Projektgruppe: Abschlussbericht
Optimierung von Algorithmen für Minimale Spannbäume

Ramtin Azimi, Florian Gläßer, Alex Stüben

27. September 2015

Zusammenfassung

Dieser Bericht ist die Abschlussarbeit der Projektgruppe zur „Optimierung von Algorithmen für minimale Spannbäume“ im Sommersemester 2015 an der Universität Bonn. Mittels der Implementierung von diversen Algorithmen aus [1] und [2] sollen die Laufzeit von dem Kruskal und Prim-Algorithmus verbessert werden. Der Abschlussbericht gibt einen Überblick über die Algorithmen und ihrer Implementierung. Außerdem wird deren Laufzeitperformanz mit bisher bekannten Algorithmen verglichen.

1 Einleitung

1.1 Motivation

Minimale Spannbäume finden in vielen Bereichen der Informatik, wie in Telekommunikationsnetzwerken oder Transportnetzwerken, ihre Anwendung. Im Rahmen der Projektgruppe „Optimierung von Algorithmen für Minimale Spannbäume“ unter der Leitung von Prof. Dr. Heiko Röglin, Michael Etscheid und Carsten Fischer im Sommersemester 2015 haben wir uns mit der Optimierung der Algorithmen Kruskal und Prim auseinandergesetzt.
Osipov et al. [2] führen den Filter-Kruskal zur Verbesserung von Kruskal ein. Die Arbeit von Navarro et al. [1] erreicht dieses Ziel mittels einer verbesserten Sortiermethoden, dem *IncrementalQuickSort*. Außerdem beschäftigen

sich Navarro und Paredes mit der Optimierung von Prim durch die Einführung von *Quickheaps*, einer Datenstruktur zur Implementierung von Prioritätswarteschlangen.
Das Ziel des Abschlussberichts wird es sein, die in den Arbeiten [2] und [1] vorgestellten Algorithmen und unsere Implementierung hierzu zu erläutern. Die Laufzeiten werden im nächsten Schritt mit anderen effizienten Algorithmen verglichen.

1.2 Struktur der Abschlussarbeit

Um einen ersten Eindruck von dem Forschungsfeld zu erhalten, wird eine kurze Zusammenfassung über die bisher diskutierten Algorithmen aus der Literatur präsentiert. Die Einführung in die Thematik wird durch formelle Definition, die in Bezug auf minimale Spannbäume von Bedeutung sind, abgeschlossen. Im nächsten Schritt wird expliziter auf die Arbeiten [2] und [1] eingegangen. Kapitel 3 stellt zunächst die eingeführten Algorithmen *Filter-Kruskal* und *Filter-Kruskal-Plus* aus [2] zur Optimierung von Kruskal vor. Aus [1] wird *IncrementalQuickSort* zur Verbesserung der Sortierung mit Kruskal und *Quickheap* als effiziente Heap-Datenstruktur für Prim eingeführt. Kapitel 4 besteht aus der Implementierung dieser Algorithmen und Datenstrukturen. Hierbei wird dem Leser auch ein Überblick über die Klassen vermittelt. In der experimentellen Analyse in Kapitel 5 werden die Laufzeitperfomance von Filter-Kruskal und Filter-Kruskal-Plus mit dem qKruskal aus [4] auf Graphen verglichen. IncrementalQuickSort, welches ein optimiertes Sortierverfahren für die Anwendung mit Kruskal ist, wird mit anderen effizienten Sortierverfahren für Kruskal verglichen. Außerdem wird ein Vergleich zwischen dem Prim mit Quickheap und dem in der Literatur bekannten Binaryheap gezogen. Zum Schluss wird im Fazit die gewonnenen Erkenntnisse dieser Arbeit zusammengefasst.

2 Vorbemerkung

2.1 Bisheriger Forschungsstand

Im Folgenden wird der Forschungsstand für jeden der wesentlichen Algorithmen *Filter-Kruskal*, *IncrementalQuicksort* und *Quickheap* einzeln behandelt.

Zur Optimierung von Kruskal findet sich reichlich Literatur. Da bezüglich der Laufzeit die Sortierung der m Kanten die Schwäche des Algorithmus ist, liegt es nahe die Anzahl der zu sortierenden Kanten auf die Kanten, die zum MST beitragen, zu beschränken. So können beispielsweise die schweren Kanten, die mit größerer Wahrscheinlichkeit nicht zu dem MST beitragen, identifiziert werden, um diese bei der Sortierung vorerst nicht zu berücksichtigen. Kershenbaum und van Slyke erzielen dies mittels einer Priority Queue in linearer Laufzeit [3]. Brennan hat diese Idee in seiner Arbeit [4] mit dem Algorithmus *qKruskal* durch die Integration einer Partitionsmethode weiter fortgeführt. Hierbei wird Kruskal – ähnlich dem Verfahren *QuickSort* – rekursiv in leichte und schwere Kanten partitioniert. Zuerst wird Kruskal rekursiv auf die leichten Kanten angewendet, bevor die schweren betrachtet werden. *Filter-Kruskal* basiert auf *qKruskal* und enthält eine Veränderung, die zu einer effizienteren Laufzeit führen soll. Ob sich dieses Ergebnis anhand unserer Implementierung bestätigen lässt, wird in 5 auf randomisierten Graphen analysiert.

Prims Laufzeit hängt maßgeblich von der zugrundeliegenden Datenstruktur ab. Da die Knoten im Allgemeinen mit einem individuellen Schlüssel in einem Priority Heap verwaltet werden, erfolgen mehrere DecreaseKey- und extractMin-Operationen, die möglichst effizient implementiert werden müssen. Hierbei wurden in der Literatur insbesondere zwei Heapformen genutzt: BinaryHeap und FinbonacciHeap. Der von Navarro et al. in [1] entwickelte Quickheap soll im Vergleich zu diesen beiden eine bessere Laufzeit erreichen. Hierzu werden wir, wie auch in der Arbeit von Navarro et al., nur einen Vergleich zwischen Quickheap und BinaryHeap ziehen, da BinaryHeaps gemäß Moret und Shapiro [5] die angeblich besten in der Praxis arbeitenden Priority Queues für Prim sind. Insbesondere wird in der Arbeit darauf hingewiesen, dass Quickheap um einen Faktor von 4 schneller läuft als BinaryHeap. Ob sich bei unserer Implementierung, die gleichen Resultate ergeben, wird in Kapitel 5 besprochen.

2.2 Problemdefinition und fundamentale Algorithmen

Zunächst wollen wir das Konzept des minimalen Spannbaums formalisieren. Hierbei orientieren wir uns bei den Definitionen an [6].

Im Folgenden bezeichne $G = (V, E)$ stets einen ungerichteten Graphen

mit $|V| = n$ und $|E| = m$. G hat reelle Kantengewichte $w : E \to \mathbb{R}$. Die Funktion w wird im Folgenden auch dazu verwendet das Gewicht einer Teilmenge E' von E zu definieren, sodass gilt

$$w(E') = \sum_{e \in E'} w(e).$$

Wir definieren einen minimalen Spannbaum wie folgt:

Definition 1 *Eine Kantenmenge $T \subseteq E$ heißt Spannbaum von G, wenn der Graph (V, T) ein Baum (das heißt zusammenhängend und kreisfrei) ist. Ein Spannbaum $T \subseteq G$ heißt minimaler Spannbaum von G, wenn es keinen Spannbaum von G mit einem kleineren Gewicht als $w(T)$ gibt.*

Die hier betrachteten Algorithmen Kruskal und Prim zur Berechnung eines MST sind Greedy Algorithmen. An dieser Stelle wird sowohl auf einen vollständigen Pseudocode, als auch auf den Beweis der Korrektheit für diese zwei Algorithmen verzichtet. Es ist jedoch erwähnenswert, dass für dünn besetzte Graphen der Kruskal-Algorithmus besser abschneidet als Prim, jedoch unterliegt Kruskal bei dichten Graphen aufgrund der $\mathcal{O}(m \log m)$ Laufzeit zur Sortierung der Kanten.

3 Ergebnisse der Paper

In diesem Kapitel sollen die Algorithmen aus den Arbeiten [1] und [2] und die daraus resultierenden Theoreme vorgestellt werden.

3.1 Optimierung von Kruskal

Wir stellen zunächst den in [2] vorgestellten Algorithmus Filter-Kruskal und Filter-Kruskal-Plus vor. Die wesentliche Idee von Filter-Kruskal basiert auf der von qKruskal aus [4].
Zunächst wird getestet, ob die Kantenanzahl kleiner als die Knotenanzahl ist. Wenn dies der Fall ist, kann Kruskal direkt angewendet werden. Ist dies nicht der Fall, wird die Kantenmenge in leichte und schwere Kanten partitioniert. Dies wird rekursiv für die leichten Kanten fortgeführt, bis die Kantenanzahl kleiner ist als die Knotenanzahl. Liegt immer noch kein MST vor, wird dieser Vorgang rekursiv auf die schweren Kanten angewendet.

```
1:  FilterKruskal(E, T, P)
2:      if m ≤ kruskalThreshold(n, |E|, |T|)
3:          Kruskal(E, T, P)
4:      else
5:          Wähle ein Pivot p ∈ E
6:          E_≤ := {e ∈ E : w(e) ≤ p}
7:          E_> := {e ∈ E : w(e) > p}
8:          FilterKruskal(E_≤,T, P)
9:          E_> = Filter(E_>, P)
10:         FilterKruskal(E_>, T, P)
```

Der konkrete Unterschied zwischen Filter-Kruskal und qKruskal liegt darin, dass geprüft wird, ob die schweren Kanten zum MST beitragen können. Diese werden vorerst gelöscht und im nächsten Schritt sortiert man die Menge der schweren Kanten. Somit kann die Anzahl der zu sortierenden Kanten verringert werden. Die Filter-Funktion setzt dies um.

```
1:  Filter(E, P)
        return {{u, v} ∈ E : u, v sind in versch. Komponenten von P }
```

Man sollte sich bei Filter-Kruskal auch den Worst-Case vor Augen führen. Ein Möglichkeit für eine schlechte Laufzeit von Filter-Kruskal ist durch die Wahl des Pivotelements gegeben. Wie bei Quicksort kann auch die Laufzeit Filter-Kruskal durch eine schlechte Wahl des Pivotelements entarten und im jeden Schritt die schwerste Kante als Pivotelement wählen. Dies hat zur Folge, dass die Partition der Kantenliste in sehr unterschiedlich große Listen unterteilt wird. Dieser Worst-Case kann mittels gleichen Kantengewichten hervorgerufen werden.

Aus diesem Grund wird ein Graph mit zufälligen Kantengewichten, die paarweise unterschiedlich sind, gewählt. Die erwartete Laufzeit von Filter-Kruskal wird auf solchen Graphen mit folgendem Theorem zusammengefasst.

Theorem 2 (Filter-Kruskals erwartete Laufzeit) *([2]) Sei $G = (V, E)$ ein beliebiger Graph mit unterschiedlich und zufällig gewählten Kantengewichten. Dann ist die erwartete Laufzeit von Filter-Kruskal $\mathcal{O}(m+n \log n \log \frac{m}{n})$.*

Mittels der oben gezeigten Optimierung des Kruskal-Algorithmus nähert sich der Laufzeit von Prim. Prim erreicht eine Laufzeit von $\mathcal{O}(m + n \log n \log \frac{m}{n})$ nach der Arbeit von Noshita in [7].

3.2 IncrementalQuicksort

IncrementalQuicksort (IQS) ist ein online Sortierverfahren, welches der Reihe nach das nächstkleinste Element zurückgibt. Dies bedeutet, dass die ersten k Elemente in möglichst kurzer Zeit für jedes k bestimmt werden können.

Theorem 3 (IQS erwartete Laufzeit) *Sei eine Menge A mit m Zahlen gegeben und $k \leq m$. IQS braucht zum Bestimmen des k-kleinsten Elements in A eine erwartete Laufzeit von $\mathcal{O}(m + k \log k)$.*

Im Wesentlichen verwendet IQS *Quickselect* um das k-kleinste Element in dem Bereich A[k-1, m-1] zu finden. Die Hauptidee um die erwartete Laufzeit zu erreichen, besteht darin die von QuickSelect erzeugte, abfallende Sequenz von Pivotelementen in einem Stack zu speichern und für die darauffolgenden Aufrufe von IQS erneut zu verwenden. So muss immer nur der Bereich zwischen dem vordersten Element und dem nächsten Pivotelement betrachtet werden.

