Bachelorarbeit mit dem Thema

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

verfasst von:

Marius Hagemann

Matrikelnummer 5732742
marius@ing-hagemann.de

7. 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

Ziel dieser Bachelorarbeit ist es, einen **effizienten** Algorithmus zur Randomisierung massiver bipartiter Graphen zu entwickeln. Dazu wurde das Konzept des Global Curveball für bipartite Graphen angepasst. Es werden verschiedene Möglichkeiten zur Umsetzung diskutiert. Anhand von Benchmarks wird unter den getesteten Methoden diejenige ausgewählt, welche die geringste Laufzeit aufweist. Im Vergleich zu dem schon existierenden Global Curveball Algorithmus wird mit dem in dieser Arbeit entwickelten Algorithmus auf manchen Testinstanzen ein Speedup von bis zu 17 erreicht.

1 Einleitung

„Bei der Analyse komplexer Netzwerke, wie beispielsweise soziale Netzwerke, werden die zugrundeliegenden Graphen häufig mit zufälligen Graphen verglichen, um deren Struktur zu untersuchen.“[1]

Zum Erzeugen von zufälligen Graphen existieren diverse Modelle wie beispielsweise der Erdos-Renyi-Graph [5] oder der Gilbert-Graph [6]. **Diesen beiden Methoden ...?** Dabei weisen jedoch die erstellten Graphen kaum eine Ähnlichkeit zu dem zu analysierenden Netzwerk auf. Deshalb sind meist Zufallsgraphen gesucht, die zu einem gegebenen Graphen eine identische Gradsequenz besitzen. Im Zufallsgraph soll also jeder Knoten denselben Grad haben wie im originalen Graph.

Ein Algorithmus, welcher diese Eigenschaft erfüllt, ist beispielsweise Curveball [3]. Hierbei wird ein Graph „randomisiert, indem eine Sequenz an lokalen Modifikationen ausgeführt wird“[2]. Diese lokalen Modifikationen werden als Curveball-Tausch bezeichnet. Bei einem Curveball-Tausch werden von zwei zufälligen Knoten die disjunkten Nachbarschaften durchgetauscht. Damit bleiben die Knotengrade unverändert. Um den Graph zu randomisieren, werden einige von diesen Curveball-Tauschen hintereinander auf jeweils zufälligen Knoten ausgeführt. Um sicherzugehen, dass alle Knoten Teil eines Curveball-Tausches waren, werden auf diese Weise „in Erwartung $\Theta(n \log(n))$ Curveball-Tausche benötigt“[3]. Um dies zu umgehen, **wurde** eine Erweiterung namens Global Curveball [4] eingeführt. Ein Global Curveball Tausch ist ein „Super-Schritt“[2], in dem mehrere Curveball-Tausche auf jeweils unterschiedlichen Knoten nacheinander ausgeführt werden, sodass möglichst jeder Knoten Teil eines solchen Tausches ist. Somit werden lediglich $\Theta(n)$ viele Curveball-Tausche benötigt, um sicherzugehen, dass möglichst alle Knoten abgedeckt sind.

Ziel dieser Bachelorarbeit ist die Anpassung von Global Curveball an bipartite Graphen zur Reduzierung der Laufzeit. Zum einen werden gewisse Eigenschaften von bipartiten Graphen ausgenutzt, sodass Teile des originalen Global Curveball Algorithmus vereinfacht werden können. Zum anderen ist es möglich, einzelne Curveball-Tausche parallel auszuführen. Auf diese Weise soll — wie bereits erwähnt — eine deutlich geringere Laufzeit im Vergleich zu dem ursprünglichen Global Curveball Algorithmus erreicht werden. Der neu entwickelte Algorithmus wird unter dem Namen `BipartiteGlobalCurveball` Teil vom Open-Source Projekt NetworKit werden.

[1] frei übersetzt aus [4] Abschnitt 1
[2] frei übersetzt aus [7] Abschnitt 6.4
[3] frei übersetzt aus [4] Abschnitt 3.3

Zu Beginn dieser Arbeit werden die wichtigsten Begriffe und mathematischen Grundlagen definitiert. Im Anschluss werden verschiedenen Methoden, Global Curveball umzusetzen, aufgezeigt und **unter Einbeziehung theoretischer Aspekte verglichen.** Mit Hilfe von Benchmark Tests wird schließlich die Methode ausgewählt, welche die geringste Laufzeit aufweist.

2 Grundlagen

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 spielen 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 ist in Abbildung 2.1 dargestellt. Dabei gilt für die Partitionsklassen: $V_1 = \{v_1, v_2, v_3\}$ und $V_2 = \{v_4, v_5, v_6\}$. 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.

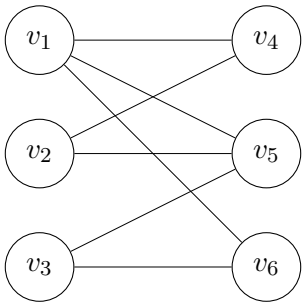

Abbildung 2.1: Beispiel eines bipartiten Graphen

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 $N_d(u, v) = \big[N(u) \cup N(v)\big] \setminus \big[N(u) \cap N(v)\big]$. Weiterhin gilt $N_c(u, v) \cap N_d(u, v) = \emptyset$ und $N_c(u, v) \cup N_d(u, v) = N(u) \cup N(v)$. Jeder Knoten $x \in [N(u) \cup N(v)]$ aus den beiden Nachbarschaften liegt also entweder in der gemeinsamen oder in der disjunkten Nachbarschaft.

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 $d_i = \deg(v_i)$ dem Grad des Knotens v_i entspricht.

