Bachelorarbeit

Experimentelle Evaluation der Fragile Complexity randomisierter Selektionsalgorithmen

Author:
Julian Lorenz
Mtr.: 3383863

Supervisor:
Prof. Dr. Ulrich Meyer
Manuel Penschuck

2019 – 04 – 20

Inhaltsverzeichnis

Eidesstattliche Erklärung

Ich versichere an Eides statt durch meine eigene Unterschrift, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen genommen sind, als solche kenntlich gemacht habe. Die Versicherung bezieht sich auch auf in der Arbeit gelieferte Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Datum: Unterschrift:

1 *Einleitung*

Das Thema dieser Bachelorarbeit ist die Implementierung der Algorithmen RMinimum und RMedian aus den am 14. Juni 2018 und 9. Januar 2019 publizierten Versionen des Papers *Fragile Complexity of Comparison-Based Algorithms* [1], sowie eine Evaluation der experimentell ausgewerteten Daten.

Der Begriff der *Fragile Complexity* genannten Klassifizierung von vergleichsbasierten Sortieralgorithmen bezüglich der maximalen Teilnahme ihrer Elemente ist ein erst vor kurzem in das Interesse der wissenschaftlichen Forschung gerückt. Dementsprechend existiert ein konkreter Bedarf an empirischen Daten, die man gegen die theoretisch entwickelten Schranken abgleichen kann.

Zu Beginn dieser Arbeit wird der Begriff der *Fragile Complexity* sowie weitere benötigte Definitionen und Begrifflichkeiten eingeführt. Im Anschluss wird die Implementierung der Algorithmen sowie die Auswahl der untersuchenden Schranken besprochen. Entsprechend dieser Wahl wird eine sinnvolle Parametrisierung der Eingabe diskutiert.

Im Folgenden werden die verwendeten Analysemethoden vorgestellt. Das primär genutzte Werkzeug ist nicht-linearen Regression, die mithilfe des *Marquardt-Levenberg*-Algorithmus [5] realisiert wurde. Dieser nutzt die Methode der kleinsten Quadrate, um eine Kurve durch die gewonnene Datenwolke zu berechnen.

Abschließend werden die Resultate der Analyse in tabellarischer sowie grafischer Form präsentiert. Für den Algorithmus RMinimum konnten alle empirisch getesteten Schranken eindeutig bekräftigt werden. Für den Algorithmus RMedian hingegen konnten nicht genug Experimentaldaten gesammelt werden, um eine Aussage fest unterstützen zu können. Im Verlauf der Arbeit hat sich jedoch auch kein merklich Widerspruch zu den im Paper diskutierten Schranken ergeben, so dass es ein positiver Ansatz für weitere Forschung ist.

2 Fragile Complexity

Laufzeiteffiziente Algorithmen zum Auffinden eines bestimmten Elements innerhalb einer Menge sind für die Wissenschaft zu jedem Zeitpunkt von Interesse. Der theoretische Schwerpunkt dieser Arbeit liegt jedoch auf der Analyse der vorliegenden Algorithmen in Bezug auf ihre *Fragile Complexity* beziehungsweise die spezifischer Elemente.

Gerade bei Suchalgorithmen ist oft ein spezifisches Element von besonderem Interesse. Wird dieses Element im Laufe des Algorithmus oft aufgerufen, so kann man von einer dauerhaft erhöhten Belastung eines für den Nutzer wichtigen Elements ausgehen. Bei vergleichsbasierten Algorithmen entspricht ein Aufruf eines Elements dessen Teilnahme an einem direkten Vergleich mit einem anderen Element.

Als weltliches Beispiel sei hier ein fiktiver Staat genannt, der den Boxer des Landes für die Teilnahme an den olympischen Spielen bestimmen will. Verfährt der Staat hier nach einem klassischen K.O.-Runden Prinzip, so wird zwar der Beste ermittelt, dabei jedoch enorm strapaziert. Die *Fragile Complexity* entspricht in diesem Gleichnis der Belastung des Sportlers.

Die Nummerierung der hier folgenden Definitionen und Theoreme wurde an die des Papers [2] angelehnt und ist somit nicht durchlaufend.

Definition 1 (Fragile Complexity)**.** *Ein vergleichsbasierter Algorithmus $\mathcal{A}$ hat eine fragile complexity von $f(n)$, falls jedes Eingabeelement an maximal $f(n)$ Vergleichen teilnimmt. Insbesondere besitzt ein Element e bezüglich eines Algorithmus $\mathcal{A}$ eine fragile complexity von $f_e(n)$, falls e bei der Ausführung von $\mathcal{A}$ an maximal $f_e(n)$ Vergleichen teilnimmt.*

Nichtsdestotrotz soll auch weiterhin der Gesamtaufwand des Algorithmus selbst bewertet werden. Ein System-unabhängiges Maß hierfür ist die Anzahl der gesamt verrichteten Vergleiche.

Definition 2 (Arbeit)**.** *Ein Algorithmus $\mathcal{A}$ verrichtet Arbeit von $w(n)$, falls bei der Ausführung von $\mathcal{A}$ maximal $w(n)$ Vergleiche ausgeführt werden.*

Da wir in unserem Fall spezifische Elemente innerhalb einer Menge X anhand der Menge der Vergleichsoperatoren $\{<,>\}$ zwischen verschiedenen Elementen $x_i, x_j \in X$, $i \neq j$ bestimmen, muss die Menge der strikten Totalordnung genügen. Dies bedeutet, dass für zwei solcher Elemente stets gilt $x_i > x_j$ oder $x_i < x_j$ und somit insbesondere $x_i \neq x_j$, $\forall i \neq j$.
Um die Position eines Elements innerhalb der Menge unabhängig seines Wertes bezüglich der gegebenen Ordnung zu klassifizieren, legen wir den Begriff des Rangs fest.

Definition 3 (Rang)**.** *Bezüglich einer Menge X bezeichnet $rank_X(e)$ den Rang eines Element $e \in X$, welcher der Größe der Teilmenge von X entspricht, die alle Elemente enthält die nicht größer als e sind.*

Anders gesagt, entspricht der Rang eines Elements seiner Position innerhalb der Menge nach vollständiger Ordnung.

Wir rekapitulieren nun kurz ein paar Theoreme des Papers [1], auf die wir uns während unserer Analyse berufen werden. Da sich die Algorithmen in diesem Ansatz mit dem Auffinden des Minimums beziehungsweise des Medians beschäftigen, wollen wir zunächst eine obere Schranke für die *Fragile Complexity* des Algorithmus $f(n)$ selbst sowie die des Minimums $f_{min}(n)$ festlegen.

Theorem 1. *Die Fragile Complexity des Auffindens des Minimums von n Elementen ist maximal* $\lceil \log(n) \rceil$.

Beweis. Nutze einen perfekt ausbalancierten Turnierbaum. □

Theorem 2. *Für jeden deterministischen Algorithmus $\mathcal{A}$ zur Bestimmung des Minimums von n Elementen hat das Minimum-Element eine Fragile Complexity $f_{min}(n)$ von maximal* $\lceil \log(n) \rceil$.

Dies sind alle innerhalb des Rahmens dieser Arbeit relevanten Begrifflichkeiten und allgemeine Theoreme.

3 RMinimum

Der im Paper [1] vorgestellte Algorithmus RMinimum ist ein randomisierter rekursiver Algorithmus zum Auffinden des kleinsten Elements einer Menge X von n verschiedenen Elementen.
Er erhält als Eingabe das 2-Tupel (X,k), bestehend aus einer Menge $X := \{x_1, \cdots, x_n\}$ mit strenger Totalordnung und einem *Tuning Parameter* $k(n)$, der den *Trade-Off* zwischen der *Fragile Complexity* $f_{min}(n)$ des kleinsten Elementes und der maximal erwarteten *Fragile Complexity* $f_{rem}(n)$ der übrigen Elemente regelt. Für $\langle \mathbb{E}[f_{min}(n)], \max \mathbb{E}[f_{rem}(n)] \rangle$ ergeben sich je nach Wahl des Parameters $k(n)$ spezifische Wertepaare.
Hierfür wird im Laufe dieses Kapitels gewisse Abschätzungen vorstellt und diese abschließend experimentell untersucht.

3.1 *Algorithmus*

RMinimum

Eingabe: Ein 2-Tupel der Menge X mit total Ordnung von n verschiedenen Elementen sowie einem *Tuning Parameter* $k(n)$.

Ausgabe: Das *kleinste Element* der Menge X.

1. Man bildet aus den n Elementen zufällig $n/2$ paarweise-disjunkte Paare und nutzt für jedes Paar einen Vergleich um X in zwei neue Mengen (gleicher Größe) W und L zu unterteilen, wobei W die *Gewinner* des vorangegangenen Vergleiches, also jeweils die kleineren Elemente der Paare enthält und L entsprechend die *Verlierer*.
   Bezeichne $|W| := w$ sowie $|L| := l$ wobei gilt $w = l = n/2$.
2. Nun wird L zuerst in l/k zufällige disjunkte Teilmengen $L_1, \cdots, L_{l/k}$ der Größe k partitioniert und anschließend in jeder Teilmenge L_i das *kleinste Element* m_i mit einem perfekt ausbalancierten Turnierbaum ermittelt.
   Anschließend werden alle m_i in einer Menge M zusammengefasst.
3. W wird nun in w/k zufällige disjunkte Teilmengen $W_1, \cdots, W_{w/k}$ der Größe k partitioniert. Dann werden in jeder Menge W_i alle Elemente herausgefiltert, die größer als das entsprechende Element m_i sind.
   Zuletzt vereinigt man alle Mengen W_i und erhält $W' = \bigcup_i \{x_w | x_w \in W_i \wedge x_w < m_i\}$.
4. Falls $|W'| < \log^2(n)$, wird ein perfekt ausbalancierter Turnierbaum genutzt, um das Minimum von W' zu finden auszugeben. Andernfalls wird RMinimum rekursiv mit Eingabe $(W', k(n))$ aufgerufen.

Die Ausgabe des Algorithmus ist das der Ordnung nach kleinste Element der Eingabemenge X.

3.2 *Analyse*

In diesem Abschnitt werden der Algorithmus RMinimum im Gesamten und seine einzelnen Phasen theoretisch analysiert.
Hierbei ist die Auswahl der diskutierten Bereiche und Theoreme auf die für diese Arbeit

relevanten Aussagen eingeschränkt. Für weitere Informationen sei hier auf das Paper [1] verwiesen.

Phase 1. In der ersten Phase nimmt jedes der n Elemente an genau einem Vergleich teil. Da je zwei Elemente an einem Vergleich teilnehmen, folgt insgesamt:

$f_e(n) = 1, \forall e \in X$ und somit
$f_{min}(n) = f_{rem}(n) = f(n) = 1$ sowie
$w(n) = n/2$ für die verrichtete Arbeit.

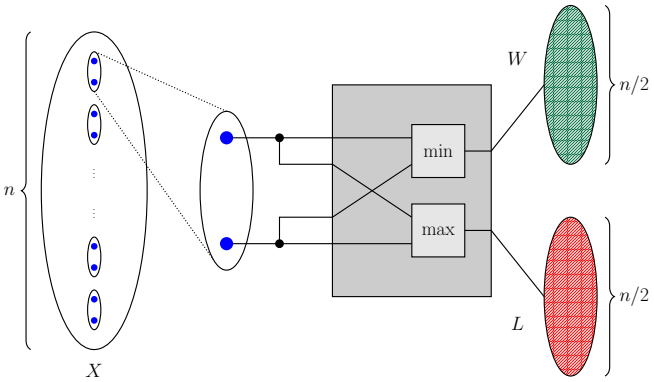

Abbildung 1: Visualisierung Phase 1

Phase 2. In der zweiten Phase gilt aufgrund des ausbalancierten Turnierbaums $1 \leq f_{x_l}(n) \leq \log_2(k)\ \forall x_l \in L$ und $f_{x_w}(n) = 0\ \forall x_w \in W$ sowie Insbesondere $f_{min}(n) = 0$ und $f_{rem}(n) = \log_2(k)$.

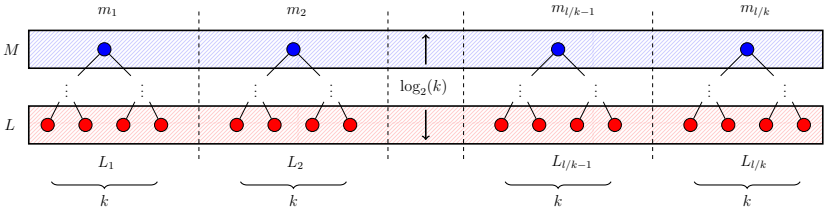

Abbildung 2: Visualisierung Phase 2

Um in einem perfekt ausbalancierten Turnierbaum bei k Teilnehmern einen Sieger zu ermitteln, bedarf es $k-1$ Vergleiche. Da es $n/2k$ Teilmengen gibt, ergibt sich $w(n) \leq (k-1)\cdot(n/2k) = n/2 - n/2k = \mathcal{O}(n)$.