3.3 Optimierung von Prim

Die Optimierung von Prim erfolgt durch den Einsatz von Quickheaps, welche eine effiziente Datenstruktur für die Implementierung von Prioritätswarteschlangen darstellen sollen. Quickheaps basiert auf ein Array, welches mittels IQS partitioniert wird. Nach einem Aufruf von IQS werden die erzeugten Pivotpositionen in einem Stack gespeichert, wodurch eine Halbordnung der Elemente des Quickheaps zustande kommt. Die erwartete amortisierte Laufzeit pro Operation liegt in $\mathcal{O}(\log m)$. Dies folgt im Wesentlichen aus dem *Exponential-Decrease Property*, welches besagt, dass die Pivotpositionen, welche von IQS erzeugt werden, durchschnittlich exponentiell kleiner werden.

Theorem 4 (Exponential-Decrease Property) *Für jedes Segment heap[idx,S[pidx]-1] gilt: $\mathbb{P}$(pivot ist groß) $\leq \frac{1}{2}$*

Bei dem Extrahieren von Elementen muss lediglich der Bereich von dem vordersten Element bis zu dem kleinsten Pivotelement betrachtet werden. Bei *decreaseKey* und *insert* muss der Schlüssel im schlechtesten Fall mit allen Pivotelementen verglichen werden und jeweils konstant viele Vertauschungsoperationen durchgeführt werden. Diese Operationen liegen in $\mathcal{O}(\log m)$, da

im Erwartungswert nur logarithmisch viele Pivotelemente existieren. Zusammengefasst erhält man

Theorem 5 (Quickheaps Komplexität) *Die erwarteten amortisierten Kosten von jeder Sequenz von m Operationen insert, delete, findMind, extractMin und increaseKey in einem ursprünglich leeren Quickheap liegen in $\mathcal{O}(\log m)$ pro Operation, angenommen, dass Einfügungen und Löschungen an uniform zufälligen Positionen geschehen. Insbesondere gilt sogar, dass die erwarteten Kosten von den Operationen insert und delete in $\mathcal{O}(1)$ liegen. Alle Kosten der Operationen müssen mit den Kosten für das Updaten der Elementposition im Wörterbuch multipliziert werden für den Fall, dass delete oder increaseKey unterstützt werden.*

4 Implementierungsübersicht

Die implementierten Klassen kann man in die fünf Kategorien User-Interface-Klassen, Datenstrukturen, Test-Klassen, Sortierungsklassen sowie Kruskal- bzw. Prim-Klassen einteilen.

4.1 User-Interface-Klassen

Zur User-Interface-Klassen gehören

- *Main*
- *InputParser*: Implementiert die Interpretation der Eingabeparameter, welche den Verlauf des Programms bestimmen.

Die Klasse *InputParser* interpretiert die Eingabeparamter und legt den weiteren Verlauf des Programmes fest. Dabei müssen in jedem Fall der Parameter $-a$ gesetzt sein, gefolgt von einem oder mehreren Kürzeln, die den genutzten Algorithmus festlegen. Diese und alle weiteren Parameter können der Dokumentation in der *Main*-Klasse entnommen werden.

Hier anzumerken ist die Beobachtung, dass durch das angeben mehrerer oder aller Algorithmen ihre Reihenfolge der Bearbeitung intern zu unterschiedlichen Laufzeiten führt und nachfolgende ausgeführte eine schlechtere Laufzeit haben. Da wir die Ursache hier nicht gefunden haben, und Java mit dem *Garbage Collector* einen unvorhersagbaren Einfluss auf die Laufzeit haben kann, haben wir immer nur einen Algorithmus für jeden Test ausgewählt

und die Ergebnisse zusammengetragen. Dies bedeutet jedoch auch, dass für die zufällig generierten Graphen die Algorithmen nur auf zufällige Graphen gleicher Größe, jedoch ungleicher Kanten und Kantengewichte getestet wurden. Um eine gute Vergleichsbasis für die verschiedenen Algorithmen zu erhalten werden 20 bzw. 50 Iteration pro Dichte und Algorithmus ausgeführt. Als Ergebnis wird jeweils der Mittelwert der Laufzeiten aller Iterationen errechnet, welche geplottet und verglichen werden können.

Die Implementierung indes erlaubt das angeben mehrerer Algorithmen, jedoch ist aus oben genannten Gründen davon abzuraten.

4.2 Generelle Datenstrukturen

Folgende Datenstrukturen dienen der Verwaltung der Graphen und deren Komponenten:

- *Graph*: Zur Speicherung eines Graphen und der Übergabe an die verwendeten Algorithmen.
- *Edge*: Zur Speicherung von Kanten.
- *AdjacentList*: Zwischenspeicherung bei der Zufallsgenerierung eines Graphen.

Graph hat ein Array aus *Edge*-Objekten und ein Attribut *numOfNodes* zur Speicherung der Anzahl der Knoten eines Graphen. Zur Initialisierung eines Graphen müssen daher diese beiden genannten Attribute als Parameter dem Konstruktor übergeben werden. Man kann mittels der *getWeight*-Methode auf das Gewicht der Kantenlisten zugreifen. Das ist insbesondere hilfreich, weil der MST als *Graph*-Objekt von den Algorithmen zurückgegeben wird und man mittels der genannten Methode somit auf das Gewicht des minimalen Spannbaums zugreifen kann.

Mit der *toString*-Methode werden alle Kanten des Graphen als String wiedergegeben.
Die Klasse *Edge* dient der Speicherung einer Kante und mittels eines Comparator können Kanten gemäß ihres Gewichts miteinander verglichen werden. Die Endpunkte werden als Integer und das Gewicht der Kante *weight* als Fließkommazahl (double) gespeichert. Mit der *toString*-Methode kann die Kante ausgeben werden. Die Klasse *AdjacentList* speichert den Graphen als Nachbarschaftsliste, so dass eine effizienter Aufbau eines Zufallsgraphen möglich ist.

Eine Instanz *AdjacentList* implementiert eine Matrix aus allen möglichen Kanten in Form von booleschen Werten, welche aus Platzgründen in einem *BitSet* abgespeichert werden (ein Byte enthält acht boolesche Werte). Dieser boolesche Wert speichert die Eigenschaft, ob eine Kante zum generierten Graphen gehört oder nicht. Die Gewichtung der Kanten wird beim Parsen in das *Graph*-Format vorgenommen.

4.3 Test Klassen

Die Test Klassen dienen dem Testen von zufallsgenerierten oder aus Textdateien importierten Graphen.
Aus dem package *testing*:

- *Tests*: Diese abstrakte Klasse verwaltet den Testvorgang.
  - *FileGraphTests*: Hier wird der Graph aus einer Datei importiert und auf dieser Instanz die Algorithmen getestet.
  - *RandomGraphsTests*: Diese abstrakte Klasse erweitert *Tests* um die Fähigkeit, Graphen bestimmter Größe durch Klassen des packages *graphGenerator* generieren zu lassen und diese anschließend zu testen.
    - *DensityGraphsTests*: Anzahl der Kanten werden anhand eines Dichte-Paranmeters gewählt.
    - *EdgeGraphsTests*: Anzahl der Kanten werden direkt mit den Parametern übergeben.
- *ValidateEdges*: Diese Hilfsklasse prüft, ob die Parameter *numberOfNodes* und *numberOfEdges* zulässig sind und das Generieren eines Graphen ermöglichen.

Die Testklasse *Tests* verwaltet den Testvorgang. Durch Polymorphie kann einerseits eine Datei geöffnet werden, die einen Graphen enthält, um auf dieser Instanz die gewählten Algorithmen anzuwenden (Instanz der Klasse *FileGraphTests*). Andererseits können auch zufällige Graphen generiert und getestet werden (Instanz der Klassen *DensityGraphsTests* oder *EdgeGraphsTests*). Allgemein dient *RandomGraphsTests* zum Testen von zufällig generierten Graphen und erweitert *Tests* entsprechend. Dabei sind Anzahl

an Wiederholungen für eine bestimmte Größe und Dichte des Graphen sowie die maximale und minimale Knotenanzahl durch die Eingabeparameter definiert.

DensityGraphsTests erweitert *RandomGraphsTests* hauptsächlich um die Funktion *testForGivenNodeNumber*, welche anhand der Parameter für die Graphendichte aus der Knotenzahl die konkrete Kantenzahl errechnet, für die ein Graph generiert werden soll. Auch *EdgeGraphsTests* implementiert *testForGivenNodeNumber*, wobei anders als in der *DensityGraphsTests*-Klasse die Kantenzahl direkt durch die Parameter definiert ist.

Die Ergebniswerte sind die für alle Wiederholungen einer bestimmten Graphengröße gemittelten Zeitwerte der einzelnen Algorithmen. Diese Ergebnisse werden in einer Ascii-Tabelle gespeichert, in der zuerst die Knotenanzahl und im Anschluss für alle genutzten Dichten in jeweils einer Zeile die Zeitwerte eingetragen sind, wobei die einzelnen Algorithmen durch die Spalten repräsentiert werden. Diese Repräsentation der Ergebnisse ist mittels Gnuplot auslesbar und somit die Zeitmessungen auch graphisch darstellbar.

Aus dem package *graphGenerator*:

- *GraphGenerator*: Diese abstrakte Klasse generiert Graphen anhand der Parameter.
  - *LightGraphGenerator*: Generierung zufälliger Graphen mit einer Kantendichte unter 50%.
  - *DenseGraphsTests*: Generierung zufälliger Graphen mit einer Kantendichte über 50%.
- *WilsonAlgorithm*: Implementierung des Algorithmus von Wilson zur Generierung eines *Unified Spanning Tree* [8].

Das zufällige generieren der Testgraphen wird von der Klasse *Graphgenerator* übernommen. Dazu wird dem Konstruktor Anzahl der Knoten und Kanten übergeben. Das generierte Objekt kann nun durch den Aufruf der Funktion *generateGraph* einen zufälligen Graphen der gewählten Größe erstellen. Der Graph wird – abhängig von der Dichte – auf zwei verschiedene Weisen generiert. In beiden Fällen wird eine Instanz der Nachbarschaftsliste *AdjacentList* genutzt.

Liegt die Dichte unter 50%, so wird *generateGraph* einer Instanz der Klasse *LightGraphGenerator* aufgerufen.
Es ein zuerst leerer Graph im Form einer Nachbarschaftsliste (Instanz der *AdjacentList*) mit zufälligen Kanten aufgefüllt, wobei schon hinzugefügte Kanten verworfen werden. Da die Dichte des Graphen zu jeder Zeit der Generierung einen Wert unter 50% hat, liegt die Wahrscheinlichkeit, dass eine Kante verworfen wird, ebenfalls unter 50%. Im Durchschnitt sollte demnach maximal jede zweite Kante verworfen werden, so dass die Laufzeit der Graphengenerierung nur linear zu der Zeit wächst, die das zufällige Ziehen aus einer vollständigen Kantenmenge ohne Zurücklegen haben müsste. Um zu garantieren, dass der Graph zusammenhängend ist, wird zuerst ein Baum erstellt, der alle Knoten umfasst. Um zufällig einen Baum aus der Menge aller aufspannenden Bäume zu wählen, wurde in der Klasse *WilsonAlgorithm* der Algorithmus von Wilson [8] implementiert, welcher einen solchen zufälligen Baum generieren kann.

Die restlichen Kanten für den Graphen werden nun zufällig gewählt und hinzugefügt, bis der Graph die erforderliche Anzahl an Kanten erreicht hat.

Eine Instanz von *DenseGraphGenerator* wird erstellt, wenn die Dichte über 50% liegt. Der resultierende Graph wird durch Aufruf der Funktion *generateGraph* auf folgende Art erstellt: Ein Objekt der *AdjacentList* wird vollständig gefüllt, enthält demnach $n * (n-1)/2$ Kanten ($n :=$ Anzahl der Knoten). Danach werden zufällig gewählte Kanten von diesem Graphen gelöscht, bis die erforderliche Kantenzahl erreicht ist. Die Wahrscheinlichkeit, dass eine Kante zu löschen versucht wird, die schon gelöscht wurde, liegt – analog zu obiger Argumentation zum leichten Graphen – unter 50%. Anschließend wird ein Test durchgeführt, ob der Graph zusammenhängend ist (Anm.: Darauf könnte bei Graphen mit großer Knotenzahl eventuell verzichtet werden, da die Wahrscheinlichkeit eines zusammenhängenden Graphen in Relation zur Knotenzahl quadratisch steigt).