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 [8] 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 Global Curveball (auf bipartiten Graphen)

Global Curveball ist ein Verfahren zum Randomisieren von Graphen. Die Aufgabe liegt also darin, 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

[1] **aus [8]**

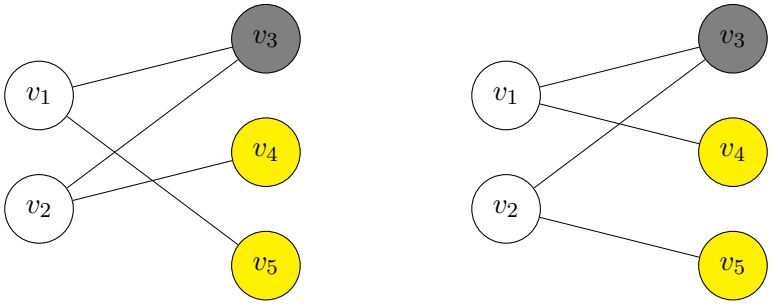

Abbildung 2.2: Es wird ein Curveball-Tausch auf den Knoten v_1 und v_2 ausgeführt. Für die grau markierte gemeinsame Nachbarschaft gilt $N_c(v_1, v_2) = \{v_3\}$, die disjunkte Nachbarschaft $N_d(v_1, v_2) = \{v_4, v_5\}$ ist in gelber Farbe gekennzeichnet. In diesem Beispiel gibt es nur die zwei gegeben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

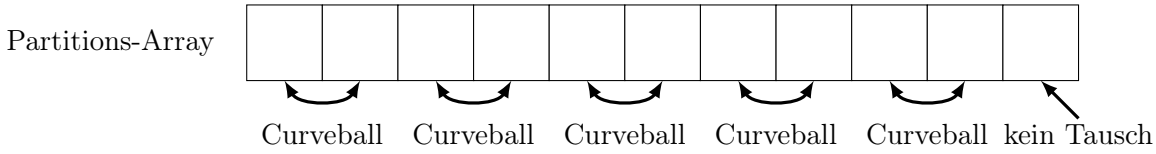

Abbildung 2.3: Global Curveball auf dem zufällig permutierten Partitions-Array

für jeden Knoten $v \in V$ sein Grad $\deg(v)$ erhalten.
Bei einem bipartiten Global Curveball ist der Eingabegraph bipartit. Dadurch entstehen Vorteile, die algorithmisch 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 Variante 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 $N(u) \cup N(v)$ **(das vielleicht einfach weglassen?)** 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.
Ein **Global Curveball-Tausch** besteht aus mehreren Curveball-Tauschen, wobei möglichst jeder Knoten aus der **Partitionsklasse auch nochmal gucken** Teil eines Curveball-Tausches sein soll. In Abbildung 2.3 ist eine Skizze des Partitions-Arrays gegeben. Für einen Global Cur-

veball 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 der Partitionsklasse **hier nochmal gucken..** 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 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 vollständig 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 interpretieren. 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 [7] **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.**

2.4 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 in einem Array speichern. Welche der beiden Partitionsklassen ausgewählt wird, bleibt dem **Nutzer** überlassen. Dabei ist es jedoch sinnvoll, die Klasse mit der geringeren Anzahl an Knoten zu wählen, da auf diese Weise weniger Curveball-Tausche auszuführen sind. Dieses Array werden wir als **Partitions-Array** bezeichnen. **wenn im folgenden von partition gepsrochen wird: diese gemeint** Für jeden dieser Knoten wird jeweils ein Array erstellt,

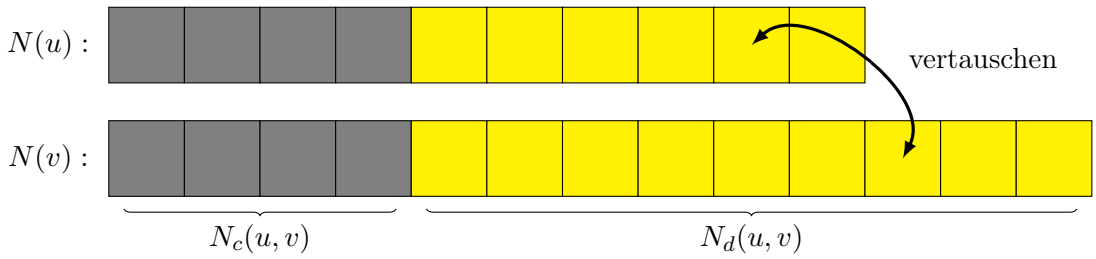

Abbildung 2.4: Skizze eines Curveball-Tausches auf den Arrays

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.

Wie ein Curveball-Tausch in der Datenstruktur, also den beiden Arrays aussieht, ist in Abbildung 2.4 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 Arrays getauscht werden.

3 Implementierung eines Curveball-Tausches

Wie bereits in Kapitel 2.3 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. Deshalb beschäftigen wir uns zuerst damit, wie man die disjunkte Nachbarschaft bestimmt.

3.1 Bestimmung der disjunkten Nachbarschaft

Gesucht sind alle Knoten aus der disjunkten Nachbarschaft der Knoten u und v. Nach der Definition 2.5 liegt jeder Knoten aus den Nachbarschaften $N(u)$ und $N(v)$ entweder in der disjunkten oder in der gemeinsamen Nachbarschaft. Das Ziel besteht also darin, für jeden Knoten aus $N(u) \cup N(v)$ zu entscheiden, ob er zu $N_d(u, v)$ oder $N_c(u, v)$ gehört.
Die Nachbarschaften $N(u)$ und $N(v)$ liegen wie in Abschnitt 2.4 beschrieben jeweils in einem Array vor. Der Übersichtlichkeit wegen, werden wir die beiden Arrays ebenfalls mit $N(u)$ und $N(v)$ bezeichnen. Die Aufgabe ist es also, für jedes Element aus den beiden Arrays zu entscheiden, ob es entweder in beiden Arrays vorkommt, oder nur in einem von den beiden. Dafür gibt es verschiedene algorithmische Ansätze.