Phase 3. In der dritten Phase wird jedes Element aus W genau ein mal mit einem Element m_i verglichen. Es gilt somit $f_{x_w}(n) = 1\ \forall x_w \in W$ sowie $f_{m_i}(n) \leq k\ \forall m_i$.
Für die in Phase 3 verrichtete Arbeit gilt $w(n) \leq n/2 + k \cdot n/2 = \mathcal{O}(n)$.

Phase 4. Die vierte Phase unterscheidet zwei Fälle. Wird die angegebene Schranke eingehalten, so gilt $1 \leq f_{x_{w'}}(n) \leq \log_2(|W'|)\ \forall x_{w'} \in W'$ sowie $f_e(n) = 0\ \forall e \notin W'$.
Insbesondere gilt $f_{min}(n) = \log_2(|W'|)$ sowie $f_{rem}(n) = \log_2(|W'|) - 1$.
In diesem Fall gilt für die verrichtete Arbeit $w(n) = |W'| - 1 = \mathcal{O}(n)$.
Im zweiten Fall findet ein rekursiver Aufruf statt, sodass keinerlei Vergleiche stattfinden.

RMinimum. Für den gesamten Algorithmus ist relevant festzustellen, dass die Wahrscheinlichkeit, die Schranke der vierten Phase einzuhalten, von der Randomisierung der vorherigen Phasen abhängig ist. Dementsprechend ist die Mächtigkeit der Menge $|W'|$ bei wiederholter Ausführung mit gleicher Parametrisierung immer noch zufällig, wobei die stochastische Verteilung abhängig von der Wahl der Parameter ist.

Folgende Aussagen wollen wir im Laufe dieser Arbeit experimentell unterstützen.

Theorem 3. RMinimum *benötigt lineare Arbeit, also* $w(n) = \mathcal{O}(n)$.

Beweis. Betrachte einen beliebigen Rekursionsschritt mit einer Eingabemenge der Mächtig-

keit n'. Die erste Phase benötigt $\mathcal{O}(n')$ viele Vergleiche um die Mengen W und L zu bestimmen. Des Weiteren beinhaltet die Menge W' kein Element der Menge L und es folgt $w(n) = w(|W'|) + \mathcal{O}(n) \leq w(n/2) + \mathcal{O}(n) = \mathcal{O}(n)$. □

Lemma 2. *Sei W_i eine beliebige Siegerpartition sowie m_i das zum Filtern genutzte Minimum und $W_i' = \{w | w \in W_i \wedge w_i < m_i\}$ die Menge aller Elemente, die kleiner als m_i sind.*
Dann gilt $\mathbb{E}[|W_i'|] \leq 2d\sqrt{k}$ für eine Konstante $d > 0$.

Theorem 4. *Sei $k(n) = n^{\varepsilon}$ für $0 < \varepsilon < 1/2$. Dann benötigt* RMinimum $\mathbb{E}[f_{min}] = \mathcal{O}(\varepsilon^{-1} \log\log(n))$ *Vergleiche für das kleinste Element und $\mathbb{E}[f_{rem}] = \mathbb{O}(n^{\varepsilon})$ für alle übrigen Elemente.*

Theorem 5. *Sei $k(n) = \log(n)/\log\log(n)$. Dann benötigt* RMinimum $\mathbb{E}[f_{min}] = \mathcal{O}(\log(n)/\log\log(n))$ *Vergleiche für das kleinste Element und $\mathbb{E}[f_{rem}] = \mathcal{O}(\log(n)/\log\log(n))$ für alle übrigen Elemente.*

Die aufgeführten Theoreme und Lemmata dienen als Grundlage für den Analyseschwerpunkt und somit auch für die in dieser Arbeit zugrunde liegende Implementierung.

4 RMedian

Der Algorithmus RMedian ist ein in Erwartung Praxis-optimaler Algorithmus zum Auffinden des Median Elements. Hierbei wird explizit auf die im Paper [2] vorgestellte Version Bezug genommen.

Mit Hilfe eines Parameters $k(n)$ kann man den *Trade-Off* zwischen der erwarteten *Fragile Complexity* $f_{med}(n)$ des Median Elements und der maximal erwarteten *Fragile Complexity* $f_{rem}(n)$ aller übrigen Elemente kontrollieren. Dies erlaubt je nach Adjustierung des Parameters verschiedene Werte für das Paar $\langle \mathbb{E}[f_{min}(n)], \max \mathbb{E}[f_{rem}(n)] \rangle$ zwischen $\langle \mathcal{O}(\log\log(n)), \mathcal{O}(\sqrt{n}) \rangle$ und $\langle \mathcal{O}(\log(n)/\log\log(n)), \mathcal{O}(\log(n)) \rangle$.

4.1 *Algorithmus*

RMedian erhält als Eingabe eine Menge X mit $|X| = n$, $n \in \mathbb{N}$ sowie zwei weitere Parameter $k(n), d(n) \in \mathbb{R}$ und liefert als Resultat den Median der Menge X.

Der Algorithmus kann in drei getrennte Phasen unterteilt werden.
Er beginnt mit der *Sampling Phase*, die aus der Ursprungsmenge X eine zufällige Teilmenge S nimmt, diese sortiert und anschließend in Segmente unterteilt, deren Größe durch die Parameter $k(n)$ und $d(n)$ bestimmt wird. Da S eine zufällige Teilmenge der Menge X ist, besitzt die Menge S in Erwartung den gleichen Median wie X. Dabei entstehen im Verlauf des Prozesses drei Teilmengen L, C und R. Am Ende enthält die Menge L Elemente kleiner, R Elemente größer als der Median der Menge S und C gerade die Elemente, die im Bereich des Medians der Menge S liegen und somit als Median-Kandidaten dienen.

Algorithm 1: RMedian : Sampling phase

```
1 procedure Sampling Phase(X = {x_1, ..., x_n}, k(n), d(n))
2     Randomly sample k elements from X and sort S with AKS
3     Distribute S into buckets L_b, L_{b-1}, ..., L_1, C, R_1, ..., R_{b-1}, R_b as follows:
4         set n_0 = 2*sqrt(k log(n)), n_1 = 3*sqrt(k log(n)), n_i = d * n_{i-1}
5         C = S[k/2 - n_0 : k/2 + n_0] median candidates
6         L_i = S[k/2 - n_i : k/2 - n_{i-1}] buckets of elements presumed smaller than median
7         R_i = S[k/2 + n_{i-1} : k/2 + n_i] buckets of elements presmued larger than median
8 return L, C, R
```

Anschließend folgt die *Probing Phase*, welche für die Filterung dieses Algorithmus verantwortlich ist. Hierbei wird die Menge S aus der *Sampling Phase* genutzt, um alle übrigen Elemente so zu Filtern, dass am Ende die Menge C eine Teilmenge der Ursprungsmenge X darstellt, mit Elementen in einer gewissen Intervall um den Median herum. Insbesondere beinhaltet diese Menge den Median selbst.

Algorithm 2: RMedian : Probing Phase

```
1  procedure Probing Phase(L, C, R)
2  for x_i ∈ X\S in random order :
3      for j ∈ [b − 1, ··· , 1] in order :
4          x_A ← arbitrary element in L_j with fewest compares
5          c ← 1 if x_A is marked else 2
6          if x_i < x_A :
7              add x_i as new pivot to L_{b+c} if j < b − c and mark it, otherwise discard x_i
8              stop processing x_i
9          x_B ← arbitrary element in R_j with fewest compares
10         c ← 1 if x_B is marked else 2
11         if x_i > x_B :
12             add x_i as new pivot to R_{b+c} if j < b − c and mark it, otherwise discard x_i
13             stop processing x_i
           ;                          // By now it is established that S[k/2 − n_1] ≤ x_i ≤ S[k/2 + n_1]
14         add x_i as median candidate to C
15 return L, C, R
```

Am Ende folgt die *Cleaning Phase*, die den Abschluss des Algorithmus bildet. Bei unglücklicher Wahl der Sample-Menge S muss der Algorithmus abgebrochen werden. Dies geschieht, wenn das Sample nicht repräsentativ für die gesamte Menge steht, der relative Median des Samples sich also stark von dem Realen unterscheidet. Dieser Fall tritt jedoch nur mit einer äußerst geringen Wahrscheinlichkeit auf. Entsprechend der Mächtigkeit der entstandenen Menge C ruft der Algorithmus sich nun auf dieser Menge erneut auf, oder sortiert diese Menge mit dem *AKS*-Algorithmus und gibt den Median aus.

Algorithm 3: RMedian : Cleaning Phase

```
1  procedure Cleaning Phase(L, C, R, n0)
2  if max(∑_i |L_i|, ∑_i |R_i|) > n/2 :
3      return DetMedian(X) ;          // Partitioning too imbalanced ⇒ median not in C
4  k = ∑_i(|L_i| − |R_i|)
5  if k < 0 :
6      add k arbitrary elements from ⋃_i R_i to C
7  else:
8      add k arbitrary elements from ⋃_i L_i to C
9  if |C| < log^4(n0) :
10     sort C with AKS and return median
11 return RMedian(C, k(n), b(n))
```

Die Ausgabe des Algorithmus ist der Median der Eingabemenge X.

4.2 *Analyse*

Im diesem Abschnitt wird der Algorithmus RMinimum im Gesamten und seine einzelnen Phasen theoretisch analysiert.
Hierbei ist die Auswahl der diskutierten Bereiche und Theoreme auf die für diese Arbeit relevanten Aussagen eingeschränkt. Für weitere Informationen sei hier auf das Paper [2] verwiesen.

Phase 1. In der ersten Phase des Algorithmus wird eine zufällige Teilmenge $S \subset X$ gebildet und mit dem *AKS*-Algorithmus [6] sortiert. Somit muss unterschieden werden, ob sich das Median Element zufällig in S befindet oder nicht. Dies geschieht offensichtlich mit der Wahrscheinlichkeit $\mathbb{P}[med \in S] = |S|/|X| = k/n$.
In diesem Fall benötigt der Algorithmus durch *AKS* $\mathcal{O}(k \log_2(k))$ Vergleiche, wovon neben weiteren nicht-Median Elementen nun auch das Median Element selbst betroffen ist.
Andernfalls gilt dies nur für nicht-Median Elemente und insbesondere $f_{med}(n) = 0$.

Phase 2. Die zweite Phase des Algorithmus filtert Elemente aus X, die als Median-Kandidat ausgeschlossen werden können. Des Weiteren ist festzustellen, dass sich das Median Element selbst am Ende der zweiten Phase definitiv in der Menge C aller Median-Kandidaten befindet.
Es ist zu bemerken, dass für das Median Element selbst eine Fallunterscheidung zutrifft. Hat sich das Median Element in der in Phase 1 ausgewählten Teilmenge S befunden, so befindet es sich, bis auf sehr wenige in Phase 3 abgefangene Ausnahmen, bereits in der Menge C. Somit wird es im Laufe der zweiten Phase nicht weiter verglichen und es gilt $f_{med}(n) = 0$. Befand sich der Median jedoch nach der ersten Phase nicht in der Menge S, so wird er mit je einem Element aus jedem Behälter L_i und R_i in L oder R verglichen, sodass gilt $f_{med}(n) = |L| + |R| = b + b = 2 \cdot b$.
Über die *Fragile Complexity* aller übrigen Elemente lässt sich aufgrund der zufälligen Auswahl der Filter-Elemente keine Abschätzungen bilden.

Phase 3. Die dritte Phase des Algorithmus überprüft die Filterung der zweiten Phase und entscheidet anschließend, wie weiter verfahren wird. Wie in der ersten Phase findet auch hier kein Vergleich zwischen zwei Elementen statt. Anhand dessen wird entschieden, ob abgebrochen und der Median deterministisch bestimmt, die Menge mit dem *AKS*-Algorithmus sortiert oder ein rekursiver Aufruf gestartet werden muss.

RMedian. Für den gesamten Algorithmus ist anzumerken, dass die Analyse für eine Eingabemenge X der Größe $n \leq 2^{16}$ uninteressant ist. Dies liegt an der in Phase drei gegebenen Schranke für eine Sortierung mit dem *AKS*-Algorithmus, denn für die entsprechende Ungleichung gilt $c \leq \log_2(n)^4 \; \forall c \leq n \leq 2^{16}$.

Lemma 26. *Die erwartete Fragile Complexity des Medians $f_{med}(n)$ ist*

$$\mathbb{E}[f_{med}(n)] = \mathbb{E}\left[f_{med}(\mathcal{O}(\sqrt{n \log(n)}))\right] + \mathcal{O}(\underbrace{\frac{k}{n}}_{\text{Sampled}} + \underbrace{(1 - \frac{k}{n}) \log_d(k)}_{\text{Not sampled}} + \underbrace{1}_{\text{Misclassified}}).$$

Lemma 27. *Die erwartete Fragile Complexity aller nicht-Median Elemente $f_{rem}(n)$ ist*

$$\begin{aligned}\mathbb{E}[f_{rem}(n)] &= \mathbb{E}\Big[f_{rem}(\mathcal{O}(\sqrt{n\log(n)}))\Big] \\ &+ \mathcal{O}(\underbrace{\log(k)}_{\text{Sampled}} + \underbrace{\log_d(n)}_{\text{Not sampled}} + \max(\underbrace{d^2}_{\substack{\text{Pivot in } R_i \\ i>2}}, \underbrace{\frac{nd}{k}}_{\substack{\text{Pivot in } R_j \\ j\leq 2}})),\end{aligned}$$

wobei $d(n) = \Omega(\log^{\varepsilon}(n))$ für ein $\varepsilon > 0$ und $k(n) = \mathcal{O}(n/\log(n))$.