Anschließend wird in beiden obigen Fällen der Graph aus dem Objekt der *AdjacentList* in ein Objekt der *Graph*-Datenstruktur umgeformt und ausgegeben. Den Kanten werden dabei zufällige gleichverteilte Gewichte zwischen null und eins vergeben.

4.4 Sortierungsklassen

Da Kruskals Laufzeit sehr stark von der Laufzeit zur Sortierung der Kanten abhängt, ist es von Bedeutung sich über die gewählte Sortierungsstrategie Gedanken zu machen. In der experimentellen Analyse in Kapite 5 werden wir uns hierzu folgende Sortierungsmöglichkeiten miteinander vergleichen.

- *JavaSort* (JS) : Die aus der Java Library angeboten Sortierungsmethode für Arrays.
- *DualPivotQuickSort* (DPQS): Optimierte Version von QuickSort aus [9].
- *IncrementalQuickSort* (IQS): Implementierung folgte anhand der Beschreibung in [1].
- *Heap* (BH): BinaryHeap zur Sortierung von Elementen aus [10] mit kleinen Modifikationen, sodass man die Sortierung auf einem Teilbereich des Arrays ausführen kann.

Zur Benutzung der Sortierungsstrategien braucht man nicht die Klassen zu instanziieren. Hierzu wurden *static*-Methode implementiert, mit welchen die entsprechenden Sortierungen aufgerufen werden können. Bei IQS und BH muss man der *init*-Methode ein Teilarray von der Kantenliste eingeben, das sortiert werden soll. Das Teilarray wird durch zwei Parameter *left* und *right* spezifiziert. Um die Kanten zu sortieren muss für IncrementalQuickSort gemäß der Größe des Teilarray genauso oft die *getMin*-Methode und bei BinaryHeap die *deleteMin*-Methode aufgerufen werden. Bei DualPivotQuickSort muss lediglich zu Beginn das Teilarray der Methode *sort* übergeben werden. Dann wird automatisch dieses Teilarray komplett sortiert.

Der Grund für die Wahl einer solchen Implementierung ist die Tatsache, dass die Klassen im Algorithmus von Kruskal nicht regelmäßig neu instanziiert werden müssen. Das führt zu Speichervorteilen.

4.5 Kruskal-Klassen

Die Klassen, die für die Implementierung der Kruskal-Algorithmen von Bedeutung sind, bestehen aus folgenden Klassen:

- *Union-Find-Datenstruktur*: Bestehend aus UnionFind und UnionFind-Weighted zur Komponentenverwaltung bei Kruskal.
- *KruskalAlgo*: Bestehend aus den Hauptmethoden.
- *PivotStrategy*: Bestehend aus verschiedene Methoden zur Partitionierung von einem Array.
- *PartitionStrategy* Bestehend aus *Hoare* und *Lomuto* zur Partitionierung.

4.5.1 Union-Find-Datenstruktur

Bei der Union-Find-Datenstruktur haben wir uns bei der Implementierung zunächst an *UF* aus [6] orientiert. Sie ist in der Klasse *UnionFind* und hat folgende Felder:

- A: Für jedes i steht in $A[i]$ dessen Repräsentant.
- L: Für jedes i ist $L[i]$ eine Liste von Zahlen, von denen i der Repräsentant ist.
- *size*: Für jedes i steht $size[i]$ für die Anzahl der Zahlen, von welchen i der Repräsentant ist.

Durch das Feld A kann die *Find*-Methode in konstanter Zeit ausgeführt werden. Jedoch liegt die Laufzeit für die Union-Find-Operation bei $n-1$ Aufrufen bei $\mathcal{O}(n \log n)$.

Mit einer geschickteren Implementierung kann nach *UFW* aus [11] diese Zeit und damit auch den Algorithmus von Kruskal verbessert werden. Diese Union-Find-Implementierung ist auch bekannt als ein gewichtetes Union-Find, die mittels Pfadkompression arbeitet. Diese kann man in der Klasse *UnionFindWeighted* wiederfinden. Sie hat folgende Felder:

- *parent*: Für jedes i ist $p[i]$ der Vaterknoten.
- *rank*: Für jedes i ist $rank[i]$ der Rang vom Teilbaum von i.
- *count*: Anzahl der Komponenten.

Hier wird auf eine detailliertere Beschreibung über die Funktionalität sowie einen formellen Beweis der Laufzeit verzichtet. Auch wird die Datenstruktur nicht weiter im Detail erklärt. Dem interessierten Leser ist hierfür [11] oder auch die Vorlesungsfolien [12] zu empfehlen. Für unseren Fall ist es wichtig zu wissen, dass ihre Laufzeit im Worst-Case $\mathcal{O}(\log n)$ beträgt und somit in der Theorie vorteilhafter erscheint als die obige Implementierung. Dies zeigt sich auch in dem praktischen Vergleich der Laufzeiten in Kapitel 5.

4.5.2 KruskalAlgo

Die Klasse *KruskalAlgo* ist die Hauptklasse für alle Kruskal-Algorithmen. Sie bekommt in den Konstruktor ein *Graph*-Objekt übergeben und stellt folgende Attribute bereit:

- *mstSize*: Speichert die bisherige Anzahl von Kanten in *mst*.
- *edges*: Die Kanten des *Graph*, der dem Konstruktor übergeben wurde.
- *numberOfNodes*: Anzahl der Kanten des *Graph*, der dem Konstruktor übergeben wurde.
- *vertexDegree*: Sie gibt die Anzahl an adjazenten Kanten pro Knoten zurück.
- *mst*: Hier werden die Kanten vom fertigen MST gespeichert.

Wir haben *Graph* direkt in seine einzelnen Komponenten (Kanten und Anzahl der Knoten) gespeichert. So muss dies nicht später beim Ausführen der Algorithmen passieren, was zusätzliche Zeit kosten würde. Auch erschien eine solche Implementierung vorteilhafter, da *Graph* diese Attribute kapselt und die Algorithmen diese nicht mehr als Parameter erhalten müssen. Allerdings agieren alle Funktionen auf den gleichen Variablen. Die Kantenliste wird jedoch in-situ manipuliert, weshalb es nicht ausreicht die KruskalAlgo-Klasse einmalig zu instanziieren und alle Funktionen mit der gleichen Instanz der Klasse alle Funktionen aufzurufen.

Zu den Algorithmen in dieser Klasse gehören der normale Kruskal, qKruskal, Filter-Kruskal und Filter-Plus-Kruskal. Diese wurden jeweils iterativ, als auch rekursiv programmiert. Um diese Klasse besser zu beschreiben haben wir diese in drei Bereiche eingeteilt: *Iterative Funktionen*, *boundedKruskal-*

der normale Kruskal begrenzt auf ein Teilarray implementiert mit verschiedenen Sortiermethoden (DPQS, BH, JS, IQS) - und *rekursive Funktionen* und *Unterfunktionen*.
Die rekursiven Funktionen sind zur Vollständigkeit im Code enthalten, allerdings raten wir von der Verwendung dieser ab. Die iterativen und rekursiven Funktionen haben beide folgende Eigenschaften:

- Sie benötigen keine Parameter und agieren direkt auf den oben beschriebenen globalen Variablen *mstSize*, *edges* und *mst*.
- Sie geben einen *Graph* mit einer Kantenliste zurück, die die Kanten des *mst* enthält.
- Sie werfen eine *IllegalArgumentException*, wenn der Benutzer in den Konstruktor der Klasse einen nicht zusammenhängenden Graphen eingegeben hat.

Die rekursiven Funktionen.
Um die rekursiven Algorithmen zu starten, werden jeweils diese Funktionen ausgeführt:

```
public Graph qKruskalRecursive() {
   qKruskal(edges, 0, edges.length-1);
   ...
}
```

```
public Graph filterKruskalRecursive() {
   filterKruskal(edges, 0, edges.length-1);
   ...
}
```

```
public Graph filterKruskalPlusRecursive() {
   filterKruskalPlus(edges, 0, edges.length-1);
   ...
}
```

Jeder dieser Algorithmen ruft seine entsprechende rekursive Funktion auf, die zum großen Teil dem Pseudocode in [2] entsprechen. Man sollte beachten, dass für dichte Graphen diese Algorithmen darunter leiden, dass zu viele

Indizes zur Speicherung der Grenzen jeder Rekursion auf dem Stack liegen und es zu einem *Stackoverflow*-Error kommt. Somit war es das Ziel diese rekursiven Funktionen in iterative umzuschreiben, sodass dies behoben werden konnte.

Die iterativen Funktionen.
Der normale Kruskal wird mittels der Funktion

```
public Graph standardKruskal() {
   boundedKruskalXX(edges, 0, edges.length-1);
   ...
}
```

ausgeführt. Dieser ruft eine Unterfunktion *boundedKruskal* auf. Am Namensende muss noch entsprechend der Sortiermethode, die genutzt werden soll, dessen Abkürzung an dem Funktionsnamen beifügen. Zum Beispiel bei boundedKruskal mit DPQS als Sortiermethode lautet diese *boundedKruskalDPQS(edges,left, right)*. Dies Funktion wendet den normalen Kruskal auf ein Teilarray mit der Sortiermethode DPQS an. In den Funktionen kann dann diese Methode aufgerufen werden. Die iterative Methoden lauten wie folgt.

```
public Graph qKruskal() {...}
```

```
public Graph filterKruskal() {...}
```

```
public Graph filterKruskalPlus() {...}
```

Diese können nun auf weitaus größere und dichtere Graphen agieren ohne den Speicher im Stack zu überfüllen, weil der Stack der Rekursion manuell implementiert ist und dieser auf dem Heap gespeichert wird. *filterKruskal* und *filterKruskalPlus* sind identisch zu *qKruskal* bis auf den Unterschied, dass diese beiden an einer Stelle eine Unterfunktion *filter* und *filterPlus* aufrufen müssen. Wir werden daher zunächst die iterative Methode *qKruskal* erklären, da dies auch Grundlage für die beiden anderen Methoden ist. Danach wird auf die Unterfunktionen eingegangen.
Die Idee bei der iterativen Implementierung ist die, dass die Grenzen für die Partition in der Rekursion immer im Stack gespeichert und diese dann immer jeweils abgerufen werden. Seien dazu s_1 und s_2 immer die ersten zwei

Elemente, die *aktuell* oben auf dem Stack liegen. $E(x, y)$ bezeichnet die Kanten in E von Index x bis y, wobei beide Indizes eingeschlossen sind. Mit S bezeichnen wir den Stack. m ist die Anzahl der Kanten.

```
1:  qKruskal()
2:      Lege m − 1 und 0 in dieser Reihenfolge auf S.
3:      while (Anzahl der Kanten im MST < n − 1)
4:          if (s2 - s1 + 1 ≤ n)
5:              Entnehme s1 und s2 aus S.
6:          else
7:              Wähle ein Pivot p_old ∈ E(s1, s2)
8:              p_new := Partition(E(s1, s2))
9:              Entnehme s1 aus S und setze first := s1.
10:             Lege Pivot p_new und p_new − 1 in dieser Reihenfolge auf S.
11:             Lege first zurück auf S.
```

Die Wahrscheinlichkeit bei unterschiedlichen Kantengewichten *nicht* die Kante mit dem größten Gewicht zu wählen, kann jedoch mittels Randomisierung minimiert werden.

Die Unterfunktionen.

Die beiden Methoden *filterKruskal* und *filterKruskalPlus* nutzten die Unterfunktionen *filter* und *filterPlus*. Wie oben erwähnt unterscheiden sich *filterKruskal* und *filterKruskalPlus* zu *qKruskal* um die zusätzliche Unterfunktion, die in den beiden zwischen Zeile 5 und 6 im oben stehenden Pseudocode aufgerufen wird.