Als ersten naiven Ansatz könnte man für jedes Element des Arrays $N(u)$ das gesamte andere Array $N(v)$ per linearer Suche nach diesem Element durchsuchen. Hierfür ergibt sich eine Laufzeit von $\mathcal{O}(|N(u)| \cdot |N(v)|)$. Dies ist aber nicht sinnvoll ist, da wir im Falle von massiven Graphen davon ausgehen können, dass die Arrays (also die Nachbarschaften) groß werden.
Dieses Problem kann man beispielsweise verhindern, indem man beide Arrays aufsteigend sortiert. Um nun zu herauszufinden, welche Werte in beiden Arrays vorkommen, muss man lediglich u und v gleichzeitig linear durchlaufen und testen, ob die Werte gleich sind, oder nicht. Somit muss man jedes Element der beiden Arrays — nach dem Sortieren — nur einmal betrachten, was offensichtlich zu einer verbesserten Laufzeit im Vergleich zum naiven Ansatz führt. Man erhält damit eine Laufzeit von $\mathcal{O}(|N(u)| \cdot \log(|N(u)|) + |N(v)| \cdot \log(|N(v)|))$. Diese Variante wird im Folgenden als **SortSort** bezeichnet.
Die Laufzeit hängt dabei im Wesentlichen vom Sortieren ab. Daher führen wir eine Variante ein, die wir **vorsortiert** nennen. Bei dieser Variante nehmen wir an, dass die Arrays immer im sortierten Zustand vorliegen. Somit würde bei **SortSort** das Sortieren wegfallen und man müsste die beiden Arrays nur noch linear durchlaufen, was zu einer Laufzeit von $\mathcal{O}(|N(u)| + |N(v)|)$ führen würde. Es ist jedoch nicht offensichtlich, dass die Variante zu einer insgesamt besseren Laufzeit eines Curveball-Tausches führt, da ein Curveball-Tausch schließlich auch noch aus dem Durchmi-

schen der disjunkten Nachbarschaft besteht. Dadurch könnte die **Vorsortierheit** verletzt werden, weshalb man am Ende des Curveball-Tausches nochmal sortieren muss, um sie wieder aufrecht zu erhalten. Ob das **Vorsortieren** zu einem Vorteil führt, wird im Kapitel 4 anhand von Laufzeitmessungen geklärt.

Die Variante **SortSort** lässt sich leicht abwandeln, indem wir nur eines der beiden Arrays sortieren. Auf diese Weise können wir die Laufzeit des einen Sortiervorgangs sparen. Sortieren wir das größere Array (ohne Beschränkung der Allgemeinheit $N(v)$), bezeichnen wir die Variante als **SortSearch**. Um zu erkennen, welche Elemente zur gemeinsamen und welche zur disjunkten Nachbarschaft gehören, kann man für jeden Knoten aus $N(u)$ per binärer Suche in logarithmischer Zeit prüfen, ob der Knoten auch in $N(v)$ vorhanden ist. Damit ergibt sich eine Laufzeit von $\mathcal{O}(|N(v)| \cdot \log(|N(v)|) + |N(u)| \cdot \log(|N(v)|)$. Analog dazu nennen wir die Variante, in der das kleinere Array sortiert wird, **SearchSort**. Auch hier könnte die vorsortiert Invariante einen Vorteil bringen.
Eine weitere Methode um viele Werte schnell zu durchsuchen, bietet die Datenstruktur `set`, welche einem binären Suchbaums entspricht, beispielsweise eines Rot-Schwarz-Baums. Dabei wird jedes Element des einen Arrays in den Suchbaum eingefügt. Für jedes Element des anderen Arrays kann nun in logarithmischer Zeit bestimmt werden, ob es im Set und somit auch im ursprünglichen Array vorhanden ist. Somit erhält man die identischen asymptotischen Laufzeiten wie bei der Verwendung der binären Suche. **Je nach Implementierung des Suchbaums könnte es aber auch zu einer verbesserten Laufzeit führen.** Für diese Möglichkeit gibt es ebenfalls zwei analoge Varianten, nämlich **SetSearch**, bei der die Elemente des größeren Arrays in das `set` eingefügt werden und **SearchSet**, bei der das kleinere Array zum `set` hinzugefügt wird. Auch bei den beiden Optionen könnte es sinnvoll sein, wenn die vorsortiert Invariante aufrecht erhalten wird. Dies hängt ebenfalls von der internen Implementierung des Suchbaums ab.
Die letzte Methode, die wir an dieser Stelle betrachten werden, 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. Ein Vorteil einer Hash-Tabelle liegt darin, dass das Einfügen und Suchen von Elementen erwartet in konstanter Zeit erfolgt. Ebenfalls gibt es hierbei wieder die Varianten, in denen das größere Array in das unordered_set eingefügt wird (**USetSearch**) oder das kleinere (**SearchUSet**). Asymptotisch ergibt sich in beiden Fällen eine erwartete Laufzeit von $\mathcal{O}(|(N(u)|+|N(v)|)$. Diese Laufzeit hängt jedoch stark von der Implementierung und dem **Füllgrad** der Hash-Tabelle ab. Schließlich werden wir auch bei den letzten beiden Methoden prüfen, ob die Invariante eventuell zu einer besseren Laufzeit führt.

Zusammenfassend betrachten wir also insgesamt sieben verschiedene Möglichkeiten um die disjunkte und gemeinsame Nachbarschaft zu berechnen. Für jede dieser Varianten prüfen wir zusätzlich, ob die vorsortiert Invariante zu einer verbesserten Laufzeit führt oder ob es sich nicht lohnt, diese aufrechtzuerhalten.

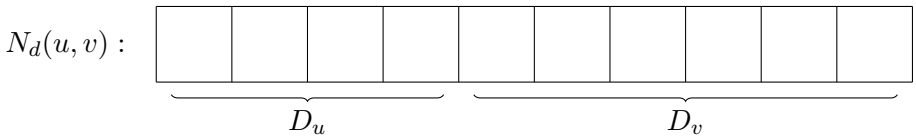

Abbildung 3.1: Beispiel für einen Tausch mit der Variante **Permutation**. Dabei wird das Array der disjunkten Nachbarschaft $N_d(u, v)$ zufällig permutiert. Die ersten D_u Elemente werden der Nachbarschaft von u zugeordnet, die restlichen D_v zur Nachbarschaft von v.

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 die Knoten der disjunkten Nachbarschaft zufällig tauscht. Als Eingabe stehen die Arrays $N_d(u, v)$, welches alle Knoten aus der disjunkten Nachbarschaft enthält und $N_c(u, v)$, das die gemeinsamen Nachbarn enthält, zur Verfügung. Weiterhin seien $\deg(u)$ und $\deg(v)$ die ursprünglichen Knotengrade. Zusätzlich definieren wir uns die Mengen $D_u = N_d(u, v) \cap N(u)$, in der die disjunkten Nachbarn von Knoten u liegen, und $D_v = N_d(u, v) \cap N(v)$, in der die disjunkten Nachbarn von v liegen. Die anfänglich leeren Arrays, in denen die Ausgabe — also die neuen Nachbarschaften von u und v — zurückgegeben werden soll, bezeichnen wir wieder als $N(u)$ und $N(v)$. Wir betrachten zwei unterchiedliche Möglichkeiten.