Theorem 28. RMedian *erreicht $\mathbb{E}[f_{med}(n)] = \mathcal{O}(\log\log(n))$ und $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(\sqrt{n})$.*

Beweis. Wähle $k(n) = n^{\varepsilon}$, $d(n) = n^{\delta}$ mit $\varepsilon = 2/3$, $\delta = 1/12$. Nach Lemmata 26 und 27 folgt:

$$\begin{aligned}\mathbb{E}[f_{med}(n)] &= \mathbb{E}\Big[f_{med}(\mathcal{O}(\sqrt{n\log(n)}))\Big] + \mathcal{O}(n^{\varepsilon-1}\varepsilon\log(n) + \tfrac{\varepsilon}{\delta} = \mathcal{O}(\log\log(n)), \\ \mathbb{E}[f_{rem}(n)] &= \mathbb{E}\Big[f_{rem}(\mathcal{O}(\sqrt{n\log(n)}))\Big] + \mathcal{O}((\varepsilon + \tfrac{1}{\delta})\log(n) + \max(2\delta, 1-\varepsilon+2\delta)) = \mathcal{O}(\sqrt{n})\end{aligned}$$

□

Theorem 29. RMedian *erreicht $\mathbb{E}[f_{med}(n)] = \mathcal{O}(\frac{\log(n)}{\log\log(n)})$ und $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(\log(n))$.*

Beweis. Wähle $k(n) = \frac{n}{\log^{\varepsilon}(n)}$, $d(n) = \log^{\delta}(n)$ mit $\varepsilon = \delta = 1/3$. Nach Lemmata 26 und 27 folgt:

$$\begin{aligned}\mathbb{E}[f_{med}(n)] &= \mathbb{E}\Big[f_{med}(\mathcal{O}(\sqrt{n\log(n)}))\Big] + \mathcal{O}(\log^{1-\varepsilon}(n) + \log_{\log^4(n)}(n)) = \mathcal{O}(\tfrac{\log(n)}{\log\log(n)}), \\ \mathbb{E}[f_{rem}(n)] &= \mathbb{E}\Big[f_{rem}(\mathcal{O}(\sqrt{n\log(n)}))\Big] + \mathcal{O}(\log(n) + \max(\log^{2\delta}(n) + \log^{\varepsilon+\delta}(n)) = \mathcal{O}(\log(n))\end{aligned}$$

□

Theorem 30. *Für $k = \mathcal{O}(n/\log(n))$ und $d = \Omega(\log(n))$,* RMedian *benötigt in Erwartung insgesamt $\mathcal{O}(n)$ Vergleiche. Dies impliziert eine erwartete Arbeit $w(n) = \mathcal{O}(n)$.*

Die aufgeführten Theoreme und Lemmata dienen als Grundlage für den Analyseschwerpunkt und somit auch für die zugrunde liegende Implementierung.

5 *Implementierung*

In diesem Abschnitt wird die beiliegende Implementierung der Algorithmen RMinimum und RMedian vorgestellt und die Auswahl der Unittests sowie die Funktionalität weiterer beiliegender Dateien diskutiert.

Als grundlegende Programmiersprache wurde für beide Projekte *Python* gewählt. Bei der Implementierung wurde weitestgehend auf externe Bibliotheken verzichtet, lediglich *matplotlib* und *pandas* zur Datenanalyse sowie *PyTest* für beiliegende Testfälle wurden eingebunden.

Der vollständige Code dieser Arbeit ist bei *Github* [1] veröffentlicht. Für das gesamte Projekt sowie die einzelnen Algorithmen liegen *README.md* Dateien zur besseren Strukturierung bei. An kritischen Stellen ist der gesamte Code einheitlich mit Kommentaren versehen, so dass hiermit auf die Ausführung einzelner Codesegmente verzichtet wird.

5.1 RMinimum

Sowohl der Algorithmus RMinimum selbst als auch alle vier Phasen des Algorithmus liegen als separate Dateien vor und sind unabhängig voneinander ausführbar. Der Hauptanteil der Arbeit des Algorithmus besteht aus dem Bilden von Teilmengen einer gegebenen Menge sowie dem Bestimmen des Minimums einer Menge durch einen perfekt ausbalancierten Turnierbaum. Aus diesem Grund wurde zur Speicherung der Eingabemenge als Datentyp Listen gewählt. Diese erlauben ein unkompliziertes Aufteilen in Teilsegmente, wie bei RMinimum in der zweiten und dritten Phase sowie bei RMedian in der Ersten vonnöten. Des weiteren erlaubt der Index basierte Zugriff auf einzelne Elemente eine erleichterte Speicherung der Anzahl benötigter Vergleiche.

Die von RMinimum genutzten perfekt ausbalancierten Turnierbäume wurden mit Hilfe einer Warteschlange realisiert. Für einen Turnierbaum mit n Elementen werden zunächst alle Elemente in einer Warteschlange gespeichert. Anschließend werden nach dem *First-In-First-Out* Prinzip zwei Elemente entnommen und miteinander verglichen. Das kleinere der beiden Elemente wird daraufhin wieder in die Warteschlange eingefügt. Dieser Vorgang wird so lange wiederholt, bis sich nur noch ein einziges, als Turniersieger deklariertes Element in der Warteschlange befindet. Der Vorteil dieses Verfahren ist, dass nur eine Warteschlange der Länge n und keine rekursiven Aufrufe benötigt werden.

Die Anzahl der Vergleiche, an denen jedes Element teilgenommen hat, wird für jedes Element separat gespeichert.

5.2 RMedian

Für RMedian liegt ebenfalls der Algorithmus wie auch die drei Phasen als separate Dateien vor. Da auch dieser Algorithmus mit dem Aufteilen von Mengen anhand von Indizes arbeitet, wurde auch hier als Datentyp Listen gewählt. Dies erlaubt zusätzlich eine hohe Konsistenz in der Auswertung beider Algorithmen und ihrer gespeicherten Daten. Für jedes Element werden zum einen die Anzahl der Vergleiche gespeichert, an denen es teilgenommen hat, und zum anderen die in der zweiten Phase auftretende Markierung gespeichert.

Der Code beider Algorithmen liegt der Arbeit in elektronischer Form bei, ist jedoch auszugsweise angehängt. Da der Schwerpunkt dieser Arbeit auf der Auswertung der Daten

[1] Github Repository: https://github.com/jfklorenz/Bachelor-Thesis

liegt und der Code an entsprechenden Stellen kommentiert wurde, wird hier auf weitere Ausführungen bezüglich der exakten Implementierung verzichtet.

5.3 *Modultests*

Für die experimentelle Auswertung dieser Arbeit ist es notwendig zu garantieren, dass die Vorliegende Implementierung funktional arbeitet. Um dies zu gewährleisten wurden im Vorfeld Modultests durchgeführt.

5.3.1 *PyTest*

Für die reine Validierung einzelner Codesegmente und für die Korrekte Ausgabe des Minimums beziehungsweise Medians wurde *PyTest* gewählt.
Neben der Möglichkeit einer freien Parametrisierung der Testfälle liegen bereits vordefinierte Eingaben bei. Diese beinhaltet zum einen eine randomisierte Parametrisierung und zum anderen ihrem Definitionsbereich nach extreme Belegungen.

Für den Algorithmus RMinimum wurde Für Phase 1 getestet, ob die Mengen W und L gleich mächtig und das Minimum in W enthalten ist. Des weiteren wird garantiert, dass jedes Element an genau einem Vergleich teilgenommen hat.
Für Phase 2 wird zunächst die korrekte Größe der Teilmengen geprüft. Anschließend wird validiert, dass jedes Element an maximal $\lceil \log_2(k) \rceil$ Vergleichen teilgenommen hat sowie das Minimum an exakt so vielen.
Für Phase 3 ist gewährleistet, dass die Größe der Teilmengen und die eine sinngemäße Filterung eingehalten wird. Ebenso wird die vorgesehene Anzahl an Vergleichen stets exakt eingehalten.
Für Phase 4 muss lediglich das korrekte Einhalten der gegebenen Schranke überprüft werden.

Anschließend wurde der gesamte Algorithmus auf die korrekte Ausgabe des Minimums beziehungsweise Medians hin getestet. Insbesondere sei hier hervorzuheben, dass für die geleistete Arbeit $w(n)$ des Algorithmus RMinimum auch für eine randomisierte Parametrisierung stets die Gleichung $n/2 \leq w(n) \leq 2 \cdot n$ eingehalten wurde. Dies bestärkt Theorem 3.

Die Gestaltung der Modultests für den Algorithmus RMedian ist jedoch komplexer. Dies liegt primär an der Tatsache, dass es auf rein theoretischer Basis sehr aufwendig ist, eine gültige Eingabe für die zweite Phase zu generieren. Aus diesem Grund wurde zuerst Phase 1 des Algorithmus validiert. Anstatt nun anschließend die zweite Phase allein zu testen, wurden die ersten beiden Phasen nun zusammen durchlaufen.
Für Phase 1 wurde die korrekte Konstruktion der Mächtigkeit aller Teilmengen L_i, R_i, L, C und R getestet.
Für Phase 2 wurde überprüft, ob sich der Median am Ende tatsächlich in der Menge C der Median-Kandidaten befindet. Außerdem wurde gewährleistet, dass die *Fragile Complexity* des Medians $f_{med}(n)$ nicht größer ist als die Anzahl der linken und rechten Behälter beziehungsweise Mengen, also $f_{med}(n) \leq |L| + |R| = 2 \cdot b$.
Für Phase 3 wurden für jeden der drei möglichen Ausgänge Eingaben konzipiert, um zu kontrollieren, ob immer eine korrekte Auswahl getroffen wurde.

5.3.2 *Jupyter Notebook*

Um die korrekte Auswertung der *Fragile Complexity* $f(n)$ und der Arbeit $w(n)$ zu gewährleisten, liegen ausführlichere Experimentalfälle als *Jupyter Notebook* Dateien bei. Diese erlauben eine schnelle Reproduzierbarkeit einzelner Experimente.
Insbesondere liegen für die hier diskutierten Theoreme spezielle Dateien bei, die es dem Nutzer ermöglichen sollen, deren Aussagen für manuelle Eingaben effizient zu überprüfen.

5.4 *Werkzeuge*

Im Laufe dieser Arbeit sind ein paar zusätzliche Funktionalitäten implementiert worden, die den Umgang mit den Daten erleichtern sollen.
Da zur empirischen Auswertung oft nur bedingte Zeitintervalle zur Verfügung stehen bietet die beiliegende *Python* Datei *merge.py* die Option, in verschiedenen *.csv*-Datei gespeicherte Daten der gleichen Eingabe in Form von *Pandas DataFrames* zu einer einzigen, korrekt benannten Datei zusammenzufügen.

Des Weiteren liegen alle Methoden bei die benötigt werden, um die in dieser Arbeit verwendeten Grafiken zu reproduzieren oder für andere Datensätze neu auswerten zu können.

Zuletzt werden in dieser Arbeit wiederholt Datensätze gegen eine bestimmte Funktion gefittet. Dies wurde durch beiliegende *Gnuplot*-Dateien realisiert und kann für neue Datensätze flexibel angepasst werden.

Für weitere Informationen bezüglich eines beiliegenden Werkzeugs sei hier auf die entsprechende *README.md* verwiesen.

6 Experimentalfälle

Die primäre Aufgabe dieser Arbeit ist die experimentelle Auswertung der Algorithmen RMinimum und RMedian und die Analyse der Resultate. In diesem Abschnitt wird zunächst diskutiert, was der Fokus der Analyse für beide Algorithmen ist und wie entsprechend unsere Auswahl an Experimentalfällen ausfällt. Hierbei werden wir uns gegebenenfalls auf die in den entsprechenden Kapiteln vorgestellten Theoreme und Lemmata berufen. Zuletzt werden die genutzten Analysewerkzeuge kurz vorgestellt.

Für beide Algorithmen sind die *Fragile Complexity* der Algorithmen $f(n)$, die des Minimums $f_{min}(n)$ oder Medians $f_{med}(n)$ und die aller nicht-Minimum beziehungsweise nicht-Median Elemente von Bedeutung und somit auch die gesamt verrichtete Arbeit $w(n)$ beider Algorithmen.

Hierbei wird sich auf die Aussagen für den Algorithmus RMinimum von Lemma 2 sowie die Theoreme 3, 4 und 5 bezogen und versucht, diese experimentell zu unterstützen.
Analoges gilt bezüglich des Algorithmus RMedian für die Theoreme 28, 29 und 30.
Für den Algorithmus RMedian wird sich auf die experimentelle Unterstützung der Theoreme 4 und 5 beschränkt.
Die ersten drei Phasen von RMinimum sowie die ersten beiden Phasen von RMedian entsprechen einer randomisierten Filterung der Eingabemenge. Das Verhältnis von der Mächtigkeit der Eingabemenge und von der Mächtigkeit der gefilterten Menge entspricht einer Zufallsvariablen $\Delta X \in [0,1]$. Inwiefern diese von den Eingabeparametern den entsprechenden Algorithmen abhängt sowie ihre Verteilung, ist abschließend Bestand der Analyse.