Wird filterKruskal aufgerufen, werden diesem die Kantenliste und die zwei aktuellen Grenzen der schweren Kantenmenge übergeben, auf welche die Kanten gefiltert werden soll. Filter-Kruskal durchläuft das Teilarray von der linken bis zur rechten Grenze und löscht diejenigen Kanten, die mit dem bestehenden MST einen Kreis ergeben. In unserer Implementierung werden diese Kanten auf **null** gesetzt und innerhalb diese Teilarrays ganz nach links verschoben. Folgender Pseudocode beschreibt *filterKruskal*, der ähnlich zu qKruskal ist und um die Zeilen 6 bis einschließlich Zeile 11 ergänzt wurde.

```
1:  filterKruskal()
2:      Lege n und 0 in dieser Reihenfolge auf S.
3:      while (Anzahl der Kanten im MST < n − 1)
```

```
4:      if (S ≠ ∅ und s_2 - s_1 + 1 ≤ n)
5:          Entnehme s_1 und s_2 aus S.
6:          Entnehme s_1 aus S.
7:          newBoundary := filter(E(s_1, s_2)).
8:          if (newBoundary > s_2)
9:              Entnehme s_2 aus S.
10:         else
11:             Lege newBoundary auf S.
12:     else
13:         Wähle ein Pivot p_old ∈ E(s_1, s_2)
14:         p_new := Partition(E(s_1, s_2))
15:         Entnehme s_1 aus S raus und setze first := s_1.
16:         Lege Pivot p_new und p_new − 1 in dieser Reihenfolge auf S.
17:         Lege first zurück auf S.
```

Nach dem Filtern von Kanten ist s_1 nicht mehr aktuell und wir setzen diese auf *newBoundary*. Von *newBoundary* bis s_2 gibt es dann Kanten, die mit dem bisherigen MST noch keinen Kreis ergeben. Ein Spezialfall tritt auf, wenn alle Kanten zwischen s_1 und s_2 gelöscht worden sind. In diesem Fall wird auch s_2 aus S entnommen.
Analog gehen wir auch für Filter-Kruskal-Plus vor und ersetzen lediglich filter mit filterPlus. Dabei funktioniert filterPlus ähnlich zu filter. Es wird getestet, ob ein Knoten der Kante genau einen adjazenten Knoten hat. Wenn dies der Fall ist, kann die Kante direkt in den MST eingefügt werden.

Wie oben erwähnt kann sich die Wahl des Pivotelements auf die Laufzeit auswirken. Bei einer ungünstigen Wahl kann dies zu einem Worst-Case-Szenario führen, wenn beispielsweise das Pivotelement die Kante mit dem größten Gewicht ist.

4.5.3 PivotStrategy

Zur besseren Laufzeitoptimerung wurden mehrere Pivotstrategien implementiert. Hierzu gehören

- *medianOf3:*
  Dieser nimmt den Median aus *left*, *right* und das mittlere Element zwischen diesen beiden Grenzen.

- *medianOfRandom3:*
  Dieser nimmt den Median aus drei zufällig gewählten Indizes in den Grenzen von *left* und *right*.

- *randomPivot:* Dieser nimmt ein zufällig gewähltes Element aus dem Bereicht *left* und *right*.

Wir werden diese drei verschiedenen Methoden bei der experimentellen Analyse vergleichen.

4.5.4 PartitionStrategy

In dieser Klasse werden zwei wesentliche Partitonsstrategien implementiert: Die *Hoare-Partition* und *Lomuto-Partition*.

Theoretisch zeigen sich zwischen diesen beiden Partitionsmethoden Unterschiede bezüglich der Anzahl der Vergleiche. Sei hierzu p eine Gleichverteilung auf der Menge $\{1, \ldots, n\}$. Sei A ein Array und in dieser wird gemäß p mit n verschiedenen Zahlen gefüllt. Dann kann man zeigen, dass für die erwartete Anzahl an swaps für Hoare $\frac{n}{6} - \frac{1}{3}$ gilt. Lomutos-Methode hat jedoch $\frac{n}{2} - \frac{1}{2}$ Aufrufe von swap. Die Analyse stammt aus [13] und wird dort ausführlicher thematisiert. Das heißt, dass die Methode von Lomutos drei Mal so viele Aufrufe wie die von Hoare hat. Inwiefern sich dieser Faktor auf die Laufzeit auswirkt, werden wir im Kapitel der experimentellen Analyse nochmal eingehen und auf unsere Graphengenerierung beide Partitionsmethoden miteinander vergleichen.

Für jeder dieser Partitionsmethoden gibt zwei verschiedene Methoden. Dieser unterscheiden sich durch die Argumente der Funktion. Schauen wir uns dies genauer für die beiden Lomuto-Methoden an. Für die Hoare-Methoden folgt es analog. Die zwei Methoden sind mit

```
public static int partitionLomuto(Edge[] a, int left, int right,
    int pivot){...}
```

und

```
public static int partitionLomuto(Edge[] a, int left, int right,
    double pivotValue) {
```

gegeben. Man erkennt, dass das letzte Funktionsargument verschieden ist. Beim Ersteren partitionieren wir bezüglich eine des Gewichts der Kante an der Position *pivot* in der Kantenlisten. Bei der zweiten Methode erhalten wir schon das Gewicht und partitionieren wir gemäß des Gewichts. Man sollte beachten, dass dieser Gewicht von einem der Kanten stammt. Allerdings ist nicht erkennbar von welcher. Beide Methoden geben den neuen Index des Elements im partitionierten Array zurück.

4.6 Prim-Klassen

Bei der Optimierung von Prim sind im Wesentlichen folgende Klassen von Bedeutung:

- *Quickheap*: Quickheap-Datenstruktur mit den unterstützten Operationen *add*, *delete*, *extractMin*, *decreaseKey*
- *EdgeWeightedGraph*: Graph als Adjazenzliste mit gewichteten Kanten
- *WeightedVertex*: Datenstruktur zum Speichern der Entfernungen der Knoten zum minimalen Spannbaum
- *PrimQH*: Algorithmus von Prim mit Quickheaps
- *PrimBH*: Algorithmus von Prim mit Binärbäumen
- *BinaryHeap*: Binärbaum-Datenstruktur

4.6.1 Quickheaps

Bei der Implementierung von Quickheaps wurde ein zyklisches Array von generischen Schlüsselelementen verwendet. Als Hilfsstrukturen dient ein Stack zur Speicherung der Positionen der von IncrementalQuicksort generierten Pivotelemente. Darüber hinaus wird mit der Variable idx auf das erste Element des Quickheaps verwiesen. Dies ist notwendig, da bei zyklischen Arrays die zu betrachtende Position mudolo der Kapazität des Arrays gerechnet wird und so früher belegte Stellen überschrieben werden können. Dies hat den Vorteil, dass beliebig lange Sequenzen von Einfüge- und Löschoperationen ausgeführt werden können, solange die Kapazität des Arrays nicht überschritten wird. Eine aufwendige Vergrößerung des Arrays ist dabei nicht notwendig. Des Weiteren wird ein zweites Array, zum Speichern der Schlüsselpositionen im

Quickheap verwendet. Dies erlaubt es bei den Operationen *delete* und *decreaseKey* die Positionen der zu betrachtenden Schlüsselelemente in konstanter Zeit zu ermitteln. Ursprünglich hatten wir versucht eine Hashmap aus dem Java.util Packet zu verwenden, allerdings traten hierbei bei langen Sequenzen von Operationen Fehler beim Zurückgeben von Schlüsselpositionen auf. Da der Quickheap vor allem für die Optimierung von Prim geschrieben wurde, können in diesem Fall nicht mehr Elemente als die Knotenanzahl des Graphen im Quickheap gespeichert werden, weshalb wir auf eine Hashfunktion verzichtet haben. Stattdessen muss jedes Schlüsselelement des Quickheaps unser Interface *hasID* implementieren, wodurch eine eindeudige ID-Nummer für jeden Schlüssel gewährleistet wird. Dieser stellt den Index im Array dar, unter welchem die Position des Schlüssels im Quickheap vermerkt wird. Im Fall von Prim wird als Index einfach der Index des Knotens im Graphen verwendet.
Quickheap verfügt im wesentlichen über die folgenden Funktionen:

- *Konstruktoren*: Quickheaps können über die Angabe der maximalen Anzahl der einzufügenden Elemente oder durch Übergabe eines Arrays mit den Schlüsseln erzeugt werden. Hierbei ist zu beachten, dass die maximale Anzahl der Elemente nicht überschritten werden kann.

- *add*: Fügt den übergebenen Schlüssel in den Quickheap ein.

- *contains*: Überprüft ob der übergebene Schlüssel im Quickheap vorhanden ist.

- *decreaseKey*: Ersetzt den alten Schlüssel durch den übergebenen Schlüssel und stellt die Heap-Eigenschaft wieder her. Es muss zusätzlich die Id des alten Schlüssels übergeben werden.

- *delete*: Löscht den übergeben Schlüssel aus dem Quickheap.

- *extractMin*: Gibt den kleinsten Schlüssel zurück und incrementiert idx, sodass der extrahierte Schlüssel überschrieben werden kann.

- *findMin*: Gibt den kleinsten Schlüssel zurück ohne diesen zum Überschreiben freizugeben.

4.6.2 Prim

Bei der Implementierung von Prim werden ausgehend von einem Startknoten in dem Quickheap bzw. dem Binärbaum die benachbarten Knoten mit ihren Distanzen zu dem bisherigen MST gespeichert und jeweils der Knoten mit der geringsten Entfernung zum MST hinzugefügt. Hierbei verwenden wir die Klasse *WeightedVertex*, welche einen Knotenindex und die Zugehörige Entfernung zum MST speichert.
Die beiden Prim-Varianten verfügen im Wesentlichen über folgende Funktionen:

- *Konstruktor*: Im Konstruktor wird ein Graph übergeben und Hilfsstrukturen initialisiert.
- *start*: Ruft die Berechnung des minimalen Spannbaums auf.
- *edges*: Gibt den minimalen Spannbaum als ArrayList von Kanten zurück.
- *weight*: Gibt das Gewicht des minimalen Spannbaums zurück.

5 Ergebnisse der experimentellen Analyse

In diesem Kapitel wird die experimentell getestete Performanz zwischen unseren Implementierungen mit den Aussagen in [2] und [1] verglichen.

Wir konnten beobachten, dass die beiden Implementierungen des Prim-Algorithmus eine weit schlechtere Performance als die Kruskal-Implementierungen haben. Die grundlegenden Unterschiede zwischen Prim und Kruskal betreffen die Datenstruktur und den Algorithmus selbst, wohingegen die einzelnen konkreten Realisierungen der Kruskal- und Prim-Derivate sich ausschließlich in der Sortierung (bei Kruskal) und in der Heap-Struktur (bei Prim) unterscheiden. Da die Performance von Prim und Kruskal so unterschiedlich ausfielen, macht der Vergleich zwischen den beiden Verfahren aufgrund dem unterschiedlichen Grad der Optimierung wenig Sinn. Aus diesem Grund werden wir in diesem Kapitel Kruskal und Prim einzeln betrachten und werden erklären, welche Maßnahmen zur Optimierung getroffen wurden.

Die Algorithmen werden auf den randomisierten Graphengenerierung aus 4.3 ausgeführt und miteinander verglichen. An einigen Stellen werde wir diese

auch auf explizite Instanzen aus der Vorlesung Algorithmischen Mathematik an der Universität Bonn ausführen, um den Leser auf bestimmte Eigenschaften hinzuweisen.

5.1 Kruskal

Unser Ziel wird es sein *filterKruskal* zu optimieren. Dieser verwendet nach einer bestimmten Anzahl an Partitionierungen den normalen Kruskal, somit ist es sinnvoll sich zunächst über die Optimierung von Kruskal Gedanken zu machen. Die Analyse wird sich daher in die zwei Teile, *Kruskal-Optimierung* und *Filter-Kruskal-Optimierung*, unterteilen. Diese haben wir in folgender Tabelle zusammengefasst.

Kruskal-Optimierung	
Union-Find-Datenstruktur	Ohne vs. mit Pfadkompression
Sortierung	JavaSort vs. BH vs. IQS vs. DualPivotQS
Filter-Kruskal-Optimierung	
Pivotwahl	Uniform zufällig vs. Median of Three
Partitionsmethode	Lomuto vs. Hoare

Tabelle 1: Übersicht über die verglichenen Implementierungen

Zu Letzt werden wir noch den normalen Kruskal, qKruskal, Filter-Kruskal und Filter-Kruskal-Plus miteinander vergleichen und analysieren, bis zu welchem Faktor man sich gegenüber dem normalen Kruskal verbessern konnte. Im nächsten Schritt werden die Ergebnisse den Laufzeiten mit denen in [2] vergleichen. Wir werden alle Vergleiche in diesem Unterkapitel auf einem Rechner mit 8GB Arbeitsspeicher, 64Bit Betriebssystem und Intel Core i7-3632QM Prozessor, 4x 2,2 GHz (Quadcore) ausführen.