Die erste Idee besteht darin, das Array der disjunkten Nachbarschaft 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 Arrays $N(u)$ und $N(v)$ kopiert. Dann werden die ersten D_u Elemente aus dem permutierten Array in $N(u)$ kopiert, die restlichen in $N(v)$. Somit haben die Nachbarschaften durch den Tausch ihre ursprüngliche Größe nicht verändert, es gilt $|N(u)| = \deg(u)$ und $|N(v)| = \deg(v)$. Zur besseren Veranschaulichung ist in Abbildung 3.1 ein Beispiel zu sehen. Bei 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 D_u liegt (also zum Array $N(u)$ hinzugefügt wird) oder nicht. Auf welcher Position genau es in diesen Bereichen liegt, ist nicht von Relevanz. Ohne Beschränkung der Allgemeinheit gelte $N(u) \leq N(v)$. Dann kann man also die Laufzeit dieser Variante verbessern, indem nicht das ganze Array der disjunkten Nachbarschaft zufällig permutiert wird, sondern nur die ersten D_u **vielen** Elemente zufällig gewählt werden. Dies setzt die Funktion `random_bipartition_shuffle` um.
Ein Nachteil dieser Methode besteht jedoch darin, dass durch das zufällige Vertauschen die beiden resultierenden Arrays $N(u)$ und $N(v)$ im Allgemeinen nicht mehr sortiert sind. Damit wird die im Abschnitt 3.1 beschriebene Invariante eventuell verletzt. Möchte man die Invariante aufrecht erhalten, müssen somit die beiden Arrays in einem letzten Schritt nochmals sortiert werden. Wir nennen diese Variante **Permutation**.

Die zweite Möglichkeit, die wir betrachten, werden wir als **Distribution** bezeichnen. Die Idee besteht dabei, dass wir über jedes Element des Arrays $N_d(u, v)$ iterieren und eine Wahrscheinlichkeit berechnen, mit der das Element in die Nachbarschaft von 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 $N(u)$ oder in $N(v)$ kopiert. Dies wird so lange wiederholt, bis eines der beiden Arrays 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 Arrays u und v entsprechen, wenn die Elemente aus der gemeinsamen Nachbarschaft nicht berücksichtigt werden. **Es gilt also $n_v = |D_v|$ und $n_u = |D_u|$.** Damit hat das erste Element des Arrays $N_d(u, v)$ eine Wahrscheinlichkeit von $p_u = \frac{n_u}{n_u+n_v}$, dem Array $N(u)$ hinzugefügt zu werden und analog eine Wahrscheinlichkeit $p_v = \frac{n_v}{n_u+n_v}$, um in $N(v)$ 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 $N(u)$ kopiert. Dabei hat sich aber offensichtlich die verbleibende Kapazität des Arrays $N(u)$ verringert. Also muss der Wert n_u dekrementiert werden. Analoges gilt, falls das Element in die Nachbarschaft von 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 $n_v = 0$, steht offenbar in einem der Arrays keine freie Kapazität mehr zur Verfügung. Somit werden die übrigen Elemente, die noch in $N_d(u, v)$ vorhanden sind, einfach dem anderen Array hinzugefügt. **Optimierung um floats zu vermeiden erwähnen**
Dieses Vorgehen ist auch als Reservoir Sampling [?] wo? bekannt.
Ein Vorteil dieser Methode ist, dass die in 3.1 beschriebene Invariante aufrecht erhalten werden kann. War das Array der disjunkten Nachbarschaft vor Beginn dieser Methode aufsteigend sortiert, dann sind auch die bisherigen Elemente der Arrays $N(u)$ und $N(v)$ aufsteigend sortiert, da für jedes Element nacheinander entschieden wurde, ob es zur Nachbarschaft von u oder von v hinzugefügt wird und dabei die Reihenfolge der Elemente untereinander nicht verändert wurde.
Zum Schluss müssen noch die die gemeinsamen Nachbarn zu den Arrays $N(u)$ und $N(v)$ hinzugefügt werden. Möchte man die Invariante aufrecht erhalten, dann sind die beiden Arrays wie beschrieben schon aufsteigend sortiert. Da auch die Elemente aus $N_c(u, v)$ aufsteigend sortiert sind, erhält man die endgültigen Arrays von $N(u)$ und $N(v)$ durch ein Mergen mit $N_c(u, v)$. Soll die Invariante jedoch nicht aufrecht erhalten werden, reicht es aus, die Elemente aus $N_c(u, v)$ jeweils an das Ende der beiden Arrays zu kopieren.

Wird die Variante vorsortiert nicht verwendet, ergibt sich für **Permutation** und **Distribution** die gleiche asymptotische Laufzeit von $\mathcal{O}(|N(u)| + |N(v)|)$. Beim Verwenden der vorsortiert Variante ändert sich die asymptotische Laufzeit für **Distribution** nicht, da das mergen ebenfalls in linearer Zeit ausgeführt werden kann. Bei **Permutation** hingegen müssen die beiden Arrays noch-

	Distribution		Permutation	
vorsortiert	**true**	**false**	**true**	**false**
SortSort	$\mathcal{O}(l)$	$\mathcal{O}(l \cdot \log(l))$	$\mathcal{O}(l \cdot \log(l))$	$\mathcal{O}(l \cdot \log(l))$
SearchSort	$\mathcal{O}(l \cdot \log(s))$	$\mathcal{O}(l \cdot \log(s))$	$\mathcal{O}(l \cdot \log(l))$	$\mathcal{O}(l \cdot \log(s))$
SortSearch	$\mathcal{O}(s \cdot \log(l))$	$\mathcal{O}(l \cdot \log(l))$	$\mathcal{O}(l \cdot \log(l))$	$\mathcal{O}(l \cdot \log(l))$
SearchSet	$\mathcal{O}(l \cdot \log(s))$	$\mathcal{O}(l \cdot \log(s))$	$\mathcal{O}(l \cdot \log(l))$	$\mathcal{O}(l \cdot \log(s))$
SetSearch	$\mathcal{O}(s \cdot \log(l))$	$\mathcal{O}(l \cdot \log(l))$	$\mathcal{O}(l \cdot \log(l))$	$\mathcal{O}(l \cdot \log(l))$
SearchUSet	$\mathcal{O}(l)\dagger$	$\mathcal{O}(l)\dagger$	$\mathcal{O}(l \cdot \log(l))$	$\mathcal{O}(l)\dagger$
USetSearch	$\mathcal{O}(l)\dagger$	$\mathcal{O}(l)\dagger$	$\mathcal{O}(l \cdot \log(l))$	$\mathcal{O}(l)\dagger$