Die bereits diskutierte Implementierung der jeweiligen Algorithmen wurde im Vorfeld auf dem Hochleistungsrechner *Goethe-HLR* der Goethe Universität Frankfurt am Main ausgeführt. Dabei handelt es sich um einen Computer Cluster basierend auf einer Intel CPU Architektur mit einem *Scientific Linux 7.6* Betriebssystem.
Für eine optimale Nutzung des Servers wäre eine Parallelisierung des Codes vorteilhaft gewesen. Dies ist durch die rekursiven Aufrufe innerhalb der Algorithmen jedoch nicht möglich. Aus diesem Grund wurde die wiederholte Ausführung einzelner experimenteller Fälle parallel ausgeführt. Die Ergebnisse wurden in der Arbeit beiliegenden *.csv*-Dateien gespeichert und anschließend lokal ausgewertet.

Beide Algorithmen besitzen als Eingabe die Parameter X und $k(n)$, wobei X eine Menge mit strikter Totalordnung und $k(n)$ einen ganzzahligen Parameter darstellt. Ebenso nutzen beide Algorithmen in der zweiten und dritten Phase die Verhältnismäßigkeiten zwischen n und $k(n)$ zur Bestimmung der Teilmengengröße. Aus diesem Grund wurde sich bei den hier präsentierten Daten für eine einheitliche Struktur der Eingaben entschieden, um ihre Vergleichbarkeit untereinander gewährleisten zu können.
Für die Menge X der Eingabe wurde o.B.d.A. für beide Algorithmen $X = \{0, \cdots, n-1\}$, $n = 2^i$, $i, k(n) \in \mathbb{N}$ gewählt. Die spezifische Wahl des Parameters $k(n)$ sowie des Parameters d des Algorithmus RMedian wird an entsprechender Stelle genauer ausgeführt.
Für den Algorithmus RMinimum wurde jeder Fall zehntausend mal, für RMedian unterschiedlich oft ausgeführt. Die in den folgenden Kapiteln angeführten Grafiken visualisieren stets alle Ausführungen einer festen Eingabe oder den Verlauf bei Änderung eines Parame-

ters. In letzterem Fall wurde stets der Mittelwert aller Wiederholungen der entsprechenden Eingabe als Repräsentant für die Darstellung gewählt.

Nun gilt es, die experimentellen Daten mit den Prognosen der entsprechenden Theoreme und Lemmata abzugleichen. Um mit ausreichender Gewissheit eine Aussage annehmen oder ablehnen zu können, muss insbesondere eine aussagekräftige Unterscheidung zwischen einer einfach-logarithmischen und einer doppelt-logarithmischen Entwicklung getroffen werden.
Für die benötigte nicht-lineare Regression wurde das Programm *Gnuplot* [4, S.74-81] genutzt. Dieses nutzt den Marquardt-Levenberg [5, S.65-68] Algorithmus zur Ausgleichsrechnung, um mit Hilfe der Methode der kleinsten Quadrate einen passenden Fit zu bestimmen. Die Auswertung der Fits sowie die Wahl der Parameter liegen der Arbeit als *.txt*-Datei bei.

6.1 RMinimum

Wir werden nun die experimentellen Auswertungen des Algorithmus RMinimum besprechen. Interessant ist, hierbei festzustellen, wie genau die im Paper [1] aufgeführten Schranken in der Praxis für den gesamten Algorithmus halten. Des Weiteren besteht ein besonderes Interesse an der Güte der Filterung der Phasen 1-3 innerhalb eines Rekursionsschrittes, da diese einen maßgeblichen Einfluss auf die Anzahl der insgesamt benötigten Rekursionsaufrufe aufweist.

6.1.1 *Filter*

Wir werden nun das Zusammenwirken der ersten drei Phasen von RMinimum genauer untersuchen. Hierbei ist der Zusammenhang zwischen einzelnen Parametern und der Güte der Filterung besonders interessant.

Wir definieren nun $\Delta X = |W'|/|X| = |W'|/n$ als die Güte der Filterung. Dies entspricht dem prozentualen Verhältnis zwischen der wie in Phase 3 festgelegten Mächtigkeit der Menge W' und der Mächtigkeit n der Eingabemenge X. Wir wissen, dass die Menge W' zum einen das Minimum und zum anderen kein Element der Menge L enthält. Es gilt $1 \leq |W'| \leq |X|/2 = n/2$ und somit $0 < \Delta X \leq 1/2$.

Bei einer Anzahl von rep Wiederholungen besitzt ΔX die absolute Häufigkeit $H_{rep}(\Delta X) = \sum_{rep} \mathbb{1}_{\Delta X}$. Die für uns interessanten Begrifflichkeiten der *relativen Häufigkeit*, des *(Stichproben-) Mittelwertes* und der *(Stichproben-) Varianz* ergeben sich nun über:

$$
\begin{array}{lrcl}
\textbf{Relative Häufigkeit:} & h_{rep}(\Delta X) & = & H_{rep}(\Delta X)/rep \\
\textbf{Stichprobenmittelwert:} & \Delta\overline{X} & = & \frac{1}{rep}\sum_{rep}\Delta X \\
\textbf{Stichprobenvarianz:} & Var[\Delta X] & = & \sum_{i=1}^{rep}(\Delta X_i - \overline{\Delta X})^2 \cdot h_{rep}(\Delta X)
\end{array}
$$

Da der Wert von ΔX von der Randomisierung der Phasen 1-3 abhängig ist, kann er als Zufallsvariable mit Wertebereich $(0, 1/2]$ aufgefasst werden.

Führt man den Algorithmus mit gleicher Eingabe wiederholt aus, so erwartet man nach dem (schwachen) Gesetz der großen Zahlen eine resultierende Normalverteilung der Ergebnisse. Wie in Abbildung 3 zu sehen ist mit steigender Anzahl an Wiederholungen immer klarer eine glockenförmige Darstellung der Verteilung der relativen Häufigkeiten $h_{rep}(n)$ zu erkennen. Hierbei sei anzumerken, dass bei einer solchen Darstellung der visuelle Trugschluss entsteht, dass sowohl der Extrempunkt als auch die Fläche unter der vermeintlichen Kurve zu sinken scheinen.

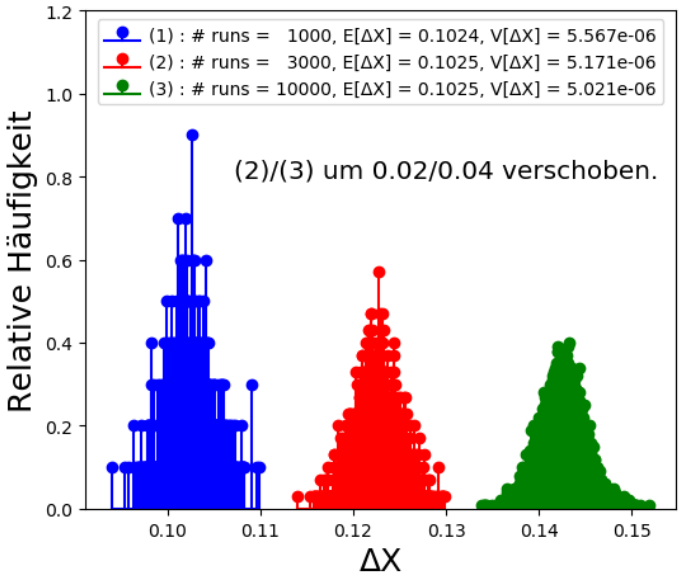

Abbildung 3: $\log_2(n) = 16$, $k = 64$

Dies ist der diskreten Natur von Messwerten geschuldet. Bei einer Normalverteilung entspricht der Extrempunkt der Dichtefunktion dem Erwartungswert der zugehörigen Zufallsvariablen. Die in der Abbildung mit angegebenen Werte dienen zur Verdeutlichung der realen Entwicklung.
Von einer zugrunde liegenden Normalverteilung ausgehend wird versucht anhand der Parameter Abschätzungen für den Erwartungswert und die Varianz zu bilden.

Um die Abhängigkeiten zu einzelnen Parametern getrennt nachzuweisen wurde für die Eingabe der Form (n,k) stets einer der beiden Parameter fixiert, während für den anderen verschiedene Werte durchlaufen wurden. Dies erlaubt bestehende Korrelationen zu dem variablen Parameter zu erkennen. Für jeden hier erwähnten Fall wurden ohne anderweitige Angabe jeweils zehntausend Durchläufe ausgeführt und die Ergebnisse von ΔX separat gespeichert.

Im Folgenden wird die relative Häufigkeit mit $h(n)$, der Mittelwert mit μ sowie die Varianz mit $V(n,k)$ bezeichnet. Die relative Häufigkeit entspricht graphisch der Dichtefunktion der Zufallsvariable ΔX.

Erwartungswert

Nun soll die Abhängigkeit des Erwartungswertes von den Parametern n und $k(n)$ untersucht werden. Hierzu wird zuerst der Parameter k nacheinander auf verschiedene Werte fixiert und die Veränderung des Erwartungswertes abhängig von n gemessen.
Den präsentierten Grafiken liegt die Fixierung von $k \in \{4, 32, 256\}$ für $\log_2(n) \in \{9, \cdots, 20\}$ zugrunde.

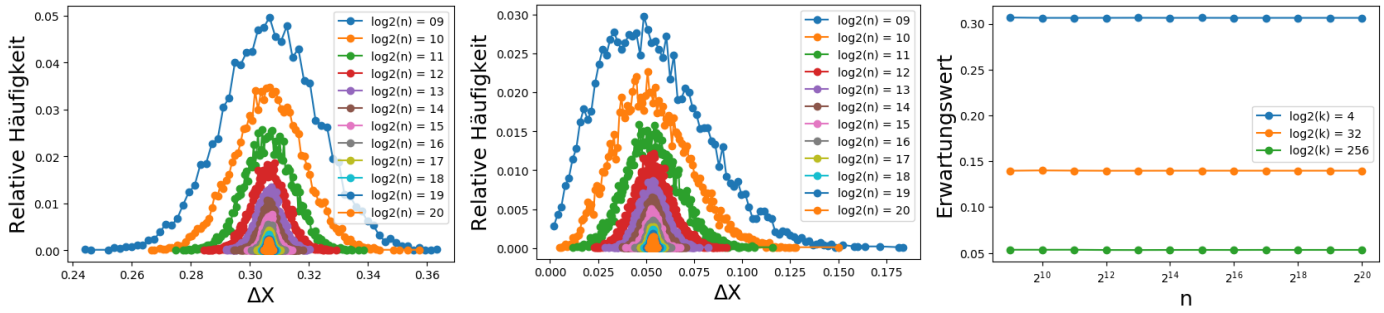

Abbildung 4: $h(n)$ für $k = 4$ und $k = 256$ sowie μ für $k = 4, 32, 256$.

Wie in Abbildung 4 eindeutig zu erkennen, ist bleibt der Erwartungswert nach Fixierung von k konstant für alle Werte für n.

Nun wird der Parameter $n = 2^{16}$ fixiert und alle Werte für $k = 2^i$, $i \in \mathbb{N}$, $k < n$ untersucht.

Diesbezüglich wird sich nun auff die Aussage von Lemma 2 berufen. Demnach gilt $\mathbb{E}[|W_i'|] \leq 2d\sqrt{k}$ für eine Konstante $d > 0$. Insgesamt existieren $n/2k$ Teilmengen, womit folgt:

$$\frac{|W'|}{|X|} = \frac{1}{n} \cdot \sum_{i=1}^{n/k} \mathbb{E}[|W_i'|] = \frac{1}{n} \cdot \frac{n}{2k} \cdot \mathbb{E}[|W'|] \leq \frac{1}{2k} \cdot 2d\sqrt{k} = \frac{d}{\sqrt{k}}$$

Anhand der empirischen Daten wurde nun mithilfe des Programms Gnuplot [4] versucht ein möglicher Fit für eine Abschätzung von $\mathbb{E}[\Delta X]$ zu berechnen.