5.1.1 Optimierung vom Standard-Kruskal

Den normalen Kruskal werden wir im Folgenden auch als Standard-Kruskal bezeichnen.

Union-Find-Datenstruktur

Es gibt verschiedene Union-Find-Datenstrukturen. Wir betrachten *UF* aus [6] und *UFW* aus [11] und ihre Performanz bei die Implementierung des Standard-Kruskals gegenüberstellen. Für den Standard-Kruskal wird ein Sortierverfahren benötigt. Um einheitliche Vergleichskontexte zu schaffen, verwenden wir für die Implementierungen des Standard-Kruskals zunächst eine einzige Sortiermethode, sodass sich die Performanzunterschiede auf die Union-Find-Datenstrukturen zurückführen lässt. Wir entscheiden uns für *DualPivotQuickSort* [9], der im Kapitel zur Implementierungsübersicht besprochen wurde. Die folgende Abbildung zeigt die Laufzeiten von dem Standard-Kruskal mit jeweils beiden Union-Find-Datenstrukturen auf der randomisierten Graphengenerierung für 10000 Knoten.

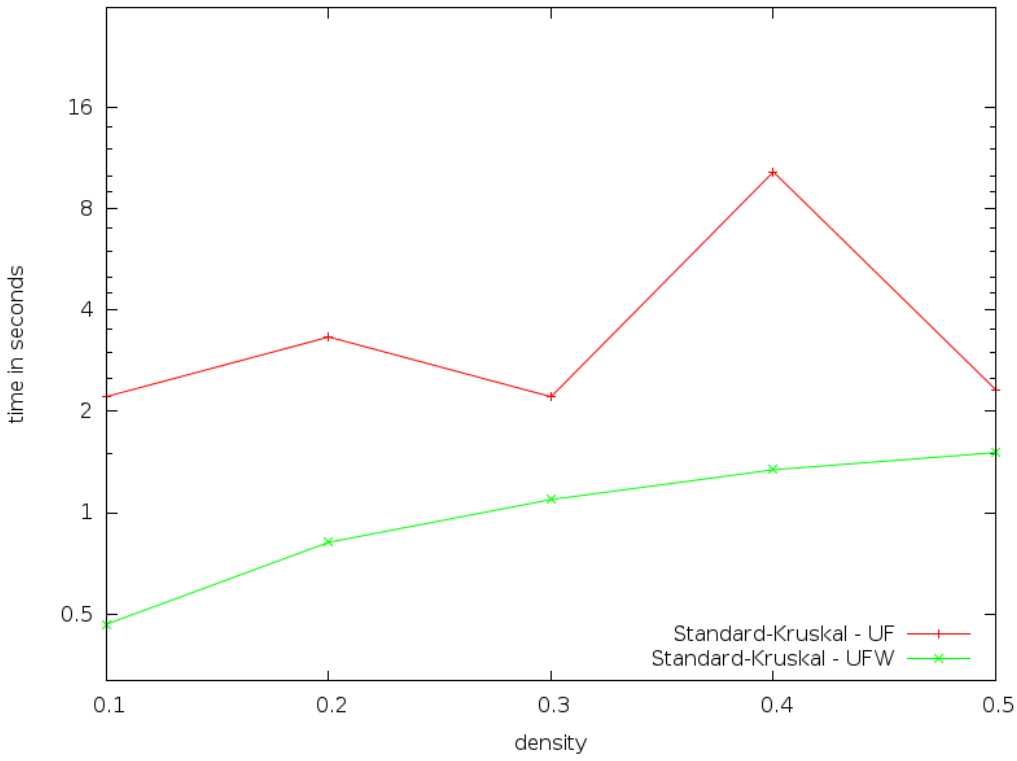

Abbildung 1: Der Standard-Kruskal mit *UF* und *UFW* für 15000 Knoten auf der randomisierten Graphengenerierung aus Kapitel 4.3. Die generierten Graphendichten reichen von 10% bis zu 50%.

Es ist anhand 5.1.1 zu erkennen, dass sich die Implementierung der Union-Find-Datenstruktur mittels Pfadkompression *UFW* aus [11] gegen *UF* aus [6] behaupten kann. Dies liegt daran, dass die Union-Methode in *UFW* wesentlich schneller ist als die in *UF*. Da der Aufruf der Union-Methode von der Anzahl der Knoten abhängt, lässt sich der Unterschied dieser beiden Datenstrukturen erst bei einer relativ großen Anzahl von Knoten erkennen. Es konnten zunächst keine signifikanten Unterschiede bei einer Knotenanzahl von 10000 erkannt werden und hatten uns daher für 15000 Knoten entschieden. Aufgrund begrenzter technischer Möglichkeiten konnten wir allerdings nur bis zu einer Dichte von 50% die Varianten testen.

Für den weiteren Verlauf werden wir für die restliche experimentelle Analyse *UFW* nutzen. Der einfachheitshalber werden wir daher mit dem Namen Standard-Kruskal stets eine Implementierung via *UFW* implizieren.

Sortierung

Zur Bestimmung eines minimalen Spannbaumes mit dem Standard-Kruskal benötigen wir ein Sortierverfahren. Bei dem Optimierung der Union-Find-Datenstruktur haben wir uns für den DualPivotQuickSort entschieden. Wir werden im Folgenden besprechen, ob wir es bei dieser Sortierung belassen sollten oder uns für einen weiteren entscheiden sollten.

Hierzu werden hierfür *JavaSort* (JS) (Sortiermethode für Arrays aus der Java Library), *DualPivotQuickSort*(DPQS) aus [9], *BinaryHeap* (BH) aus [10] und *IncrementalQuickSort* (IQS) aus [1] in Betracht ziehen. Für den weiteren Ablauf werden wir für den Standard-Kruskal die zu Grunde liegende Sortiermethode in Klammern durch die zugehörige Abkürzung benennen. Das heißt der Standard-Kruskal mit der Sortiermethode DualPivotQuickSort (DPQS) wird mit Standard-Kruskal(DPQS) benannt. In der folgenden Abbildung werden wir uns den Standard-Kruskal für die erwähnten Sortiermethode ausführen und uns für die beste entscheiden.

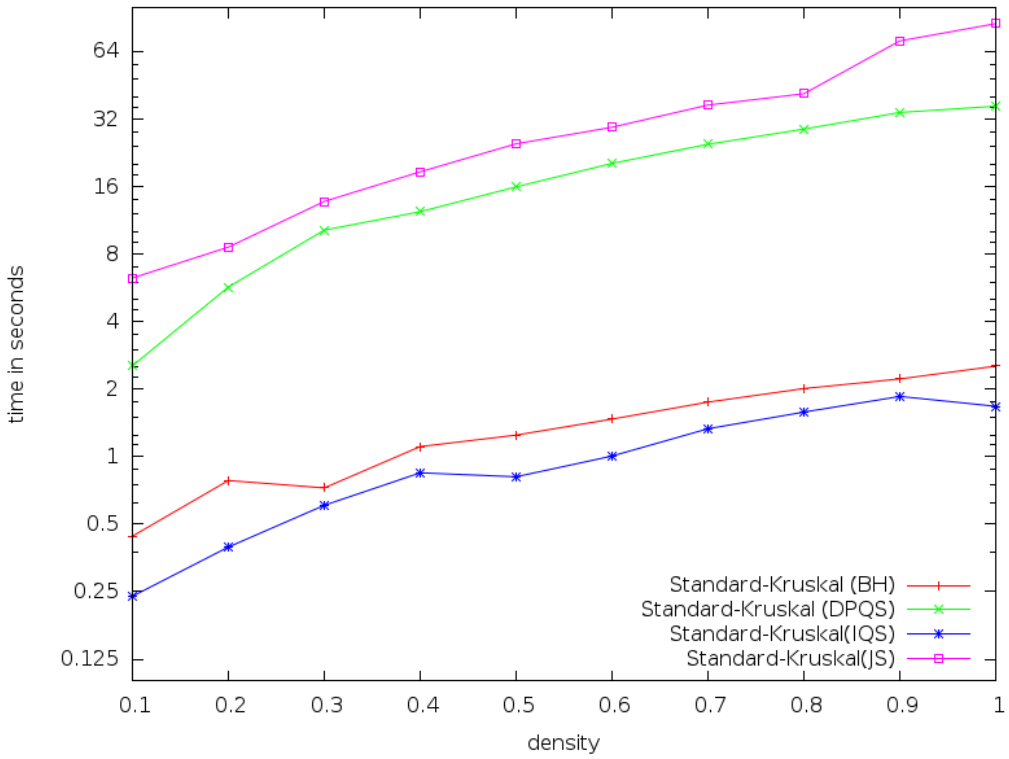

Abbildung 2: Standard-Kruskal mit jeweils JS, DPQS, BH und IQS im Vergleich für 10000 Knoten auf der randomisierten Graphengenerierung aus Kapitel 4.3. Die generierten Graphendichten reichen von 10% bis 100%.

JavaSort verwendet für Arrays *MergeSort*. Man erkennt, dass man mittels anderen Sortierverfahren, wie DPQS, BH oder IQS, eine bessere Performanz erreichen kann. Unter all den betrachteten Verfahren ist IQS für Kruskal überlegen. Dies ist im Einklang mit unserer Intuition, da beim Standard-Kruskal durch IQS online die nächst kleinere Kante bestimmt wird. Somit kann dieser stoppen sobald ein MST enstanden ist, ohne dass er das komplette Array sortieren musste im Gegensatz zu den restlichen Algorithmen. Anhand 2 erkennt an, dass der vorher benutzte DPQS um einen Faktor von fünf schlechter ist als BH und sogar um einen Faktor von 10 im Falle IQS.

5.1.2 Optimierung von Filter-Kruskal

Im Folgenden beschreiben wir die Optimierung von Filter-Kruskal. Da Filter-Kruskal auf qKruskal basiert, werden wir uns auf die Optimierung von qKruskal beschränken.

Sortierung
qKruskal verwendet als Unterfunktion den Standard-Kruskal, der auf einer Kantenmenge zwischen *left* und *right* aufgerufen wird. Hierzu ruft man die Funktion *boundedKruskal(edges, left, right)* auf. Der Name der Funktion hat noch eine Endung. Diese Endung gibt an mit welcher Sortiermethode die Kanten zwischen *left* und *right* sortiert werden müssen. Die Endung entspricht der Abkürzung des Sortierverfahrens, zum Beispiel für DualPivotQuickSort (DPQS) heißt die Methode *boundedKuskalDPQS(edges, left, right)*. Wie zuvor stellt sich die Frage mit welcher der vorgestellten Sortierverfahren qKruskal am besten perfomiert. Die erste Vermutung könnte sein, dass IQS auch an dieser Stelle sich durchsetzen müsste, da qKruskal nur den vorher beschrieben Standard-Kruskal auf ein Teilarray anwendet. Die folgende Abbildung, in der qKruskal mit den Sortierverhfahren DPQS, BH und IQS verglichen wird, widerspricht dieser Vermutung.