Tabelle 3.1: Jedes Feld in der Tabelle entspricht einer Variante für einen Curveball-Tausch. Dabei sind jeweils die asymptotischen Laufzeiten der Varianten gegeben. Zur besseren Übersicht sind die Laufzeiten nicht in Abhängigkeit von $|N(u)|$ und $|N(v)|$ angegeben, sondern in den Variablen l und s, wobei l der Größe des größeren Arrays entspricht und s der Größe des kleineren. Es gilt also $l = \max\{|N(u)|, |N(v)|\}$ und $s = \min\{|N(u)|, |N(v)|\}$ und folglich $s = \mathcal{O}(l)$. Erwartete Laufzeiten sind dabei mit † markiert.

mals sortiert werden. Somit entsteht eine Laufzeit von $\mathcal{O}(|N(u)| \cdot \log(|N(u)|) + |N(v)| \cdot \log(|N(v)|))$.

3.3 Curveball-Tausch

Wie schon in Abschnitt 2.3 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.
Die verschiedenen Methoden, die wir hierfür untersuchen werden, entstehen durch Kombination aller Varianten, die in 3.1 und 3.2 beschrieben werden. Alle diese Möglichkeiten sind in der Tabelle 3.1 mit ihren jeweiligen asymptotischen Laufzeiten zusammengefasst.

3.4 Pseudocode

Wie wir in Kapitel 4.5 sehen werden, hat die Variante mit den Methoden **SortSort** und **Distribution** und der genutzten vorsortiert Invariante das beste Laufzeitverhalten. Deswegen gehen wir an dieser Stelle noch einmal genauer auf diese beiden Methoden ein, indem wir sie in Pseudocode beschreiben. In Algorithmus 1 ist **SortSort** beschrieben. **...noch was dazu schreiben...?**

Algorithm 1 SortSort

```
1:  procedure SortSort(u, v)
2:      U ← N(u)                                   ▷ vorsortierte Nachbarschaft von u
3:      V ← N(v)                                   ▷ vorsortierte Nachbarschaft von v
4:      nu ← 0                                     ▷ Zähler für U
5:      nv ← 0                                     ▷ Zähler für V
6:      while (nu < |U|) and (nv < |V|) do
7:          if U[nu] < V[nv] then
8:              disjoint.append(U[nu])             ▷ Füge das Element in disjoint ein
9:              nu ++
10:         else if U[nu] > V[nv] then
11:             disjoint.append(V[nv])             ▷ Füge das Element in disjoint ein
12:             nv ++
13:         else if U[nu] == V[nv] then
14:             common.append(U[nu])               ▷ Füge das Element in common ein
15:             nu ++
16:             nv ++
17:     if nu ≠ U.size() then                      ▷ Die restlichen Elemente sind disjunkte Nachbarn
18:         disjoint.append( U[nu], U[nu+1], ... )
19:     else
20:         disjoint.append( V[nv], V[nv+1], ... )
21:     return common, disjoint
```

Algorithm 2 Distribution

```
1:  procedure Distribution(common, disjoint)
2:      nu ← U.size() - common.size()              ▷ Kapazität von u
3:      nv ← V.size() - common.size()              ▷ Kapazität von v
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])              ▷ Füge das Element in U ein
9:              i++
10:             nu- -                              ▷ aktualisiere die Kapazität
11:         else
12:             V.append(disjoint[i])              ▷ Füge das Element in V ein
13:             i++
14:             nv- -                              ▷ aktualisiere die Kapazität
15:     U.merge(common)                            ▷ Merge U mit der gemeinsame Nachbarschaft
16:     V.merge(common)                            ▷ Merge V mit der gemeinsame Nachbarschaft
17:     return U, V
```

4 Experimentelle Untersuchung

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 ein Versuch aufgebaut. Dazu wurden alle Methoden in C++ programmiert. Diese wurden dann auf unterschiedlichen Instanzen getestet und mittels Google Benchmark [1] wurde die Zeit gemessen, die für das Ausführen benötigt wurde.

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 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 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 prozentuale 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 einem Prozessor vom Typ Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz, mit 8 Kernen und einem Cache von 20 MB, 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 [2] 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 Instanzen 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 Vari-

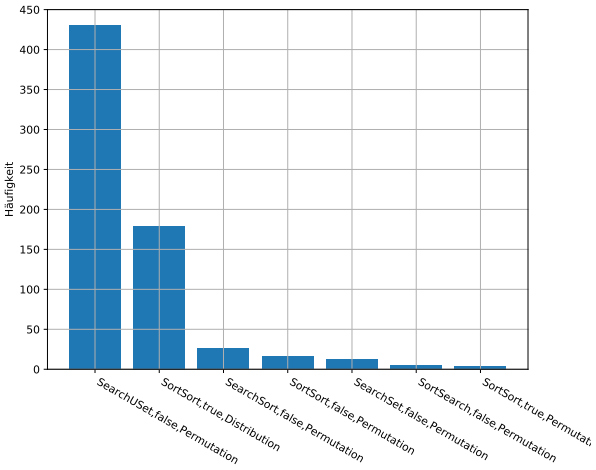

Abbildung 4.1: Vergleich der Varianten, welche am häufigsten die geringste Laufzeit hatten