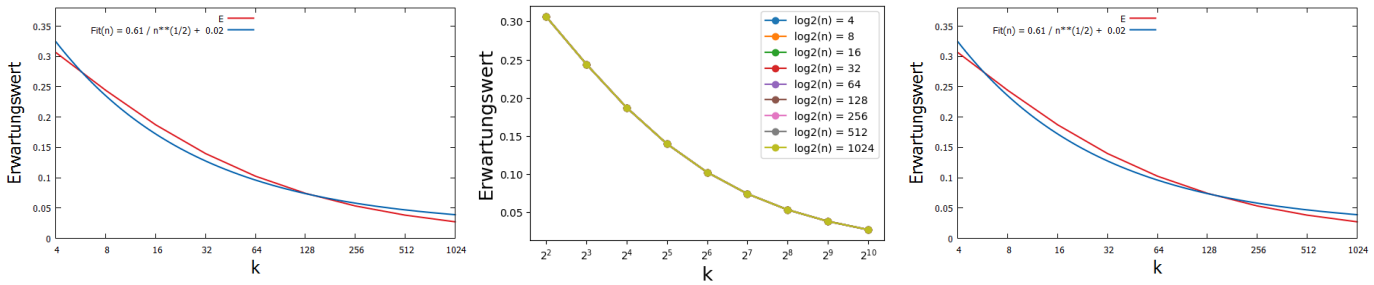

Abbildung 5: $h(n)$ und μ für $\log_2(n) = 16$, $\log_2(k) \in \{1, \cdots, 15\}$ sowie Fit für μ.

Als Fit Funktion wurde hierfür $F_E(k) = a/\sqrt{k} + b$ und als Startwerte $a = 0.5$ sowie $b = 0.01$ gewählt. Nach 5 Iterationen ist der Fit konvergiert und die finale Summe der Residuenquadrate betrug 0.0011. Es ergab sich eine Parametrisierung von 0.61 und $b = 0.03$. Vergleichbare Experimente ergaben ähnliche Ergebnisse, was die Aussage von Lemma 2 bestätigt.

Varianz

Nun kann die Abhängigkeit der Varianz von den Parametern n und $k(n)$ untersucht werden. Wie schon beim Erwartungswert wird mit der Abhängigkeit von n begonnen und k dafür auf verschiedene Werte fixiert. Für eine direkte Vergleichbarkeit wurde für die Grafiken der gleiche Datensatz genutzt.

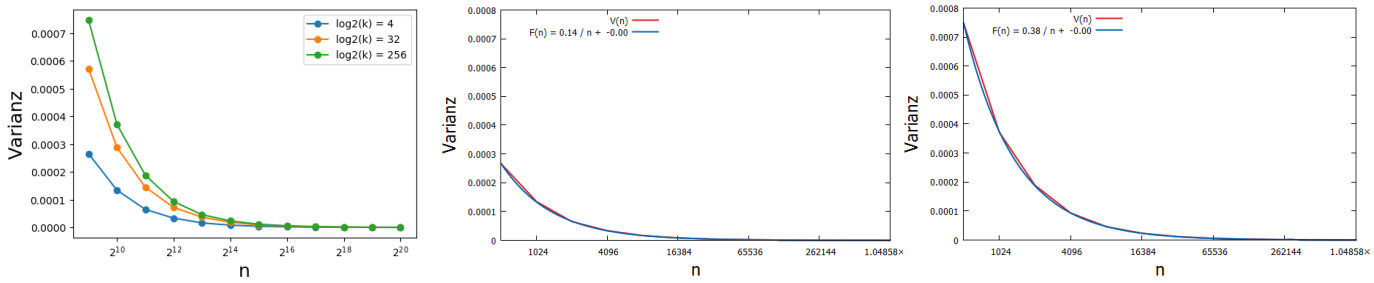

Abbildung 6: $V(n)$ für $k \in \{4, 32, 256\}$ sowie Fit für $k \in \{4, 256\}$.

Wie in Abbildung 6 zu sehen ähnelt der Verlauf der Varianz $V(n)$ dem einer Hyperbel. Der Fit wurde dementsprechend mit $F_V(n) = a/n + b$ mit den Startwerten $a = 0.4$ und $b = 0.0000001$ gewählt. Nach 5 Iterationen ist der Fit konvergiert. Die finale Summe der Residuenquadrate und weitere relevante Parameter liegen dieser Arbeit bei. Anzumerken ist hier, dass sich die Parametrisierung von $b = 0$ für alle experimentellen Werte von k ergab.

Abschließend wird noch die Abhängigkeit der Varianz $V(n,k)$ von dem Parameter k geprüft und n wie bereits zuvor auf $\log_2(n) = 16$ fixiert.

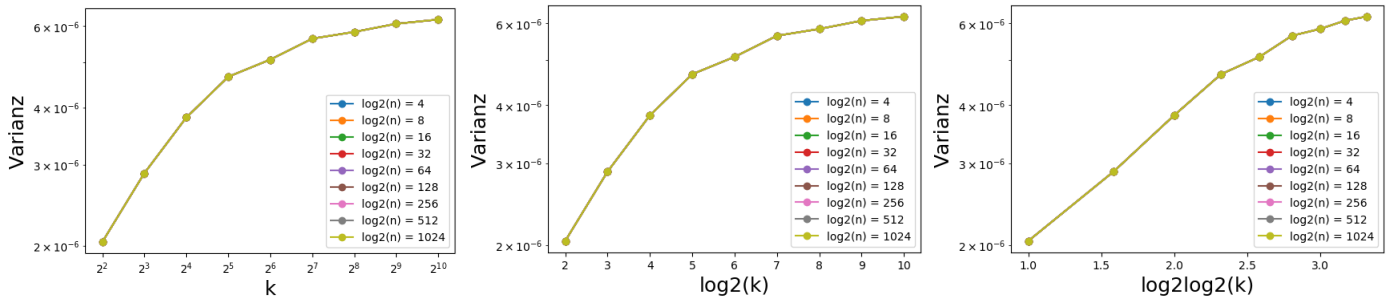

Abbildung 7: $V(k)$ für $\log_2(n) = 16$ bezüglich k, $\log_2(k)$ sowie $\log_2 \log_2(k)$.

Da sich im Verlauf der Experimente eine logarithmischer Zusammenhang erkennen ließ, wurden wie in Abbildung 7 zu sehen die Werte von $V(n)$ zuerst gegen die Werte von k, dann gegen die logarithmierten und abschließend gegen die doppelt-logarithmierten Werten von k abgetragen. Ist bei der einfach-logarithmischen Darstellung noch eindeutig eine Kurve zu erkennen, so deutet sich bei der doppelt-logarithmischen Darstellung klar ein linearer Funktionsverlauf an.

Um sicherzustellen, dass es tatsächlich um einen doppelt-logarithmischen Zusammenhang handelt wurden nun zwei potentielle Fits in Form von $F_V^1(k) = a * \log_2(k) + b$ sowie $F_V^2(k) = a * \log_2 \log_2(k) + b$ gebildet. Die nachfolgende Tabelle fasst eine um den Faktor 10^6 skalierte Gegenüberstellung beider Fits zusammen:

	Param a	Param b	Sum Res	Δ Last Iter
$F_V^1(k)$:	0.5212	1.5603	1.3470	$-4.6484e-14$
$F_V^2(k)$:	1.9097	0.0636	0.1797	$-9.00343e-10$

An Parameter b lässt sich bereits erkennen, dass $F_V^1(k)$ eine merkliche Verschiebung auf der Y-Achse benötigt, um eine passende Regression zu bilden. Dies sowie die um den Faktor

1.347/0.1797 = 7.4958 höhere Summe der Residuenquadrate legen nahe, dass es sich bei $F_V^2(k)$ um die präzisere Abschätzung handelt, was auch in Abbildung 8 noch einmal verdeutlicht wird.

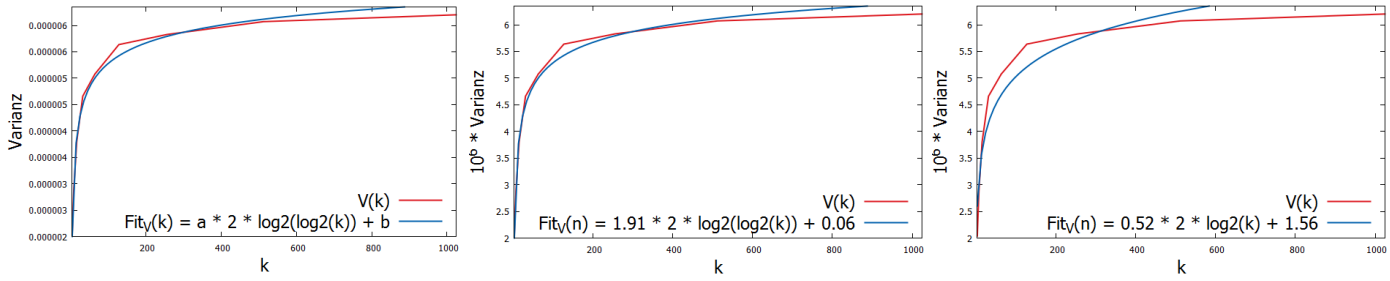

Abbildung 8: (Skalierter) Fit für $V(k)$ für $\log_2(n) = 16$ bezüglich $\log_2(k)$ und $\log_2 \log_2(k)$.

Abschließend ist festzuhalten, dass die Güte ΔX der Filterung einer normalverteilten Zufallsvariable entspricht, deren Erwartungswert von $\sqrt{k}$ und deren Varianz von $1/n$ sowie $\log_2 \log_2(k)$ abhängig ist.

6.1.2 *Theorem 4*

Nach Theorem 4 benötigt RMinimum $\mathbb{E}[f_{min}(n)] = \mathcal{O}(\varepsilon^{-1} \log\log(n))$ Vergleiche für das Minimum und $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(n^{\varepsilon})$ für alle übrigen Elemente.
Somit hat die Eingabe nun die Form (n, n^{ε}). Im Folgenden wird zuerst der Parameter ε und anschließend der Parameter n fixiert, um die Abhängigkeit zu dem jeweils anderen Parameter zu überprüfen.
Für die aufgeführten Experimente wurden die Parameter n und ε so gewählt, dass n^{ε} weiterhin eine Zweierpotenz ist.

Nun werden die Auswertungen für $\varepsilon = 1/2, 1/4, \gamma$ mit $0 < \gamma \ll 1/2$ diskutiert.
Es wurden wie auch im Anhang beiliegend weitere Werte durchlaufen, welche jedoch keine nennenswerten Unregelmäßigkeiten aufweisen.

Für $\varepsilon = 1/2$ wird eine *Fragile Complexity* von $\mathbb{E}[f_{min}(n)] = \mathcal{O}(2 \cdot \log\log(n))$ für das Minimum und $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(n^{1/2})$ für alle übrigen Elemente erwartet.
Um diese Aussage zu überprüfen, wurde der Algorithmus nun wiederholt mit dem Eingabetupel $(n', n'^{1/2})$ für die Werte $n' = 2^{2 \cdot i}$, $i \in \{3, \cdots, 20\}$ jeweils Zehntausend mal durchlaufen.

Fragile complexity des Minimum Elements

Zuerst untersuchen wir die *Fragile Complexity* des Minimums $f_{min}(n)$.

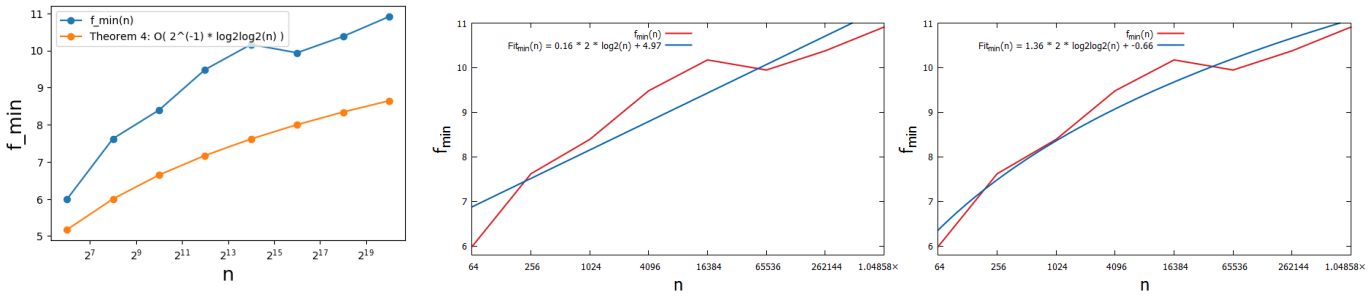

Abbildung 9: (Skalierter) Vorhersage und Fit für $f_{min}(n)$ mit $k(n) = n^{1/2}$.

Wie im ersten Bild der Abbildung 9 zu sehen wurde die aus Theorem 4 stammende Abschätzung ohne weitere Parametrisierung in ein Koordinatensystem eingetragen. Abgesehen von einer leichten Verschiebung stimmen die Bilder der Funktionen gut überein. Im mittleren Bild wird eine einfach-logarithmischer Abschätzung dargestellt. Wie jedoch nach dem logarithmischen Skalieren der X-Achse zu sehen ist, weist $f_{min}(n)$ weiterhin eine kurvenförmige Darstellung auf, womit der einfach-logarithmische Fall nahezu ausgeschlossen werden kann.