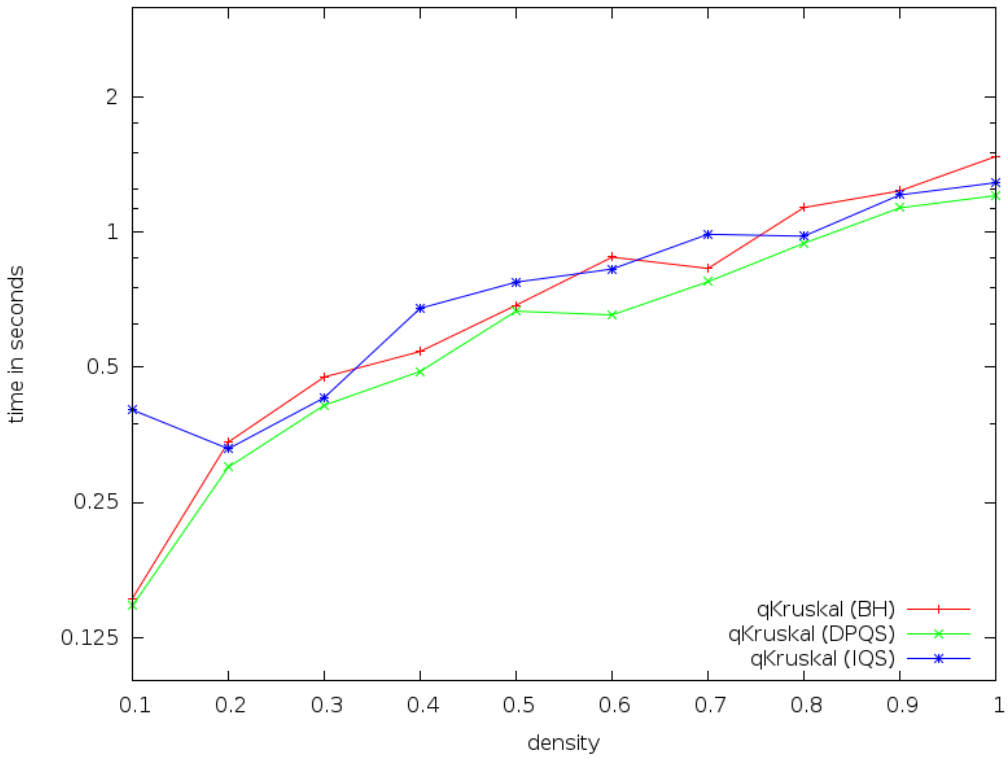

Abbildung 3: qKruskal mit jeweils DPQS, BH und IQS im Vergleich für 10000 Knoten auf der randomisierten Graphengenerierung aus Kapitel 4.3. Die generierten Graphendichten reichen von 10% bis 100%.

Abbildung 3 zeigt, dass man für DPQS (grün) und IQS (blau) keinen signifikanten Unterschied mehr erhält wie zuvor in Abbildung 2. qKruskal mit DPQS ist sogar besser als die qKruskal mit IQS.

Der Grund, dass IQS nicht mehr sonderlich gut abschneidet, kann anhand folgender Vermutung erklärt werden. IQS Vorteil beim Standard-Kruskal ergab sich dadurch, dass nicht alle Kanten sortiert werden mussten im Vergleich zu den anderen Sortierverfahren. Bei qKruskal müssen alle Sortierungsverfahren die Kanten nur in dem Bereich zwischen *left* und *right* sortieren. Somit begrenzt sich der Vorteil von IQS auf diese beiden genannten Grenzen *left* und *right*. Auch hier ist vorstellbar, dass IQS weniger Kanten sortiert als BH und DPQS. Allerdings kann es sein, dass der Mehraufwand von IQS die klein-

ste Kante zu bestimmen nicht mehr lohnenswert ist durch die Begrenzung auf das Teilarray. Da die größe dieses Teilarrays laut Definiton von qKruskal linear in der Größe der Knoten ist, entspricht diese Situation das Ausführen vom Standard-Kruskal auf einen Graphen, in welcher die Kanten linear in der Anzahl der Knoten ist.

Insbesondere gilt die obige Erklärung ebenfalls für die Filter-Kruskal bezüglich eines Sortierverfahrens. Zu einem basiert dieser au qKruskal und wir können die gleiche Argumentation nutzen. Zweitens ergibt sich durch das Filtern von Kanten das sich der Teilbereich, in dem sortiert werden muss, sich noch weiter verringert und dementsprechend auch IQS Vorteil gegenüber DPQS. Somit vermuten wir, dass ähnlich wie oben, dass der Mehraufwand, den IQS zur Bestimmung der nächst kleineren Kante benötigt, nicht mehr lohnenswert sein kann.

Diese Erkenntnis werden wir nutzen, um für den weiteren Verlauf nicht nur qKruskal, sondern auch filterKruskal und filterKruskalPlus mit DPQS zu implementieren.

Pivotisierung
Bei qKruskal muss ein Pivot zur Partitionierung des Arrays ausgewählt werden,um auf jeden dieser Teilarrays wieder qKruskal ausführen zu können. Die Pivotwahl entscheidet die Größe der Teilarrays. Der Best-Case wäre hier ein Pivot, der das Array in zwei gleichgroße Arrays teilt. Der Worst-Case wäre einer, der das Array sehr ungleichmäßig sortiert, sodass ein Teilarray aus einem Element besteht und der andere aus den vorherigen Elemente abzüglich eines Einzigen. Somit kommt der Pivotwahl eine große Bedeutung zu und werden deshalb im Folgenden drei verschiedene Pivotstrategien miteinander vergleichen.

Wir haben drei verschiedene Pivotisierungsverfahren, *RandomPivot*, *MedianOf3* und *MedianOfRandom3*, besprochen. Wir werden zunächst eine feste Partitionsmethode für qKruskal auswählen, um einen Laufzeitunterschied allein auf die Pivotwahl zurückführen können. Wir entscheiden uns hierbei für die HoarePartitionsmethode für alle Pivotstrategien. Die folgende Abbildung zeigt den Unterschied der drei Pivotmethoden auf der randomisierten Graphengenerierung für 10000 Knoten.

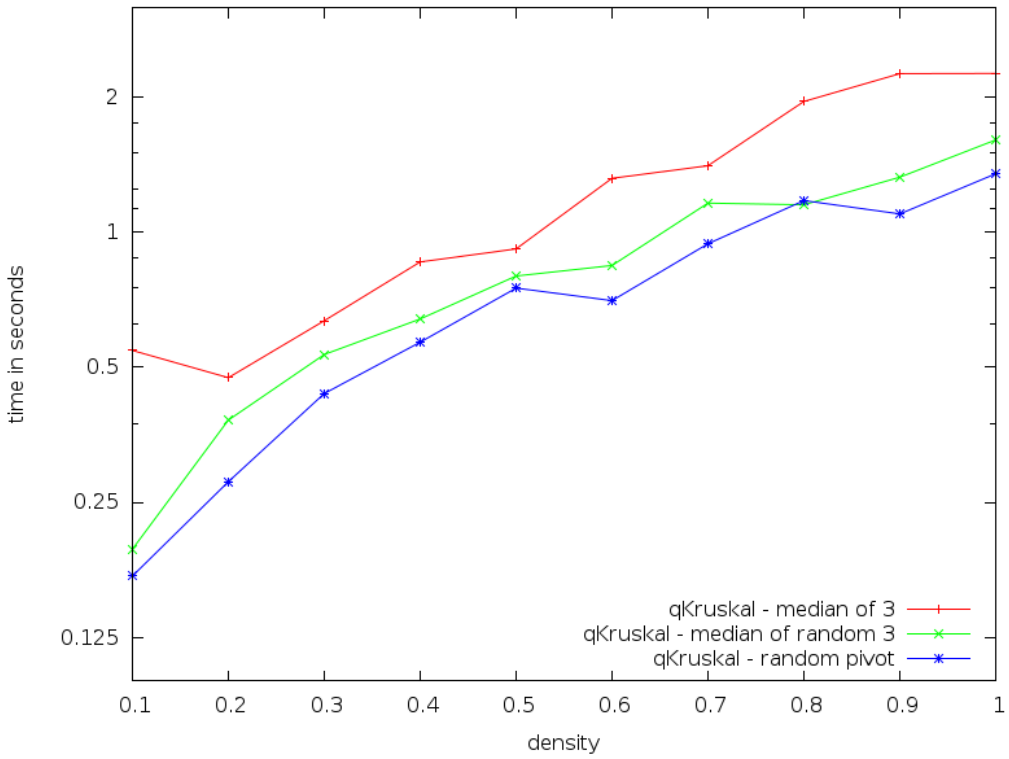

Abbildung 4: qKruskal, der als Unterfunktion den Standard-Kruskal mit *UFW* und IQS verwendet, wird jeweils mit der Pivotstrategie RandomPivot, MedianOf3 und MedianOfRandom3 für 10000 Knoten ausgeführt. Als Partitionsmethode wurde die HoarePartition ausgewählt. Verglichen wird auf der randomisierten Graphengenerierung aus Kapitel 4.3. Die generierten Graphendichten reichen von 10% bis 100%.

Anhand der Abbildung 4 lässt sich erkennen, dass MedianOf3 am schlechtesten abschneidet. Eine Begründung könnte sein, dass der Median aus *left*, *right* deterministisch ausgewählt wird. So kann das Worst-Case-Szenario mit höherer Wahrscheinlichkeit auftreten, als wenn die Wahl der drei Zahlen randomisiert erfolgt wie bei den MedianOfRandom3. Das Worst-Case-Szenario wäre gegeben, wenn als Pivotelement eine Kante mit größtem Gewicht ausgewählt wird, da als Folge Kruskal auf nicht gleichgroßen Mengen ausgeführt wird. Randomisiert man die Wahl des Pivotelementes, verbessert sich auch die Laufzeit von qKruskal. RandomPivot ergibt für qKruskal die beste Lauf-

zeit. Man hätte vermuten können, dass MedianOfRandom3 besser als RandomPivot ist, da sich die Wahrscheinlichkeit, den Worst-Case zu erhalten, verringert. Dies ist aber laut Abbildung nicht der Fall. Fortan verwenden wir zur Pivotisierung das Verfahren des RandomPivots.

Partitionierung

Wir werden uns im nächsten Schritt mit der Optimierung der Partitionsmethode in qKruskal beschäftigen. Bei dem Vergleich der Pivotisierung haben wir die Hoaremethode angenommen. Es gibt eine weitere Partitionsmethode, die unter dem Namen Lomuto bekannt ist. Wie im letzten Kapitel bereits erwähnt, ist die Anzahl der Vertauschungen der Kanten zur Partitionierung des Arrays bei Hoare um ein drittel geringer als bei Lomuto. Es stellt sich die Frage, ob dieses theoretische Ergebnis in der Praxis anhand der Laufzeit erkennbar ist. Die folgende Abbildung gibt uns einen ersten Eindruck über die Laufzeitunterschiede auf randomisierten Graphen für 10000 Knoten.

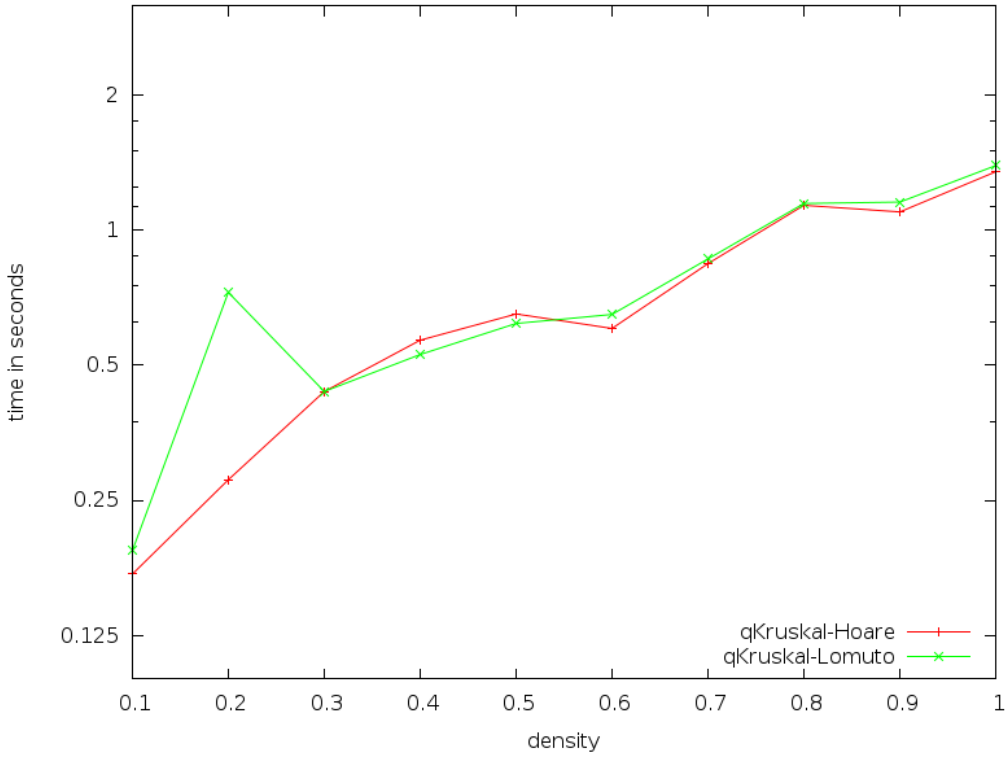

Abbildung 5: qKruskal, der als Unterfunktion den Standard-Kruskal mit *UFW* und IQS verwendet, wird jeweils mit der Partitionsmethode Hoare und Lomuto auf 10000 Knoten ausgeführt. Als Pivotisierungsverfahren verwenden wir RandomPivot. Verglichen wird auf der randomisierten Graphengenerierung aus Kapitel 4.3. Die generierten Graphendichten reichen von 10% bis 100%.