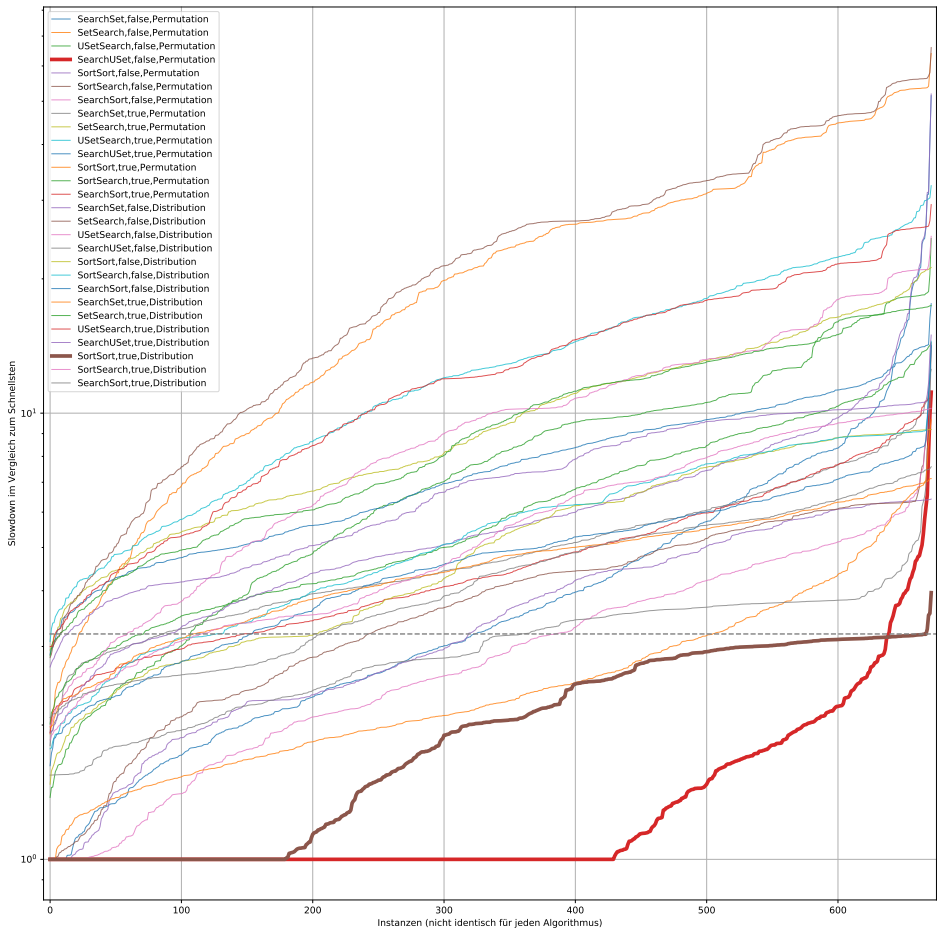

Abbildung 4.2: **Slowdown im Vergleich zur schnellsten Variante**

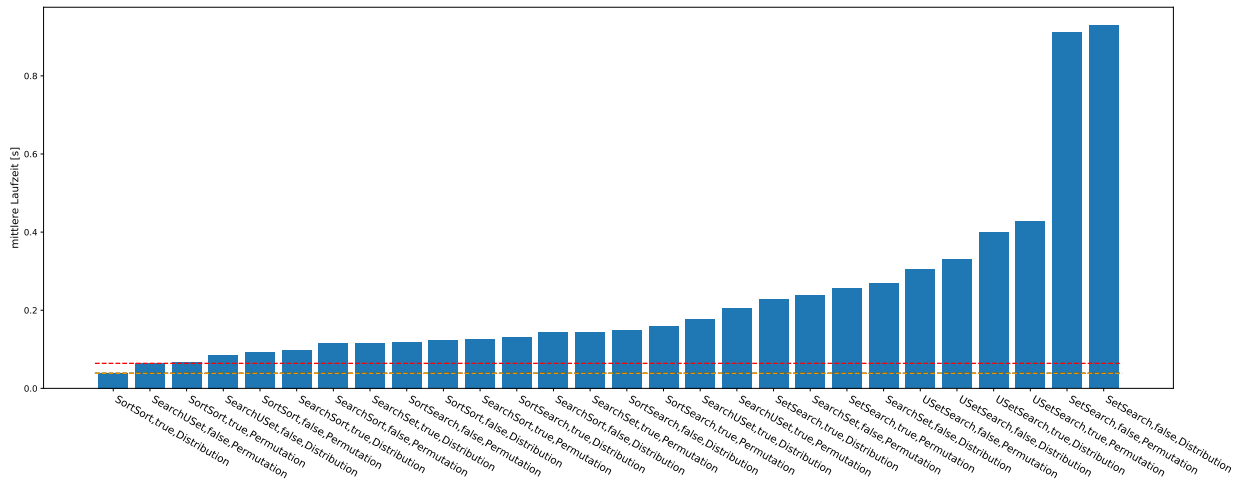

Abbildung 4.3: Mittlere Laufzeiten der Varianten über allen Instanzen

ante 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 voneinander 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 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

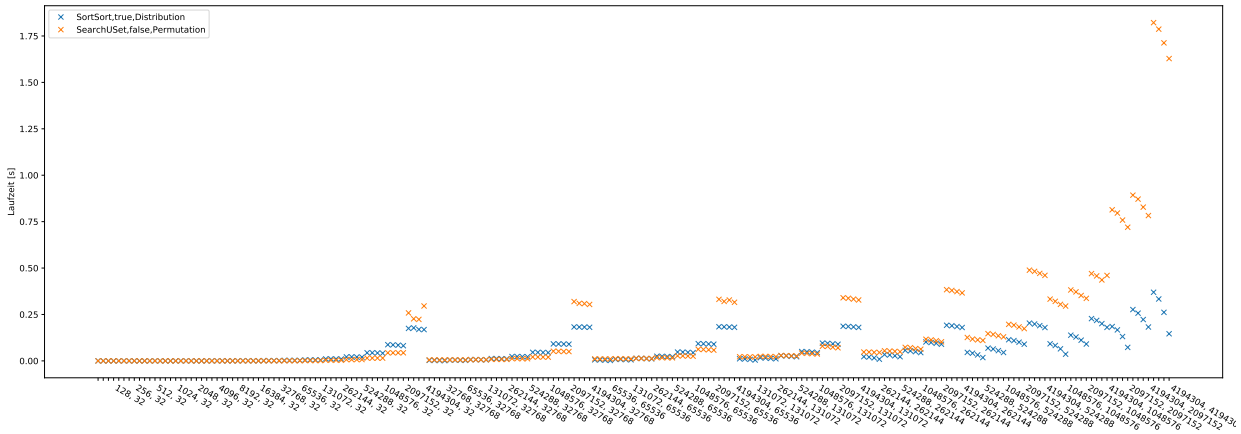

Abbildung 4.4: Vergleich der Laufzeiten zweier Varianten auf ausgewählten Instanzen