In der Praxis wurde sowohl ein Fit der Form $F^1_{min}(n) = a \cdot \varepsilon^{-1} \cdot \log_2(n) + b = a \cdot 2 \cdot \log_2(n) + b$ als auch ein Fit der Form $F^1_{min}(n) = a \cdot 2 \cdot \log_2 \log_2(n) + b$ mit Startwerten $a = 1$ und $b = 0.001$ versucht. Es ergab sich folgende Gegenüberstellung beider Fits:

	Param a	Param b	Sum Res	Δ Last Iter
$F^1_V(k)$:	0.1595	4.9664	2.1959	$-2.03746e-09$
$F^2_V(k)$:	1.3571	-0.6563	0.7404	$-1.34947e-13$

Die Tatsache, dass der Parameter b im einfach-logarithmischen Fall um den Faktor 7.57 weiter von 0 entfernt konvergiert ist als der doppelt-logarithmische, zusammen mit der deutlich höheren Summe der Residuenquadrate lassen den Schluss zu, dass die Vorhersage des Theorems für die *Fragile Complexity* des Minimums $f_{min}(n)$ auch in der Praxis bestätigt werden kann.

Fragile complexity aller nicht-Minimum Elemente

Nun wird die *Fragile Complexity* aller übrigen Elemente $f_{rem}(n)$ untersucht. Theorem 4 prognostiziert in diesem Fall eine *Fragile Complexity* von $\mathcal{O}(n^\varepsilon)$. Vergleicht man die experimentell gemessenen Werte für $f_{rem}(n)$ mit dem Term n^ε bei gleicher Parametrisierung, so ergibt sich eine nahezu perfekte Übereinstimmung. Aus diesem Grund wurde in diesem Fall auf einen Fit verzichtet.

Wie in Abbildung 10 ebenfalls zu sehen wird die in Theorem 3 prognostizierte Linearität der Arbeit $w(n)$ exakt eingehalten. Abschließend sei noch bemerkt, dass der Algorithmus erst ab einer Mächtigkeit der Eingabemenge $\log_2(n) = 10$ den ersten rekursiven Aufruf tätigt.

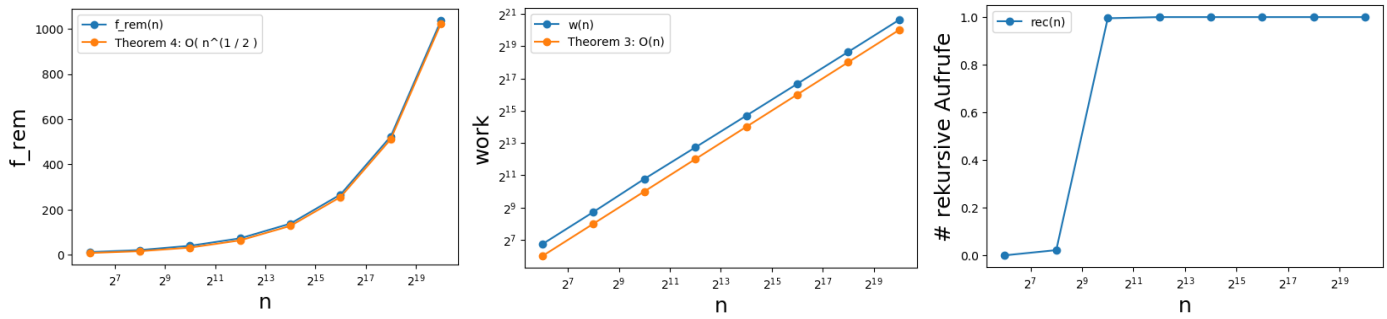

Abbildung 10: Vorhersage für $f_{rem}(n)$, $w(n)$ und $rec(n)$ mit $k(n) = n^{1/2}$.

6.1.3 *Theorem 5*

Nach Theorem 5 benötigt RMinimum $\mathbb{E}[f_{min}(n)] = \mathcal{O}(\varepsilon^{-1} \log\log(n))$ Vergleiche für das Minimum und $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(n^{\varepsilon})$ für alle übrigen Elemente.

Da $k(n)$ bei der Eingabe ganzzahlig sein muss, wurde der Wert vor der Eingabe sowohl auf- als auch abgerundet und beide Fälle separat betrachtet. Wie zuvor wird für beide Fälle einen passenden Fit $F_{\lceil k \rceil}(n)$ beziehungsweise $F_{\lfloor k \rfloor}(n)$ untersucht. Für eine präzisere Repräsentation des Parameters $k(n)$ wird zusätzlich ein weiter Fit mit folgender Gewichtung gebildet:

$$F^{\star}(n) = (\lceil k \rceil - k) \cdot f_{\lfloor k \rfloor}(n) + (k - \lfloor k \rfloor) \cdot f_{\lceil k \rceil}(n)$$

Hierbei werden die auf- und abgerundeten Werte relativ entsprechend ihrer absoluten Differenz von dem realen Wert des Parameters $k(n)$ gewichtet. Die Repräsentation für Theorem 5 beruht auf Eingaben für $\log_2(n) = \{6, \cdots, 20\}$, mit denen wie zuvor zuerst die *Fragile Complexity* des Minimums $f_{min}(n)$ und dann die aller übrigen Elemente $f_{rem}(n)$ experimentell analysiert werden.

Fragile complexity des Minimum Elements

Nun wird die Aussage von Theorem 5 bezüglich der *Fragile Complexity* des Minimums $f_{min}(n)$ untersucht. Es wurden nun wie besprochen drei separate Fits bezüglich der entsprechend gesammelten Datenmenge gebildet. Bei einer Wahl der Startwerte von $a = 5$, $b = 0.001$ ist der Fit nach 8, für $a = 6.5$, $b = -8.5$ nach 4 Iteration konvergiert und es ergab sich in beiden Fällen:

	Param a	Param b	Sum Res	Δ Last Iter
$F_{\lceil k \rceil}(n)$:	6.97	−9.91	0.4546	$-3.23746e-09$
$F_{\lfloor k \rfloor}(n)$:	6.42	−7.88	2.2034	$-4.2210e-13$
$F^{\star}(n)$:	6.66	−8.78	1.3096	$-8.8166e-10$

Wie zu erkennen, ist die sich ergebende Parametrisierung von a und b in allen Fällen relativ ähnlich. Die Parameter fallen für $F_{\lfloor k \rfloor}(n)$ am geringsten aus, jedoch ist die Summe der Residuenquadrate hier auch am Höchsten.

Wie in Abbildung 11 zu sehen, weicht die gesammelte Datenmenge vorerst leicht von der Prognose des Theorems ab. Skaliert man diese jedoch entsprechend des errechneten Fits,

so ergibt sich eine beinahe deckungsgleiche Darstellung zu der gesammelten Datenmenge. Der Fit ist so eindeutig, dass auf eine einfach-logarithmische Abschätzung als Vergleich verzichtet wurde.

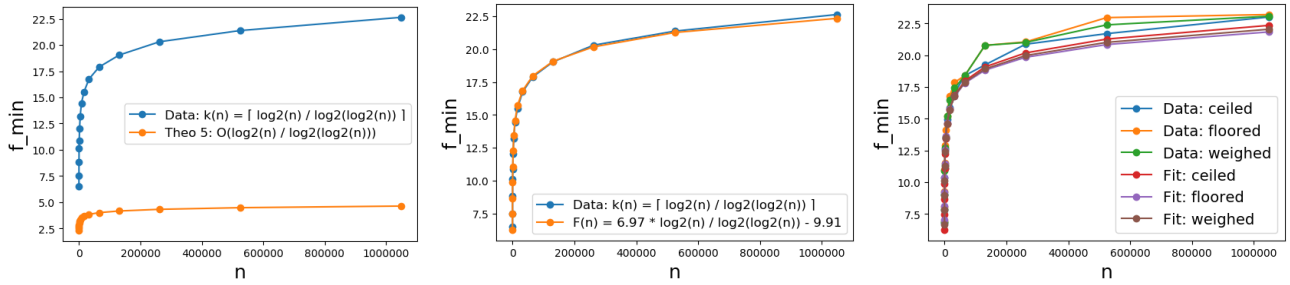

Abbildung 11: Vorhersage und Fit für $f_{min}(n)$ bezüglich Auf- und Abrundung.

Dies lässt die Aussage von Theorem 5 bezüglich der *Fragile Complexity* des Minimums $f_{min}(n)$ eindeutig bestätigen.

Fragile complexity aller nicht-Minimum Elemente

Nun wird eine analoge Betrachtung bezüglich der *Fragile Complexity* aller übrigen Elemente $f_{rem}(n)$ durchgeführt. Die Prognose von Theorem 5 bezüglich der *Fragile Complexity* ist identisch für das Minimum und für alle übrigen Elemente. Das Ergebnis für $f_{rem}(n)$ weist keine Unregelmäßigkeiten auf und es bilden sich ähnliche Abschätzungen wie bereits für $f_{min}(n)$:

	Param a	Param b	Sum Res	Δ Last Iter
$F_{\lceil k \rceil}(n)$:	7.14	−10.38	0.8468	$-7.22016e-09$
$F_{\lfloor k \rfloor}(n)$:	7.44	−10.96	2.0026	$-8.9850e-10$
$F^{\star}(n)$:	7.37	−10.88	1.8968	$-9.3543e-10$

Abschließend sei hier noch anzumerken, dass die für diese Eingabe von RMinimum benötigte Arbeit $w(n)$ mit wachsendem n linear skaliert. Somit kann die Aussage von Theorem 3 weiterhin bestärkt werden und neben einer Darstellung des Fits für $f_{rem}(n)$ abschließend in Abbildung 12 eingesehen werden.

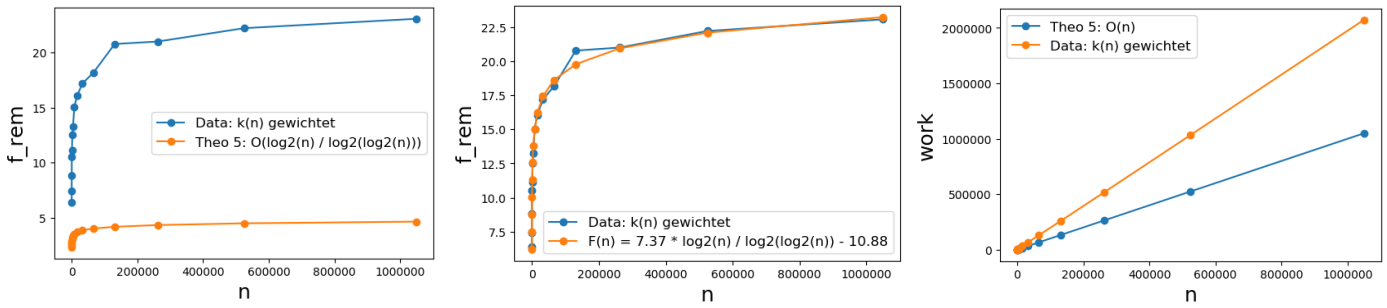

Abbildung 12: Vorhersage und Fit für $f_{rem}(n)$ sowie die Arbeit $w(n)$.

Dies beendet die experimentelle Analyse dieser Arbeit bezüglich des in Paper [1] vorgestellten Algorithmus RMinimum und der *Fragile Complexity* des Minimums sowie aller übrigen Elemente.

Es konnte ein Zusammenhang zwischen den Eingabeparametern n und $k(n)$ sowie der Güte der Filterung der ersten drei Phasen nachgewiesen, sowie die Aussagen der Theoreme 3-5 des Papers eindeutig bekräftigt werden.

Weitere untersuchte Fälle liegen der Arbeit bei und können zusätzlich im Anhang nachgeschlagen werden.

6.2 RMedian

Nun wird eine vergleichbare Analyse für den Algorithmus RMedian durchgeführt. Hierbei wird die Güte der Approximation der Theoreme 28 und 29 aus dem Paper [2, S.24] experimentell untersucht.

Aufgrund der limitierten Zeit dieser Arbeit und der im direkten Vergleich zu RMinimum deutlich höheren Laufzeit, konnte der Datensatz für den Algorithmus RMedian nicht so umfangreich ausfallen wie gehofft. Die vorliegende Implementierung erlaubt diesen jedoch nach Bedarf aufzustocken. Wie bei seiner Analyse bereits festgehalten wurde, ist die Auswertung des Algorithmus RMedian für eine Eingabemenge X mit einer Mächtigkeit von $n \leq 2^{16}$ uninteressant. Somit wurden für den Parameter n die Werte $n = 2^i$, $i \in \{16, \cdots, 20\}$ gewählt.

Um jedoch ein Gefühl für die Auswirkung der Parameter zu gewinnen wurde zum Test der Parameter $d(n)$ auf die nächst größere beziehungsweise kleinere Ganzzahl gerundet. Bei der Wahl von $n = 2^{18}$, $k(n) = 2^{12}$ sowie $d_1(n) = 2$ und $d_2(n) = 3$ halbiert sich, wie in Abbildung 13 zu sehen, sowohl die *Fragile Complexity* des Median Elements, als auch die Arbeit $w(n)$ bei Auf- und Abrundung zwischen 2 und 3. Die *Fragile Complexity* aller nicht-Median Elemente $f_{rem}(n)$ verdoppelt sich hingegen.

