark_repetitions` 5 mal wiederholt. Mit dem Parameter `benchmark_min_time` wird noch festgelegt, dass jede Variante so oft getestet wird, bis mindestens eine Gesamtlaufzeit von 0.1 Sekunden **erreicht** wird. Alle Messungen wurden **auf einem Rechner** mit **64GB** 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. Mit dieser Konfiguration hat die Dauer für alle 28 Varianten in Summe ungefähr 19 Stunden betragen.

4.3 Auswertung

Die ermittelten Messdaten werden schließlich als `json`-Datei gespeichert. Mit Hilfe von Jupyter Notebook lassen sie sich auswerten. Dabei handelt es sich um ein **Tool**, mit dem man Python-Programme auf einfache Art und Weise erstellen und ausführen kann. Innerhalb von Python werden wir die Bibliotheken **Matplotlib** und **pandas** nutzen, um die Daten zu analysieren und grafisch aufzuarbeiten.
Zuerst betrachten wir für jede Instanz, welche Methode am schnellsten war, also für welche die geringste Laufzeit gemessen wurde. Interessant sind dann jeweils die Methoden, die häufig am schnellsten waren. In Abbildung 4.1 ist dazu ein Balkendiagramm gegeben. Dabei sieht man eindeutig, dass die Variante (**SearchUSet**, **false**, **Permutation**) mit Abstand auf den meisten In-