Auf den von uns generierten Graphen lässt sich in Abbildung 5 kein signifikanter Laufzeitunterschied zwischen Hoare und Lomuto erkennen. Betrachtet man hingegen die Laufzeit der beiden Partitionsmethoden auf Beispielinstanzen anhand 6, so stellen sich deutlichere Unterschiede ein. Die Ursache dafür könnte darin liegen, dass die Anzahl der swaps von der Anzahl der gleichen Kantengewichten abhängt. Bei der randomisierten Graphengenerierung ist die Wahrscheinlichkeit gleichen Kantengewichte zu erhalten sehr gering. Bei den Beispielinstanzen hingegen existieren eine große Anzahl von Kanten mit gleichen Gewichten und Hoare braucht diese Elemente nicht zu vertau-

schen im Gegensatz zu Lomuto. Der Laufzeitunterschied ist vor allem an Instanz 3 erkennbar.

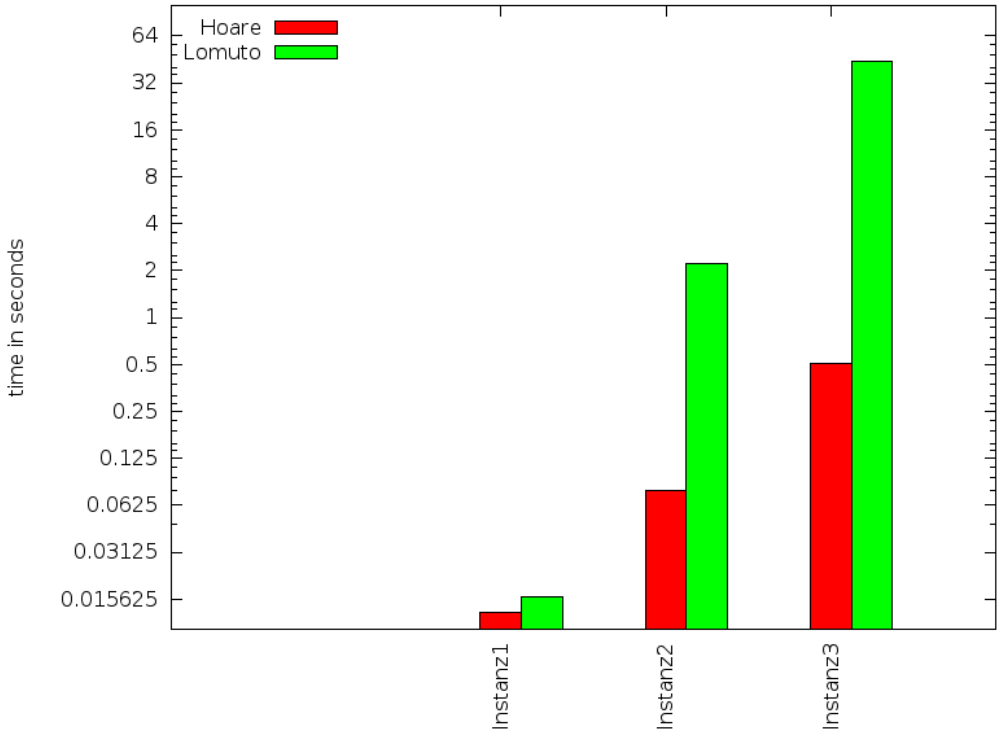

Abbildung 6: qKruskal, der als Unterfunktion den Standard-Kruskal mit *UFW* und IQS verwendet, wird jeweils mit der Partitionsmethode Hoare und Lomuto auf Beispielinstanzen ausgeführt. Als Pivotisierungsverfahren verwenden wir RandomPivot.

Um die Fälle mit vielen gleichen Kantengewichten abzufangen, werden wir deshalb Hoare bevorzugen.

Zusammenfassung
Aus den gewonnen Ergebnissen entscheiden wir uns nun qKruskal mit der DPQS als Sortierverfahren, dem randomPivot als Pivotstrategie und der Hoare-partition als Partitionsstrategie zu implementieren. Ähnliches gilt auf für FilterKruskal und FilterKruskalPlus, da diese außer der Unterfunktion

filter und *filterPlus* auf qKruskal basieren. folgenden Implementierungen zu verwenden.

5.1.3 Vergleich zwischen Kruskal-Varianten

Als nächstes werden wir die Performanz der Algorithmen Standard-Kruskal mit JS und IQS, qKruskal, Filter-Kruskal und Filter-Kruskal-Plus vergleichen. Hierzu werden wir zunächst auf Beispielinstanzen einen ersten Vergleich ziehen und im nächsten Schritt einen Vergleich auf der randomisierten Graphengenerierung.

Auf den Beispielinstanzen werden wir Standard-Kruskal(JS), Standard-Kruskal(DPQS), qKruskal(DPQS), filterKruskal(DPQS) und qKruskal(DPQS) miteinander vergleichen. Der Grund, dass wir den Standard-Kruskal(IQS) nicht hierauf testen liegt an den vielen gleichen Kantengewichten bei den Instanzen und dies bei IQS zu einer äußerst langen Laufzeit führt. Die folgenden zwei Abbildungen ist der Vergleich dieser Algorithmen auf den Beispielinstanzen.

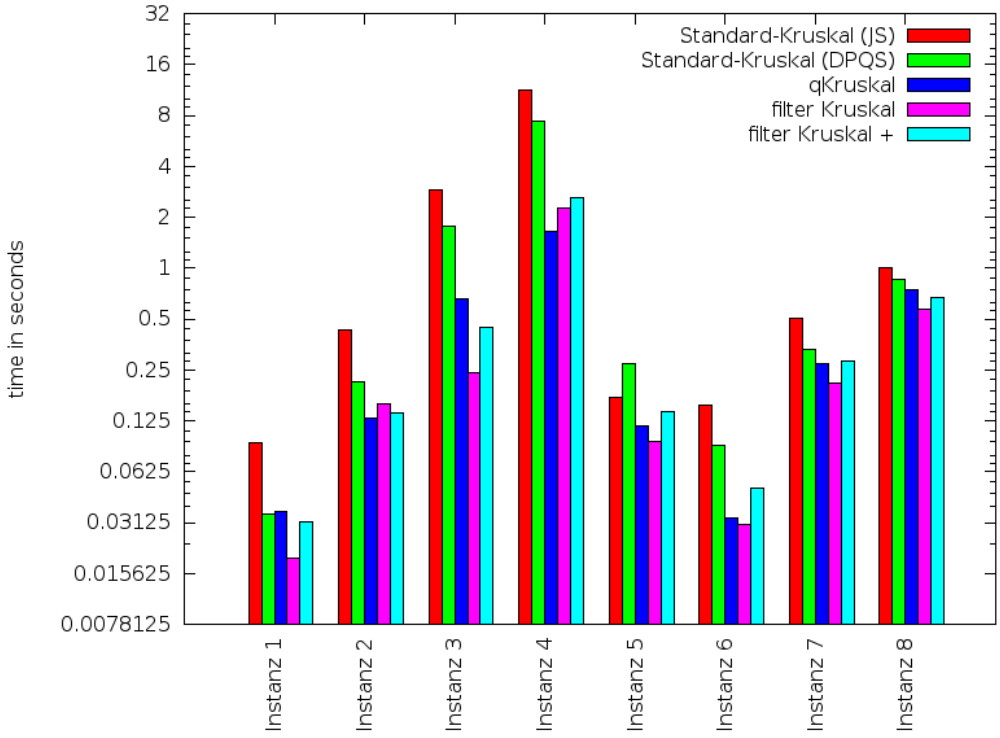

Abbildung 7: Vergleich zwischen dem Standard-Kruskal (vor der Optimierung mit JS), Standard-Kruskal (DPQS), qKruskal, FilterKruskal und FilterKruskalPlus auf Beispielinstanzen Instanz 1 bis 8.

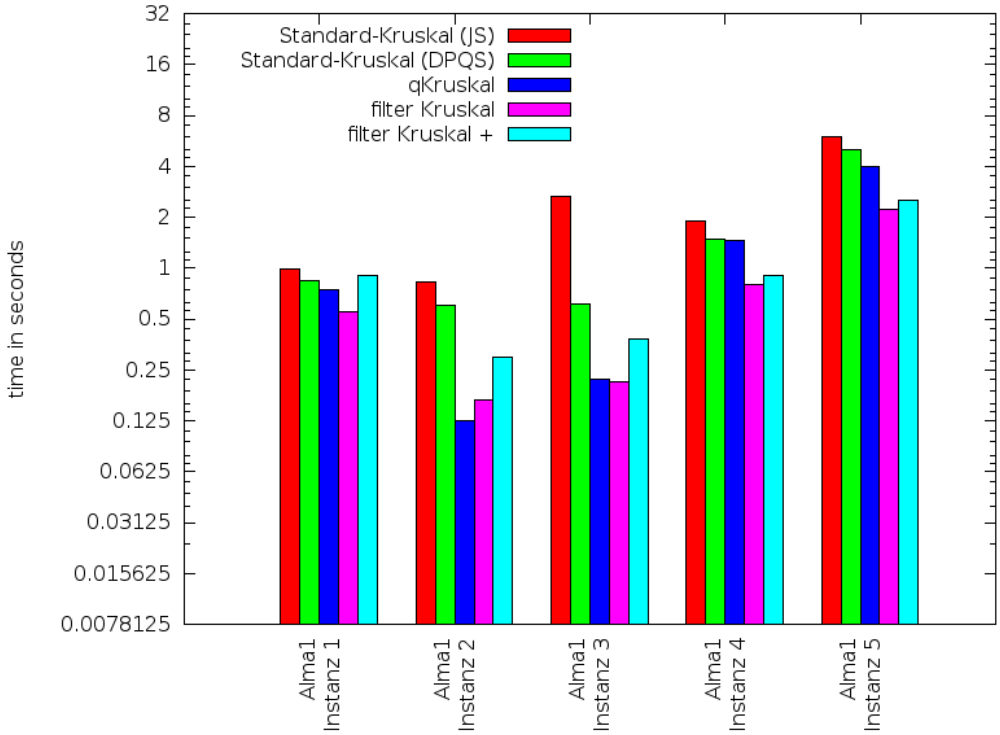

Abbildung 8: Vergleich zwischen dem Standard-Kruskal (vor der Optimierung mit JS), Standard-Kruskal (DPQS), qKruskal, FilterKruskal und FilterKruskalPlus auf Beispielinstanzen Alma1 Instanzen 1 bis 5.

In Abbildung 7 und 8 zeigen sich, dass Strandard-Kruskal (DPQS) sich auf den Beispielinstanzen gegenüber JS behaupten kann. Das Ergebnis konnten wir allerdings schon in 2 beobachten. Interessanter wird es beim Vergleich zwischen qKruskals Laufzeitperformanz gegenüber dem Standard-Kruskal(DPQS). qKruskal ist auf manchen Instanzen bis zu drei mal schneller als dieser. Filter-Kruskals Laufzeit ist auch besser als die von qKruskal und entspricht auch der Beobachtungen in [2]. Um welchen Faktor Filter-Kruskal besser als qKruskal ist, hängt natürlich auch von der Anzahl der Kanten ab, die im vornherein gefiltert werden können. Wie in der Arbeit [2] schon hingewiesen, kann mit FilterKruskalPlus keine bessere Laufzeit als die mit FilterKruskal erzielt werden. Die Laufzeit von FilterKruskalPlus liegt zwischen FilterKruskal und qKruskal und manchmal sogar über der von qKruskal.

Wir betrachten im nächsten Schritt den Vergleich der oben genannten Algorithmen auf unserer randomisierten Graphengenerierung für 10000 Knoten. Um den Grad der Verbesserung seit dem Standard-Kruskal (JS) zu verdeutlichen, werden wir alle besprochenen Kruskals in der folgenden Abbildung zusammenfassen.