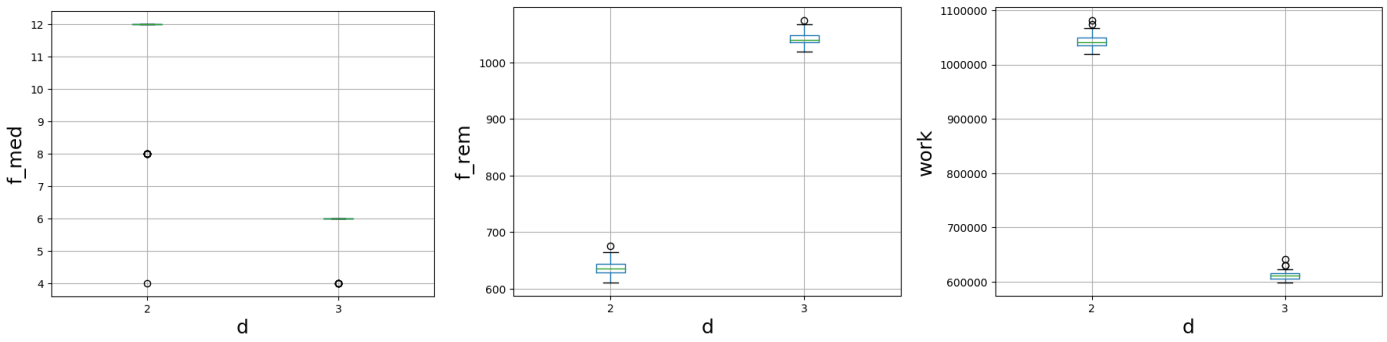

Abbildung 13: Vorhersage und Fit für $f_{rem}(n)$ sowie die Arbeit $w(n)$.

Somit ergibt sich ein merklicher Sprung für die *Fragile Complexity* aller Elemente, sobald $d(n)$ die nächste Ganzzahl überschreitet. Dies hat direkte Konsequenzen für die Analyse dieser Arbeit, da für die notwendige Parametrisierung nach Theorem 28 bei $k(2^{18}) < 3$ sowie $k(2^{19}) \geq 3$ gilt, was das Ergebnis einer Regression verzerrt. Für zukünftige Forschungen wäre es interessant, den Parameter $d(n)$ speziell zu gewichten oder auf bestimmten Intervallen gesondert zu betrachten, was im Rahmen dieser Arbeit zeitlich nicht mehr realisierbar war.

6.2.1 *Theorem 28*

Nach Theorem 28 des Papers [2] erreicht das Median Element für $k(n) = n^{2/3}$ und $d(n) = n^{1/12}$ eine erwartete *Fragile Complexity* von $\mathbb{E}[f_{med}(n)] = \mathcal{O}(\log\log(n))$ und alle übrigen Elemente $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(\sqrt{n})$.

Fragile complexity des Median Elements

Zuerst wird die *Fragile Complexity* des Medians $f_{med}(n)$ untersucht. Wie bereits erwähnt findet ein die Werte verzerrender Sprung statt, wie in Abbildung 14 deutlich zu sehen. Die Arbeit $w(n)$ verhält sich jedoch wie erwartet linear, was Theorem 30 bestätigt.

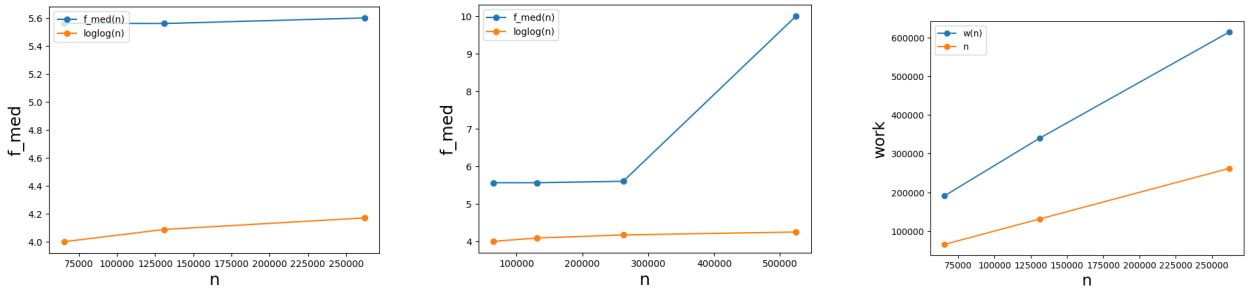

Abbildung 14: $\log(n) = 16, \cdots, 18$ gegenüber $\log(n) = 16, \cdots, 19$ sowie die Arbeit $w(n)$.

Auch wenn die Datenmenge sowie der gesammelte Bereich nicht ausreichen, um eine feste Aussage zu tätigen, wurde die *Fragile Complexity* trotzdem für $\log(n) = 16, \cdots, 18$ untersucht. Die Daten wurden hierbei wieder gegen einen einfach-logarithmischen mit $F^1(n) = a \cdot \log(n) + b$ sowie eine doppelt-logarithmischen Term $F^2(n) = a \cdot \log\log(n) + b$ abgeglichen.

	Param a	Param b	Sum Res	Δ Last Iter
$F^1(n)$:	0.2272	4.6452	0.0002	$-2.42876e-07$
$F^2(n)$:	0.0195	5.2419	0.0002	$-1.9372e-14$

Die sich ergebenden experimentellen Daten sowie ihre grafische Veranschaulichung in Abbildung 15 deuten stark auf einen, wie von Theorem 28 angegebenen, doppelt-logarithmischen Zusammenhang hin. Eine logarithmische Achsenskalierung hilft hier jedoch nicht, um eine Abschätzung jeglicher Form zu unterstützen.

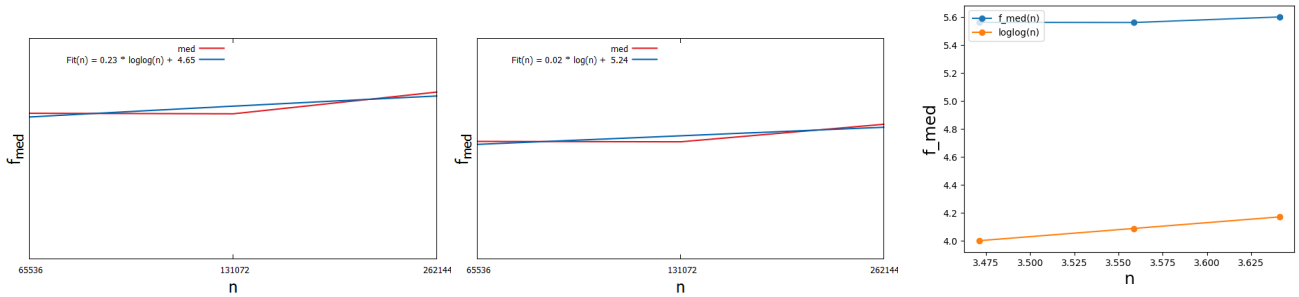

Abbildung 15: log vs log log Fit sowie logarithmische Achsenskalierung.

Fragile complexity aller nicht-Median Elemente

Für die *Fragile Complexity* aller nicht-Median Elemente $f_{rem}(n)$ hingegen konnte eine aussagekräftige Regression gewonnen werden.
Wie in Abbildung 16 zu sehen wurde hier ein Fit der Form $F(n) = a \cdot \sqrt{n} + b$ mit einer Parametrisierung von $a = 2.3374$ $b = -155.424$ bestimmt.

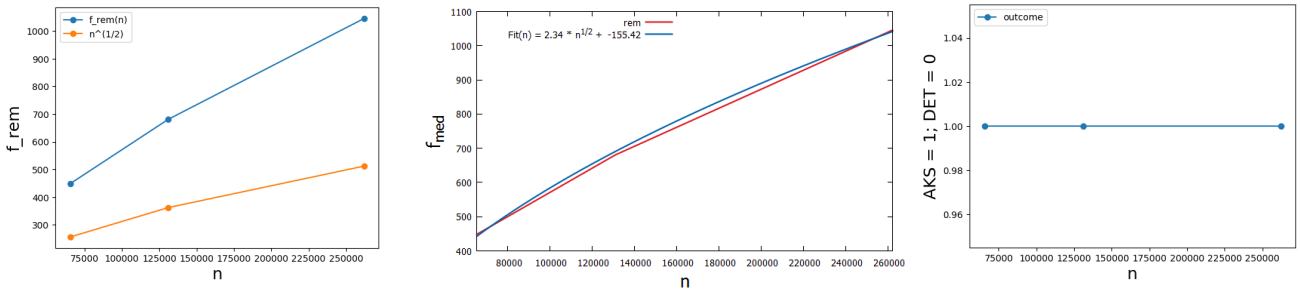

Abbildung 16: Vorhersage und Fit für $f_{rem}(n)$ sowie der Ausgang von Phase 3.

Abschließend sei hier noch angemerkt, dass bei allen für die Grafiken genutzten Experimentaldaten der Algorithmus RMedian mit der Ausführung des *AKS*-Algorithmus endet.

6.2.2 *Theorem 29*

Nach Theorem 29 des Papers [2] erreicht das Median Element für $k(n) = n/\log^{1/3}(n)$ und $d(n) = \log^{1/3}(n)$ eine erwartete *Fragile Complexity* von $\mathbb{E}[f_{med}(n)] = \mathcal{O}(\log\log(n))$ und alle übrigen Elemente eine erwartete *Fragile Complexity* von $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(\sqrt{n})$.

Fragile complexity des Median Elements

Für die *Fragile Complexity* des Medians $f_{min}(n)$ lässt sich die Schranke aus Theorem 29 trotz beschränkter Datenmenge eindeutig bestätigen.
In Abbildung 17 zu sehen erkennt man schon bei direktem Vergleich zwischen den Experimentaldaten und der Funktion $f(x) = \log(x)$ der Daten die bestehende Korrelation. Parametrisiert man diese über einen Fit der Form $F(x) = a \cdot \log(x) + b$, so wie zu erkennen eine gute Übereinstimmung.

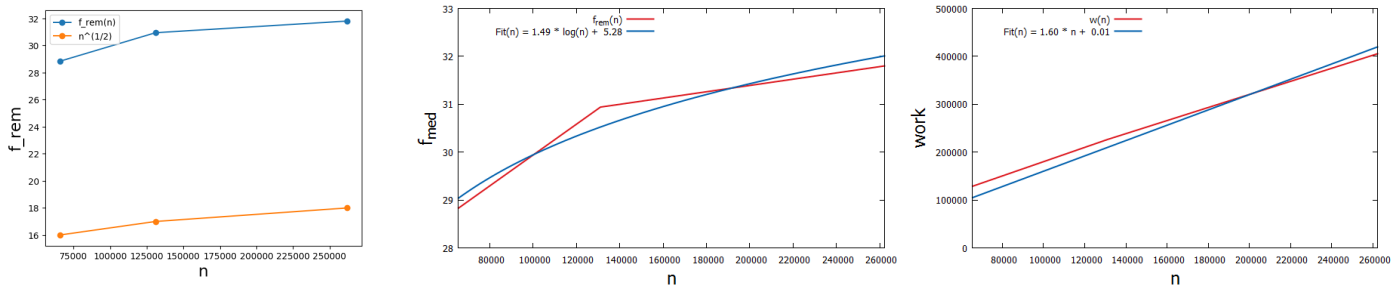

Abbildung 17: Vorhersage und Fit für $f_{rem}(n)$ sowie die Arbeit $w(n)$.

Auch bei einer Parametrisierung nach Theorem 29 wurde der Algorithmus RMedian stets durch den *AKS*-Algorithmus beendet. Um eine Abschätzung für die Wahrscheinlichkeit ei-

ner unglücklichen Wahl eines Samples der ersten Phase des Algorithmus zu liefern bedarf es einer deutlich umfangreicheren Datenmenge, so dass im Rahmen dieser Arbeit diesbezüglich keine Aussage getroffen werden kann.

Fragile complexity aller nicht-Median Elemente

Abschließend untersuchen wir noch die *Fragile Complexity* aller nicht-Median Elemente $f_{rem}(n)$ sowie die von dem Algorithmus RMedian verrichtete Arbeit $w(n)$.
Wie in Abbildung 18 zu sehen ist bereits eine direkte Ähnlichkeit zwischen den Experimentaldaten und der Funktion $f(x) = \log(x)$ erkennbar. Wie zu erwarten konnte ein passender Fit der Form $F(n) = a \cdot \log(n) + b$ mit Parametern $a = 1.49$ und $b = 5.28$ berechnet werden.

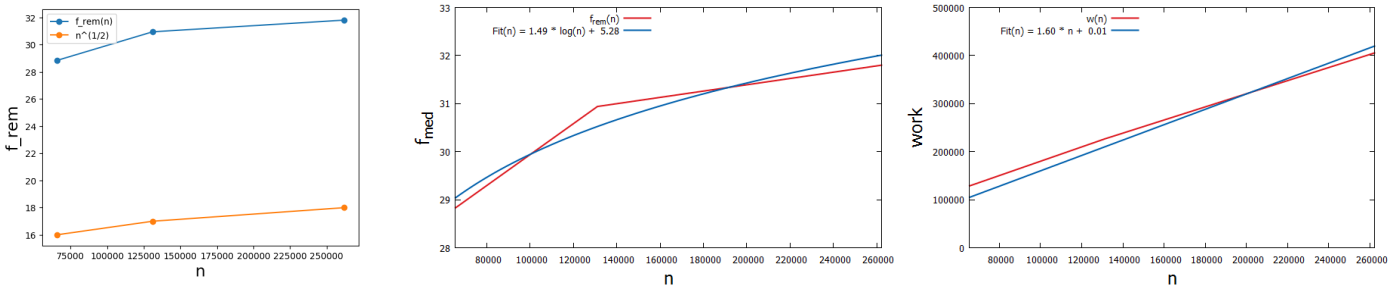

Abbildung 18: Vorhersage und Fit für $f_{rem}(n)$ sowie die Arbeit $w(n)$.