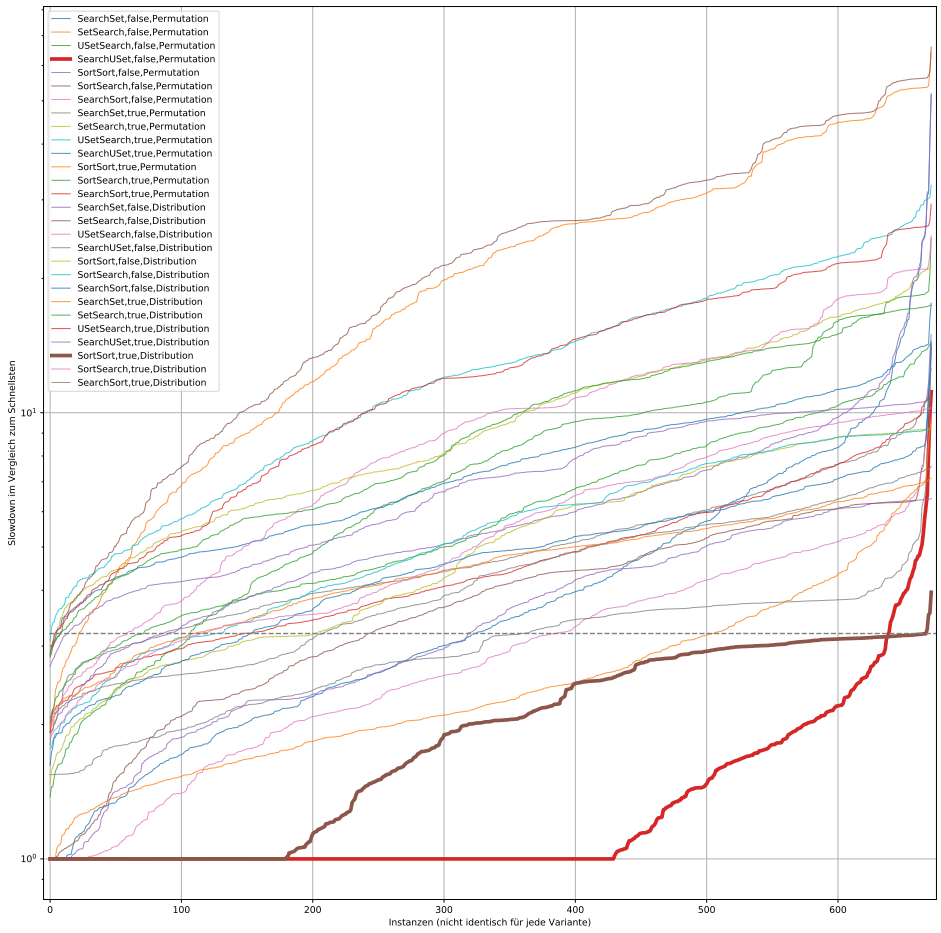

Abbildung 4.2: **Slowdown im Vergleich zur schnellsten Variante**

stanzen die schnellste Laufzeit aller Methoden hat. Die 430 Instanzen, auf denen (**SearchUSet**, **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. Daher liegt der Schluss nahe, sich beim Suchen der „besten“ Variante, auf diese beiden Methoden zu beschränken. Um nicht fälschlicherweise „gute“ Methoden auszuschließen betrachten wir einen weiteren Plot in Abbildung 4.2. Um diesen Plot zu erstellen wurde jede Variante einzeln betrachtet. Dann wird für jede Instanz bestimmt, welche Variante die kürzeste Laufzeit hat und der Quotient aus dieser Laufzeit und der Laufzeit der betrachteten Variante berechnet. Dieses Verhältnis wird als **Slowdown** bezeichnet und gibt an, um welchen Faktor die Variante langsamer als die schnellste ist. Die Slowdowns zu jeder Instanz werden schließlich aufsteigend sortiert und als **Kurve** in den Plot eingefügt. Da die Slowdowns jedoch für jede Variante unabhängig voneinader sortiert werden, geht dadurch die Ordnung über die Instanzen verloren. Somit entspricht eine Stelle auf der horizontalen Achse nicht für jede Variante der gleichen Instanz. Dieser Plot legt ebenfalls nahe, sich auf die beiden Varianten (**SearchUSet**, **false**, **Permutation**) und (**SortSort**, **true**, **Distribution**) zu konzentrieren, da die beiden Kurven am wenigsten stark **wachsen** und damit die Slowdowns vergleichsweise klein sind. Jedoch lässt sich auch hier nicht bestimmen, welche der beiden Varianten die bessere ist. Ein Vorteil von (**SearchUSet**, **false**, **Permutation**) ist, dass die Methode auf den meisten Instanzen einen Slowdown von eins hat – was wir ja auch schon in Abbildung 4.1 gesehen haben – und damit langsamer anwächst. Ein Nachteil liegt