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 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 Diskussion der Ergebnisse

Nun werden wir überprüfen, ob die ermittelten Ergebnisse zu erwarten waren, indem wir sie mit den Asymptotischen Laufzeiten aus der Tabelle 3.1 vergleichen. Laut den Asymptotischen Laufzeiten sollte die Varianten mit **SortSort**, **Distribution** und vorsortiert am schnellsten sein. Dies sieht man auch bei den mittleren gemessenen Laufzeiten aus Abbildung 4.3. Dabei liegt diese Varianten deutlich auf dem ersten Platz. Die dazugehörige Variante mit der Tausch-Methode **Permutation** liegt auf dem dritten Platz, obwohl sie im Vergleich eine der schlechtesten asymptotischen Laufzeiten hat. Der Grund für diese Diskrepanz liegt vermutlich darin, dass diese asymptotische Laufzeit durch das einmalige Sortieren am Ende der Methode entsteht. In anderen Varianten, die die gleiche asymptotische Laufzeit haben, wird jedoch eventuell häufiger sortiert oder eine Datenstruktur verwendet, die eine höhere Laufzeit produziert.
Auffällig ist, dass die Varianten mit **USetSearch** jeweils deutlich langsamer sind, als die analogen Varianten, welche **SearchUSet** verwenden, obwohl die erwarteten asymptotischen Laufzeiten gleich sind. Dies liegt höchstwahrscheinlich an der Datenstruktur `unordered_set`, einer Hash-Tabelle.

Bei dieser Datenstruktur sind die Laufzeiten stark abhängig vom Füllgrad, also dem Anteil der gespeicherten Elemente im Bezug zur Größe der Hash-Tabelle. Da bei **USetSearch** die Elemente des größeren Arrays in die Hash-Tabelle eingefügt werden, ist der Füllgrad dementsprechend auch größer, was zu der schlechteren Laufzeit führt. Im Gegensatz dazu, ist die Methode, welche **SearchUSet**, **Permutation** und keine Vorsortierung verwendet, die im Mittel zweitschnellste aller gemessenen Varianten, da sich hierbei weniger Elemente in der Hash-Tabelle befinden und der Füllgrad demnach geringer ist. Somit wird auch eher die Laufzeit des Erwartungswerts erreicht.
Mit einem ähnlichen Argument lassen sich auch die schlechten Laufzeiten der Varianten, welche **SetSearch** verwenden, erklären. Hierbei wird ebenfalls das größere Array in die Datenstruktur eingefügt, was zu der schlechteren Laufzeit führt. Dabei fällt jedoch noch auf, dass vor allem die Varianten, bei welchen nicht vorsortiert wird, die mit Abstand längsten Laufzeiten besitzen. Als Begründung dient hier **vermutlich**, dass das Erstellen eines Binärbaums bei sortierten Elementen in dieser Implementierung des Baumes dem Anschein nach wesentlich schneller ist, als bei beliebigen Elementen.

Im Großen und Ganzen lassen sich also die gemessenen mittleren Laufzeiten gut erklären. **Man kann die Ergebnisse somit als vertrauenswürdig einstufen.**

4.5 Auswahl der besten Variante

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**, **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.

4.6 Überschrift..?

In einem letzten Experiment wird geprüft, ob das Ziel, einen Randomisierungs Algortihmus für bipartite Graphen zu entwickeln, welcher eine geringere Laufzeit, als Global Curveball aufweist, erreicht wurde.
Dazu werden erstellen wir verschiedene Test-Instanzen. **Als Grundlage dient ein Netflix Datensatz, welcher Nutzer- und Film-Knoten enthält, wobei eine gerichtete Kante von einem Film zu einem Nutzer gezogen wird, wenn dieser den Film gut bewertet hat.** Damit liegen die Film-Knoten in einer Partitionsklasse und die Nutzer-Knoten in der anderen. Aus diesem Graph werden drei Subgraphen erstellt, wobei jeweils 1000, 10.000 und 100.000 zufällige Nutzer-Knoten ausgewählt werden und deren vollständige Nachbarschaft hinzugefügt wird. Für jeden dieser drei Graphen gibt es zwei Varianten. Der Unterschied zwischen den beiden Varianten liegt darin, welche der beiden Partitionsklassen aktiv sind, also auf welcher Partitionsklasse die Curveball-Tausche ausgeführt werden. Somit ergeben sich sechs Instanzen. Auf diesen wird **die in dieser Arbeit erarbeite** bipartite Version des Global Curveball ausgeführt. Dazu müssen die einzelnen Graphen jedoch in einen ungerichteten Graph transformiert werden. Weiterhin wird auch der Standard **Curveball Algorithmus was wurde angepasst?** sowohl auf den ungerichteten als auch auf den gerichteten Graphen ausgeführt.
In Abbildung 4.5 sind die gemessenen Laufzeiten grafisch dargestellt. Auf der horizontalen Achse sind die einzelnen Instanzen als Tripel aufgetragen. Die erste Stelle steht dabei für die Knotenanzahl des Graphen, die zweite Stelle für die Anzahl an Kanten und die dritte für die Anzahl der Knoten aus der aktiven Partition. Um eventuelle Messfehler zu verringern, wurde jede Messung zehn mal wiederholt. Als Datenpunkte sind jeweils Mittelwerte der Messungen aufgetragen. Zusätzlich sind die jeweiligen Standardabweichungen als Fehlerbalken eingetragen. Da diese im Vergleich zu den Messwerten aber relativ klein sind, sind sie in der Grafik kaum zu erkennen.
Dabei fällt auf, dass sich die Laufzeiten von Curveball und Bipartitem Global Curveball auf den Instanzen mit über 100.000 deutlich unterscheiden. Auf diesen Instanzen erreicht der bipartite Global Curveball einen Speedup von bis zu 17. Während die Laufzeit der angepassten Curveball Variante auf dem gerichteten Graphen in etwa 9 Sekunden und auf dem ungerichteten etwa 10,5 Sekunden beträgt, liegt sie beim bipartiten Global Curveball bei lediglich 0,6 Sekunden. Dies entspricht — auf dieser Instanz — einem Speedup von ungefähr 17. Auf den kleineren Instanzen Die Laufzeit Dabei fällt auf.... deutlcih schneller... speedup auf Instanz mit 100.000 bis zu 17! also ist die um