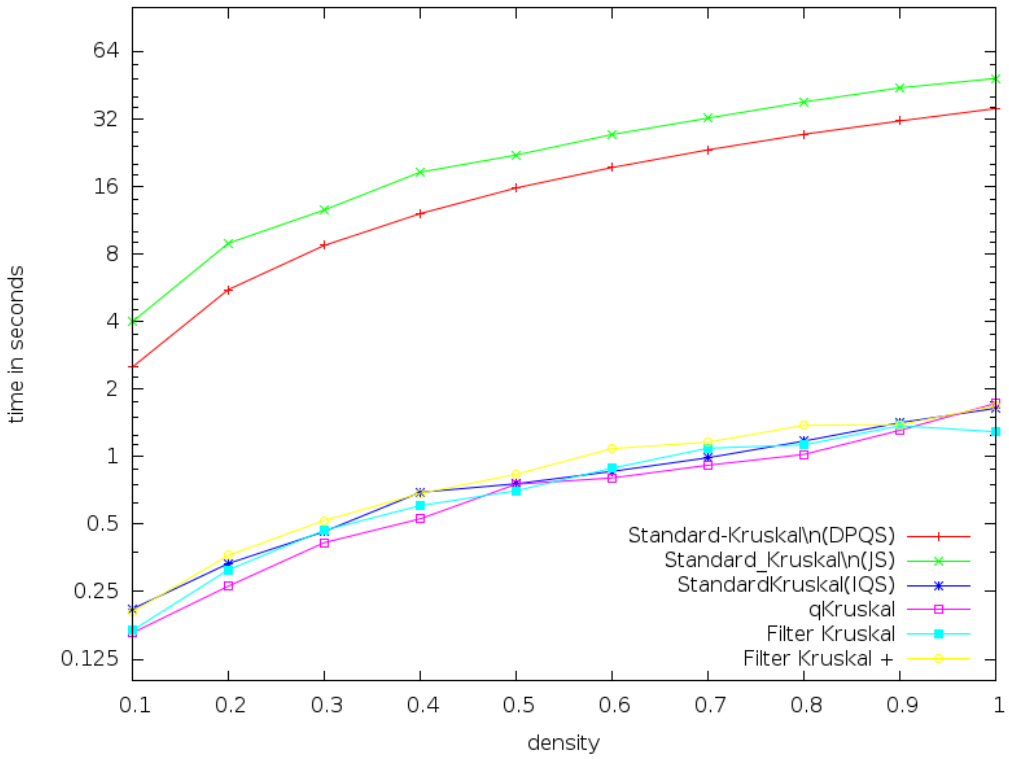

Abbildung 9: Vergleich zwischen dem Standard-Kruskal(JS), Stadard-Kruskal(IQS), Standard-Kruskal(DPQS),qKruskal, FilterKruskal und Filter-KruskalPlus auf unserer randomisierten Graphengenerierung aus Kapitel 4.3. Die generierten Graphendichten reichen von 10% bis 100%.

Anhand Abbildung 9 zeigt sich, dass der Grad der Verbesserung schon deutlich ist. Hierbei ist qKruskal der im Allgemeinen beste Algorithmus. Dies entspricht nicht mit der Beobachtung von [2] überein. Man hätte erwartet, dass sich FilterKruskal gegenüber qKruskal behaupten. Außer bei einer Dich-

te von 100% ist dies aber nicht der Fall. Da qkruskal und filterKruskal bis auf die *filter*-Methode in FilterKruskal ähnlich implementiert worden sind, liegt die Vermutung nahe, dass man eventuell durch eine bessere Optimierung dieser Methode eine bessere Laufzeit als qKruskal erzielen könnte. Man erkennt allerdings auch, dass FilterKruskalPlus stets schlechter ist als FilterKruskal ist. Dies entspricht auch den Beobachtungen in der Arbeit.

5.2 Vergleich zwischen Prim Varianten

In diesem Abschnitt kommen wir zu dem Vergleich von Prim mit Binärbäumen und Quickheaps. Wir verwenden Binärbäume, da diese zu den bekanntesten Verfahren zur Implentierung von Prioritätswarteschlangen gehören. Wie in der Abbildung zu sehen ist, liegt die Laufzeit von Prim mit Binärbäumen bei etwa einem Dreifachen bis Vierfachen von der Laufzeit mit Quickheaps. Insbesondere wird der Unterschied der beiden Verfahren bei dichteren Graphen größer. Dies könnte man auf die Speicherlokalität der Daten bei Quickheaps zurückführen, die sich bei höheren Kantendichten stärker auswirkt.

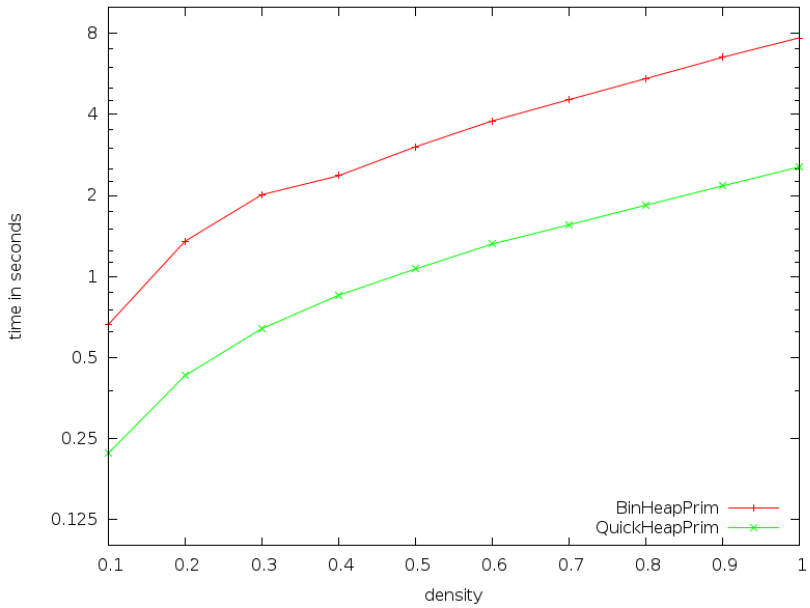

Abbildung 10: Prim mit Binärbaum und Quickheap bei Graphen mit 10000 Knoten

Instanz	Knoten	Kanten	Dichte
1	500	25316	20,293 %
2	5.000	1251736	10,015 %
3	20.000	9993303	4,997 %
4	15.000	37506352	33,341 %
5	10.000	1000279	20,008 %
6	1.000	149921	30,014 %
7	100.000	1.000.000	0,020 %
8	250.000	2.500.386	0,008 %

Tabelle 2: Die Dichte der Instanzen 1 bis 8 in Prozent.

Bei den Tests, die in der unteren Abbildung dargestellt sind, schneidet Prim mit Quickheaps vergleichsweise zu vorherigem Test etwas schlechter ab. Hierbei ist anzumerken, dass die Kantendichten der Testinstanzen zwischen 0.008% und 33,341% liegen.

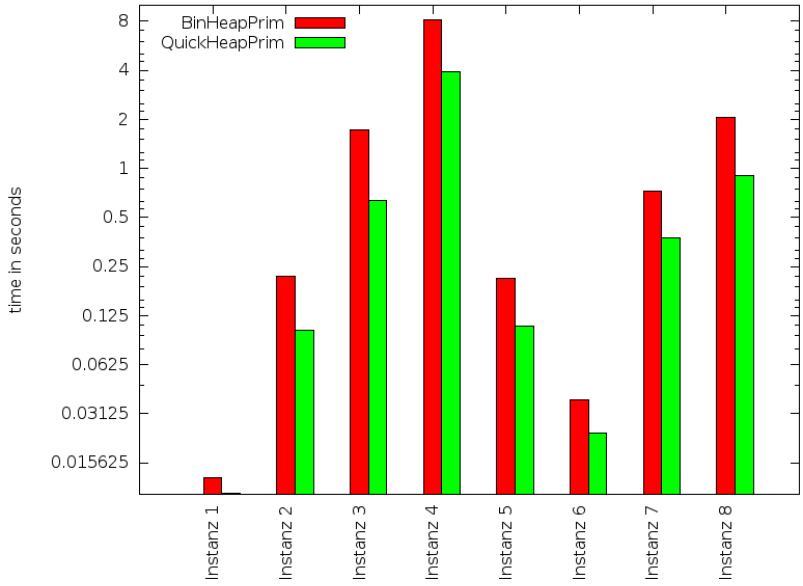

Abbildung 11: Prim mit Binärbaum und Quickheap bei Testinstanzen

Die Laufzeiten von Quickheaps sind in etwa im Einklang mit den Resultaten aus dem Paper, welche besagen, dass Quickheaps bis zu viermal schneller als Binärbäume laufen.

6 Fazit und Ausblick

Das Ziel dieser Projektgruppe war die Optimierung von Algorithmen zur Bestimmung von minimalen Spannbäumen. Hierzu galt es den Algorithmus von Kruskal mittels FilterKruskal aus [2] und IQS aus [1] zu verbessern. Im nächsten Schritt galt es auch Prim mittels Quickheap aus [1] zu verbessern. Wir haben unsere Implementierung dieser Algorithmen vorgestellt und dessen Performanz auf randomisierte Graphengenerierung getestet.

Zunächst haben wir uns auf die Optimierung von Kruskal beschäftigt. Hierbei zeigten FilterKruskal und der Standard-Kruskal mittels IQS eine deutliche bessere Laufzeit als der bisher bekannte Kruskal. Zwischen FilterKruskal und der Standard-Kruskal mit IQS konnte allerdings kein Gewinner ermittelt werden. FilterKruskal haben wir außerdem mit qKruskal und FilterKruskalPluse verglichen. FilterKruskal zeigte eine bessere Laufzeit als FilterKruskalPlus und entspricht auch der Beobachtung in [2]. Sie hat allerdings eine schlechtere Laufzeit als qKruskal und somit widerspricht dies den Ergebnissen aus [2].

Im zweiten Teil konzentrierten wir uns auf die Laufzeit von Prim mittels Quickheap. Wir haben hierzu einen Vergleich zu BinaryHeap gezogen. Quickheap zeigte sich hierbei bis zu vier Mal schneller als BinaryHeap und war im Einklang mit den Ergebnissen aus [1].

Im gesamten Vergleich konnte keiner der Prim-Varianten, weder mit Quickheap noch mit Binaryheap, sich gegenüber den Kruskal-Varianten behaupten. Hierbei kann über eine mögliche Verbesserung unserer Implementierung nachgedacht werden.

Literatur

[1] Gonzalo Navarro and Rodrigo Paredes. On sorting, heaps, and minimum spanning trees. *Algorithmica*, 57(4):585–620, 2010.

[2] Vitaly Osipov, Peter Sanders, and Johannes Singler. The filter-kruskal minimum spanning tree algorithm. In *10th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 52–61, 2009.

[3] A. Kershenbaum and R. Van Slyke. Computing minimum spanning trees efficiently. In *Proceedings of the ACM Annual Conference - Volume 1*, ACM '72, pages 518–527, New York, NY, USA, 1972. ACM.

[4] J.J. Brennan. Minimal spanning trees and partial sorting. *Operations Research Letters*, 1(3):113 – 116, 1982.

[5] Bernard M. E. Moret and Henry D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 400–411. Springer, 1991.

[6] Heiko Röglin. Algorithmen und berechnungskomplexität 1. `http://www.roeglin.org/teaching/WS2013/AlgoI/AlgoI.pdf`, Vorlesungsskript, Universität Bonn, Wintersemester 2012/2013. [Aufgerufen am 29.08.2015].

[7] Kohei Noshita. A theorem on the expected complexity of dijkstra's shortest path algorithm. *J. Algorithms*, 6(3):400–408, 1985.

[8] David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *PROCEEDINGS OF THE TWENTY-EIGHTH ANNUAL ACM SYMPOSIUM ON THE THEORY OF COMPUTING*, pages 296–303. ACM, 1996.

[9] Vladimir Yaroslavskiy. Dual-pivot quicksort. *Research Disclosure*, 2009.

[10] Jessica Miller. Binaryheap. `http://courses.cs.washington.edu/courses/cse373/11wi/homework/5/BinaryHeap.java`, 201. [Aufgerufen am 29.08.2015].

[11] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.

[12] Ernst W. Mayer. Effiziente Algorithmen und Datenstrukturen I. `http://www14.informatik.tu-muenchen.de/lehre/2009WS/ea/2009-12-17.pdf`, 2009. [Aufgerufen am 15.08.2015].

[13] Sebastian Wild. Java 7's dual pivot quicksort. Master's thesis, 2013.