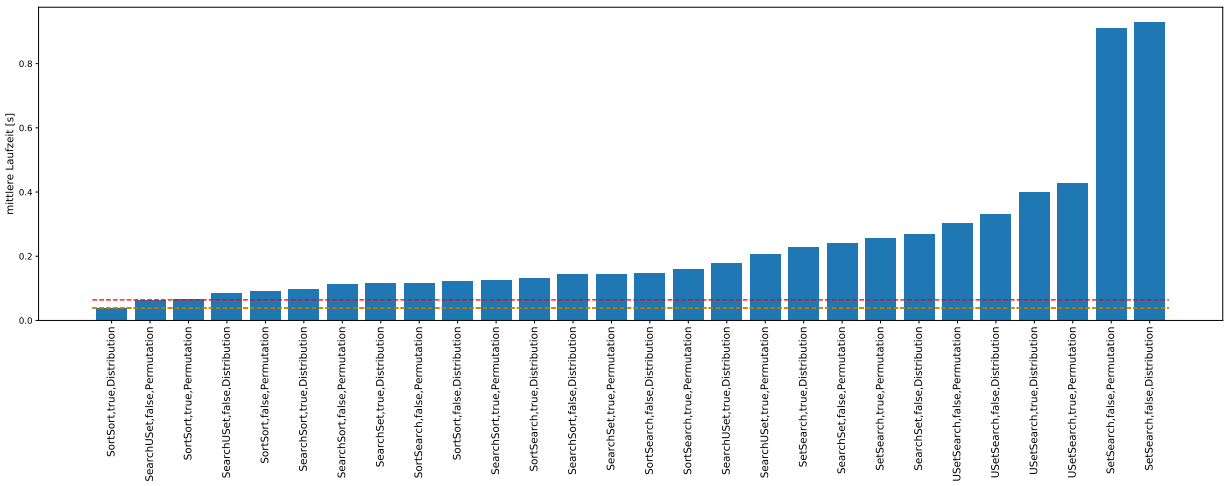

Abbildung 4.3: Mittlere Laufzeiten der Varianten über allen Instanzen

aber darin, dass der Slowdown für manche Instanzen eine Größe von bis zu 10.0 erreicht, während der Slowdown von (**SortSort**, **true**, **Distribution**) durch den maximalen Wert von 3.5 beschränkt ist (in Abbildung 4.2 als gestrichelte Linie eingezeichnet).
Dieser Nachteil spiegelt sich ebenfalls in Abbildung 4.3 wieder. In diesem Plot wurden über alle Instanzen für jede Variante jeweils die mittlere Laufzeit bestimmt. Obwohl es nicht in den meisten Fällen die schnellste Variante ist, hat (**SortSort**, **true**, **Distribution**) die kleinste mittlere Laufzeit mit rund 0.0387 Sekunden. Auf Platz zwei folgt (**SearchUSet**, **false**, **Permutation**) mit etwa 0.0640 Sekunden, was schon einer Abweichung von ungefähr 60% entspricht. Auch in diesem Plot sieht man deutlich, dass es sich nicht lohnt noch weiter Varianten zu betrachten. Zwar hat auch (**SortSort**, **true**, **Permutation**) eine mittlere Laufzeit, die annähernd so groß ist wie (**SearchUSet**, **false**, **Permutation**), die aber **nicht an die schnellste herankommt**. Alle anderen Varianten haben eine deutlich größere mittlere Laufzeit. Abschließend betrachten wir noch die zwei ausgewählten Varianten im direkten Vergleich. Dazu wurden in Abbildung 4.4 auf der horizontalen Achse die getesteten Instanzen aufgetragen und dazu die jeweiligen Laufzeiten von (**SortSort**, **true**, **Distribution**) und (**SearchUSet**, **false**, **Permutation**) als Kreuze eingezeichnet. Die Instanzen sind nach aufsteigenden Werten für small sortiert. Aus Gründen der Übersichtlichkeit wird bei der Beschriftung der Instanzen der Teil fraction weggelassen, die Instanzen werden nur mit large, small bezeichnet. Weiterhin wurden der Übersichtlichkeit wegen die Instanzen aus dem Bereich $32 < \text{small} < 32768$ ausgelassen, da sie das gleiche Bild wie die üblichen Instanzen zeigen. Man sieht dabei, dass sich die Laufzeiten in den meisten Instanzen nicht so stark unterscheiden. Dies sind vor allem die Instanzen, bei denen der Unterschied zwischen large und small nicht so groß ist. In den Instanzen, in denen sich die Werte für small und large stark unterscheiden, ist jedoch ein deutlicher Vorteil von (**SortSort**, **true**, **Distribution**) zu erkennen. Auch bei den „großen“ Instanzen mit Werten von $\text{small} > 500.000$ hat diese Variante einen deutlichen Laufzeitvorteil. Auf der Instanz (4.194.304, 4.194.304, 75) beispielsweise hat (**SearchUSet**, **false**, **Permutation**) eine Laufzeit von circa 1.628 Sekunden, während (**SortSort**, **true**, **Distribution**) nur etwa 0.146 Sekunden benötigt. Der Slowdown beträgt für diese Instanz also in etwa 11.

4.4 Fazit

Abschließend muss an Hand der Messdaten entschieden werden, welche Variante zum Einsatz für einen Curveball-Tausch am Besten geeignet ist. Zusammenfassend haben wir im vorherigen Abschnitt festgestellt, dass nur die Varianten (**SortSort**, **true**, **Distribution**) und (**SearchUSet**,