Zur Sicherheit wurde die verrichtete Arbeit $w(n)$ noch gegen einen Fit der Form $F(n) = a \cdot n + b$ abgeglichen, der einen nahezu deckungsgleiche Darstellung ergibt.

7 Ausblick

Im Verlaufe dieser Arbeit konnten die in den Theoremen 3, 3 und 3 der Version [1] aufgestellten Schranken bezüglich der *Fragile Complexity* der Elemente des Algorithmus RMinimum eindeutig bestärkt werden. Auch wenn weitere Testfälle denkbar wären, so ist die Korrelation zwischen den experimentell gesammelten Daten bereits jetzt unverkennbar.

Für den Algorithmus RMedian konnte die Linearität der Arbeit $w(n)$ und somit auch Theorem 30 eindeutig bestätigt werden. Für die *Fragile Complexity* des Medians sowie aller übrigen Elemente scheinen die Schranken für alle bisher angesammelten Daten eine gute Abschätzung zu liefern, so dass die positive Erwartung besteht, auch die Theoreme 28 und 29 aus Paper [2] in naher Zukunft durch weitere Experimentalanalyse weiter stützen zu können.

Von weiterem wissenschaftlichen Interesse wäre hier die Entwicklung einer bezüglich ihrer Laufzeit optimierten Implementierung, da die Menge an experimentell gesammelten Daten entscheidend für die Qualität der Analyse ist.

8 Data

Die Grafiken dieser Arbeit beruhen auf den durch die Implementierung experimentell gewonnen Daten, welche in vollem Umfang bei bei Git Hub eingesehen werden können.
Zur Veranschaulichung werden hier die jeweiligen Mittelwerte der experimentellen Daten bezüglich der nach Theorem 5 geforderten Parametrisierung von $k(n)$ aufgeführt.
Die tabellarischen Werte entsprechen jeweils dem Mittelwert der Daten aus zehntausend Wiederholungen bezüglich der Auf- und Abrundung des Parameters $k(n)$ sowie der in Kapitel 6.1.3 diskutierten Gewichtung.

Mittelwerte für $k(n) = \log_2(n)/\log_2\log_2(n)$										
$\log_2(n)$	k	$\lfloor k \rfloor$	$f_{min}(n)$	$f_{rem}(n)$	$\lceil k \rceil$	$f_{min}(n)$	$f_{rem}(n)$	**Ratio**	$f_{min}(n)$	$f_{rem}(n)$
6	2.32	2	6.63	6.41	3	6.50	6.32	68 – 32	6.59	6.38
7	2.49	2	7.75	7.56	3	7.51	7.33	51 – 49	7.61	7.42
8	2.67	2	9.11	9.11	3	8.84	8.79	33 – 67	8.88	8.84
9	2.84	2	10.64	10.56	3	10.10	10.03	16 – 84	10.60	10.51
10	3.01	3	11.25	11.20	4	10.87	11.04	99 – 1	11.12	11.14
11	3.18	3	12.65	12.55	4	12.05	12.50	82 – 18	12.30	12.52
12	3.35	3	13.78	13.66	4	13.19	13.18	65 – 35	13.29	13.26
13	3.51	3	15.01	15.04	4	14.45	14.95	49 – 51	14.97	15.04
14	3.68	3	16.43	16.33	4	15.51	15.43	32 – 68	16.15	16.05
15	3.84	3	17.53	17.39	4	16.75	17.01	16 – 84	17.11	17.19
16	4	4	17.89	18.14	4	17.89	18.14	*any*	17.89	18.14
17	4.16	4	18.36	20.77	5	19.03	19.11	84 – 16	18.36	20.77
18	4.32	4	19.40	21.04	5	20.30	20.83	68 – 32	19.61	20.99
19	4.47	4	20.55	22.96	5	21.37	21.30	53 – 47	20.92	22.21
20	4.63	4	21.6	23.20	5	22.63	23.00	37 – 63	22.30	23.06

Tabelle 1: Data : RMinimum - $k(n) = \log_2(n)/\log_2\log_2(n)$

9 Code

Der in diesem Abschnitt präsentierte Teil des in dieser Arbeit verwendeten Codes soll zur besseren Gesamtverständlichkeit dieser Arbeit beitragen.

9.1 RMinimum

Anbei die Realisierung der zweiten und dritten Phase des Algorithmus RMinimum durch die Methoden **def** `rmin_phase2` und **def** `rmin_phase3`.

9.1.1 *Phase 2*

```
#   Import
import random
import queue

# ================================================
#   RMinimum: Phase 2
def rmin_phase2(L, k, cnt):

    # Generate subsets
    random.shuffle(L)
    subsets = [L[i * k:(i + 1) * k]
               for i in range((len(L) + k - 1) // k)]

    # Init M
    M = [0 for _ in range(len(subsets))]

    # Perfectly balanced tournament tree using a Queue
    for i in range(len(subsets)):
        q = queue.Queue()

        for ele in subsets[i]:
            q.put(ele)

        while q.qsize() > 1:
            a = q.get()
            b = q.get()

            if a < b:
                q.put(a)
            else:
                q.put(b)
            cnt[a] += 1
            cnt[b] += 1
        M[i] = q.get()

    return M, cnt
```

9.1.2 *Phase 3*

```python
#   Import
import random

# ================================================
#   RMinimum: Phase 3
def rmin_phase3(W, k, M, cnt):

    # Generate subsets
    random.shuffle(W)
    W_i = [W[i * k:(i + 1) * k]
                for i in range((len(W) + k - 1) // k)]
    W_i_filt = [0 for _ in range(len(W_i))]

    # Filter subsets
    for i in range(len(W_i_filt)):
        W_i_filt[i] = [elem for elem in W_i[i] if elem < M[i]]
        cnt[M[i]] += len(W_i[i])
        for elem in W_i[i]:
            cnt[elem] += 1

    # Merge subsets
    Wnew = [w for sublist in W_i_filt for w in sublist]

    return Wnew, cnt
```

9.2 RMedian

Anbei die Realisierung der zweiten Phase des Algorithmus RMedian durch die Methoden **def** rmed_phase3.

9.2.1 *Phase 2*

```
#   Import
import random

# ================================================
#   RMedian: Phase 2
def rmed_phase2(S, XS, L, C, R, cnt):

    mark = [False for _ in range(len(XS) + len(S))]
    random.shuffle(XS)
    b = len(L)

    #   Loop
    for x_i in XS:
        med = 0 #       Check if remaining or discarded
        for j in reversed(range(0, b - 1)):

            current = 2 ** 50   # Arbitrary value
            random.shuffle(L[j])
            for l in L[j]:
                if cnt[l] < current:
                    x_A = l

            if mark[x_A] == True:
                c = 1

            else:
                c = 2

            cnt[x_i] += 1
            cnt[x_A] += 1

            if x_i < x_A:
                if j + c < b:
                    mark[x_i] = True
                    L[j + c].append(x_i)
                    med = -1
                else:
                    med = -2
                break

            current2 = 2 ** 50  # Arbitrary value
```

```
            random.shuffle(R[j])
            for r in R[j]:
                if cnt[r] < current2:
                    x_B = r

            if mark[x_B] == True:
                c = 1
            else:
                c = 2

            cnt[x_i] += 1
            cnt[x_B] += 1

            if x_i > x_B:
                if j + c < b:
                    mark[x_i] = True
                    R[j + c].append(x_i)
                    med = 1
                else:
                    med = 2
                break

        #       Sort remaining elements in C
        #       and discarded in the outer buckets.
        if med == 0:
            C.append(x_i)
        elif med == -2:
            L[len(L) - 1].append(x_i)
        elif med == 2:
            R[len(R) - 1].append(x_i)

    return S, XS, L, C, R, cnt
```

9.3 *Werkzeug*

Neben den Algorithmen wurden weitere Werkzeuge zur Datenverarbeitung implementiert. Anbei beispielhaft das *Bash*-Skript, welches genutzt wurde, um mit dem *Goethe-HLR* Cluster zu kommunizieren, sowie eine *Gnuplot*-Datei [4] zur Berechnung eines Fits mithilfe des *Marquardt-Levenberg* Algorithmus [5].

9.3.1 *Server*

Bash-Skript zur Ausführung der Datei *python_py1.py* auf dem *Goethe-HLR* Cluster.

```
#!/bin/bash
#SBATCH --partition=general1
#SBATCH --nodes=1
#SBATCH --ntasks=40
#SBATCH --cpus-per-task=1
#SBATCH --mem=100000
#SBATCH --time=10:00:00
#SBATCH --mail-type=FAIL

export OMP_NUM_THREADS=1
export PYTHONHOME=/home/memhierarchy/lorenz/venv
export PYTHONPATH=/home/memhierarchy/lorenz/venv/python

# Run 10 times
/home/memhierarchy/lorenz/venv/bin/python py1.py >& 01.out &
...
/home/memhierarchy/lorenz/venv/bin/python py1.py >& 01.out &

# Wait for all child processes to terminate.
wait
```

9.3.2 *Fit*

Gnuplot-Skript zur Berechnung eines Fits bezüglich der Daten aus der Datei *filename.csv* und der Funktion $f(x) = a \cdot \log_2(x)/\log_2\log_2(x) + b$ mit Startwerten $a = 6.5$ sowie $b = -8.5$.

```
#  General
set datafile separator ",";

#  Beschriftung
#set title 'Fit : f(n) = a * eps^(-1) * log2log2(n) + b' font ",18"
set key autotitle columnhead
set xlabel 'n' font ",18"
set ylabel 'f_{min}' font ",18"
set key left top

#  Ranges
set logscale x
set xrange[64:1048576]
set yrange[0:30]
```

```
15
16 # Style
17 set style line 1 \
18     linecolor rgb '#0060ad' \
19     linetype 1 linewidth 2 \
20     pointtype 7 pointsize 1.5
21
22 set style line 2 \
23     linecolor rgb '#dd181f' \
24     linetype 1 linewidth 2 \
25     pointtype 5 pointsize 0.5
26
27 # Fit Parameters
28 FIT_LIMIT=1e-6;
29 a=6.5; b=-8.5
30
31 # Fitting Function:
32 f(x) = a * log(x, 2) / log(log(x ,2) , 2) + b
33 fit f(x) "filename" u 1:2 via a,b
34
35 # Plot
36 set logscale x 2
37 ti = sprintf("F(n) = %.2f * log2(n)/log2(log2(n)) + %.2f", a, b)
38 plot "filename.csv" using 1:2 with lines ls 2, \
39   f(x) t ti with lines ls 1
40
41 # Output
42 set term png size 800,800;
```

Literatur

[1] *Fragile complexity of some classic comparison based problems*, Afshani, Peyman and Fagerberg, Rolf and Hammer, David and Jacob, Riko and Kostitsyna, Irina and Meyer, Ulrich and Penschuck, Manuel and Sitchinava, Nodari, Institute for Computer Science, J. W. Goethe University, 60325 Frankfurt/Main, Germany, 2018 - 06 - 14

[2] *Fragile complexity of some classic comparison based problems*, Afshani, Peyman and Fagerberg, Rolf and Hammer, David and Jacob, Riko and Kostitsyna, Irina and Meyer, Ulrich and Penschuck, Manuel and Sitchinava, Nodari, Institute for Computer Science, J. W. Goethe University, 60325 Frankfurt/Main, Germany, 2019 - 01 - 09

[3] *Probabilistic recurrence relations revisited*, Shiva Chaudhuri, Devdatt Dubhashi, Theoretical computer science, Elsvier, 1997

[4] *gnuplot 5.2, An Interactive Plotting Program*, Thomas Williams & Colin Kelley, Gnuplot Documentation.
Link: http://www.gnuplot.info/docs_5.2/Gnuplot_5.2.pdf

[5] *Optimierung und inverse Probleme*, Prof. Dr. Bastian von Harrach, Goethe-Universität Frankfurt am Main Institut für Mathematik, Wintersemester 2016/17

[6] *V. Chv´atal. Lecture notes on the new AKS sorting network*, Technical Report DCS-TR-294, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1992, October

Abbildungsverzeichnis

Tabellenverzeichnis

Danksagung

Hiermit möchte ich mich in aller Form bei meiner Familie für all die Liebe und konstruktive Kritik, meinen Freunden für ihre aufbauenden Worte und den Stress vergessenden Abenden, meiner Freundin für all die aufrichtige Fürsorge und meinen Kommilitonen und Kollegen für ihre anregenden Gedanken und messerscharfen Blick beim Auffinden meiner Fehler danken.
Ohne euch wären diese Zeilen nie geschrieben worden. Ich danke euch allen aus tiefstem Herzen.