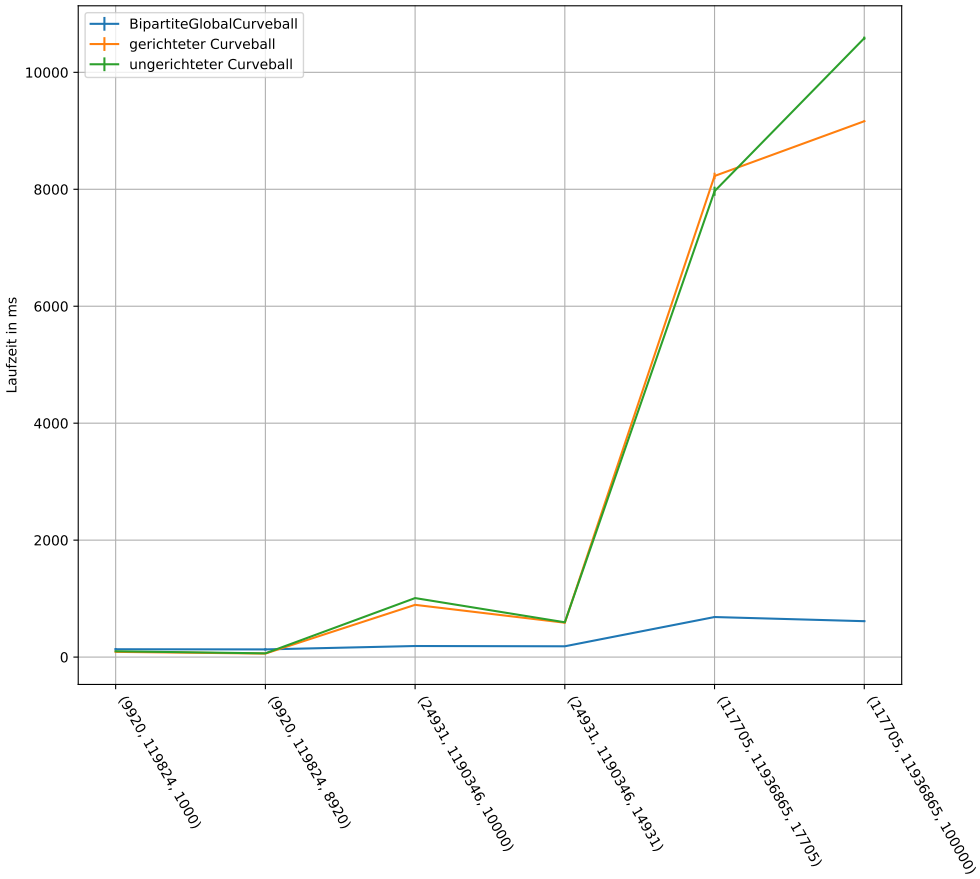

Abbildung 4.5: Laufzeit Vergleich von BipartiteGlobalCurveball und Originalem Curveball

das 17 fache laufzeit gewinn! von

5 Fazit?

Zum Abschluss dieser Arbeit lässt sich feststellen, dass das Ziel, einen Algorithmus zum Randomisieren massiver bipartiter Graphen zu entwickeln, welcher eine bessere Laufzeit aufweist, als der schon bekannte Global Curveball Algorithmus, erreicht wurde.

Dies ist — wie in Kapitel **??** besprochen — gelungen

Jedoch könnte man auch noch Verbesserungen vornehmen:

Im Bezug auf die experimentelle Auswertung zur Bestimmung der schnellsten Methode einen Curveball-Tausch umzusetzen, könnte man noch folgende Verbesserungen untersuchen. Zum Einen werden die Varianten nur auf Instanzen getestet, welche Nachbarschaften mit maximal 4 Millionen Knoten enthalten. Je Nachdem,

Wir haben gesehen, wie ein Global Curveball Tausch und damit auch ein Curveball-Tausch aufgebaut ist.

std in fehlerbalken aber so klein, dass sie nicht sichtbar sind.

Verbessern könnte man noch: verschiedene benchmarks, größere instanzen, einfach mehr...

auf anderem Rechner Architektur/system testen

hier auf jeden Fall noch die Python Laufzeiten undso

Laufzeit Vergleiche... Ausblick?!

was hat das alles gebracht?

Algo wird Teil von networkit?

ausblick?

was könnte man verbessern? mehre tests, andere maschinen

was ist die Laufzeit von einem Global Curveball tausch im mittel? für große graphen?

TODO:

- Einleitung
- Zusammenfassung
- **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?!**
  – Seitenrand
  –Zeilenabstand
  – Schriftgröße
  –
- **noch was rotes?** oder was blaues?
- **vektor/array?!**
- **LaTeXformat ?!**

Literaturverzeichnis

[1] Google benchmark. `https://github.com/google/benchmark`. **Abgerufen am 04.03.2020**.

[2] Jupyter notebook. `https://jupyter.org/`. **Abgerufen am 04.03.2020**.

[3] Corrie Jacobien Carstens, Annabell Berger, and Giovanni Strona. Curveball: a new generation of sampling algorithms for graphs with fixed degree sequence. *CoRR*, abs/1609.05137, 2016.

[4] 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. 112:11:1–11:15, 2018.

[5] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae Debrecen*, 1959.

[6] E. N. Gilbert. Random graphs. *Ann. Math. Statist.*, 30(4):1141–1144, 12 1959.

[7] Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in scalable network generation, 2020.

[8] 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

Tabellenverzeichnis