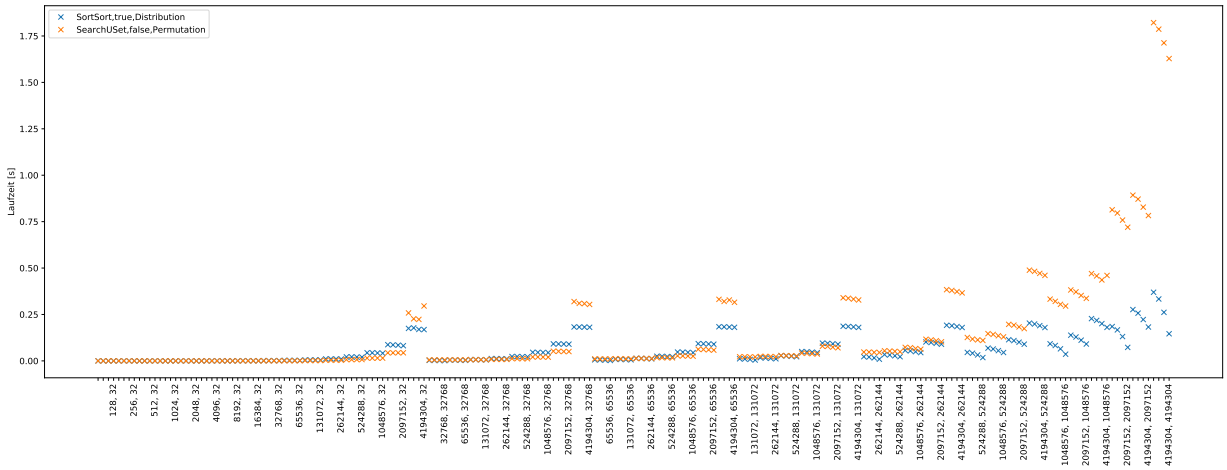

Abbildung 4.4: Vergleich der Laufzeiten zweier Varianten auf ausgewählten Instanzen

false, **Permutation**) zur Auswahl stehen. Während die eine Variante am häufigsten die geringste Laufzeit hat, liegt die andere bei der durchschnittlichen Laufzeit weiter vorne. Dies liegt vor allem daran, dass sich die Laufzeiten bei den Instanzen auf denen (**SearchUSet**, **false**, **Permutation**) „gewinnt“ kaum unterscheiden. Unter den anderen Instanzen gibt es jedoch welche, bei denen (**SortSort**, **true**, **Distribution**) bis auf einen Faktor von circa 11 schneller ist.
Um ein bestmögliches Laufzeitverhalten für einen Curveball-Tausch zu erreichen, könnte man auf die Idee kommen, beide Varianten zusammen zu nutzen. Man könnte sich eine **Heuristik** überlegen, die jeweils angibt, auf welcher Instanz man welche Variante verwenden sollte. Somit würde bei jedem Curveball-Tausch abhängig von der Eingabe, also den Nachbarschaften, entschieden werden, welche der beiden Instanzen man benutzt. Das Problem dabei liegt jedoch darin, dass bei diesen Varianten einmal die Vorsortiert-Invariante genutzt wird und einmal nicht. Dies ist natürlich nicht beides gemeinsam möglich. Entweder man hält die Nachbarschaften immer sortiert, oder eben nicht. Man muss sich also für eine Möglichkeit der Invariante entscheiden. Dadurch ändert sich dann aber zwangsläufig eine der beiden Varianten. Entscheidet man sich, die Nachbarschaften sortiert zu halten, würde (**SearchUSet**, **false**, **Permutation**) zu (**SearchUSet**, **true**, **Permutation**) werden, andernfalls würde sich (**SortSort**, **true**, **Distribution**) zu (**SortSort**, **false**, **Distribution**) verändern. Diese beiden „veränderten “ Varianten haben aber jeweils deutlich schlechtere Laufzeiten als die ursprünglichen.
Es ist also nicht sinnvoll möglich, die beiden Varianten miteinander zu kombinieren. Wir müssen uns also auf eine Variante festlegen. Dies ist die Variante (**SortSort**, **true**, **Distribution**), da sie im Vergleich zur Alternative – wie schon beschrieben – in kaum einer Instanz eine wesentlich schlechtere Laufzeit hatte, jedoch auf manchen Instanzen wesentlich bessere Laufzeiten. Außerdem ist es die Variante mit der geringsten mittleren Laufzeit. Ein Vorteil dieser Variante neben der Laufzeit liegt noch in der Einfachheit. So werden lediglich die beiden Arrays sortiert und linear durchlaufen. Es muss keine weitere Datenstruktur erstellt werden – wie bei der anderen Variante das `unordered_set` – und damit wird auch kein zusätzlicher Speicherplatz verbraucht.

4.5 Einordnung?! passen die ergebnisse zu erwartungen?!

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?!

Anführungszeichen suchen!! + Leerzeichen dahinter

Laufzeit test für das python ding.. wie schnell funktioniert der algo so? **wie siehts aus mit der anzahl an global trades?!**

Deckblatt?!

noch was rotes? oder was blaues?
vektor/array?!

Literaturverzeichnis

[1] Corrie Jacobien Carstens, Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. Parallel and i/o-efficient randomisation of massive networks using global curveball trades. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, volume 112 of *LIPIcs*, pages 11:1–11:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[2] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016.

Abbildungsverzeichnis