Hochschule für Technik, Wirtschaft und Kultur Leipzig (FH)
Fachbereich Informatik, Mathematik und Naturwissenschaften

Ausarbeitung zur Lehrveranstaltung Algorithm Engineering

Leipzig, Januar 2008

Vorgelegt von:	Matthias Jauernig (06INM)
Lehrveranstaltung:	Algorithm Engineering
Semester:	Wintersemester 2007/2008
Verantwortlicher Professor:	Prof. Dr. rer.nat. Karsten Weicker

Inhaltsverzeichnis

1 Allgemeine Rahmenbedingungen 1

2 Quicksort und Heapsort 2
2.1 Aufgabenbeschreibung . . . 2
2.2 Konkrete Rahmenbedingungen . . . 3
2.3 Variationen der Algorithmen . . . 3
2.4 Auswertungen . . . 5
2.4.1 Verbesserungen von Quicksort . . . 5
2.4.2 „Tuning“ des verbesserten Quicksort . . . 7
2.4.3 Direkter Vergleich der Algorithmen . . . 7
2.4.4 Künstliche Erhöhung der Vergleichszeit . . . 8

3 Prioritätswarteschlangen 10
3.1 Aufgabenbeschreibung . . . 10
3.2 Konkrete Rahmenbedingungen und Algorithmenauswahl . . . 11
3.3 Testfälle und Auswertungen . . . 12
3.3.1 Betrachtung der einzelnen Operationen . . . 12
3.3.2 Involvierte Konstanten . . . 18
3.3.3 Untersuchtes Szenario . . . 24

4 Labyrinthe 26
4.1 Aufgabenbeschreibung . . . 26
4.2 Konkrete Rahmenbedingungen . . . 26
4.3 Testfälle . . . 27
4.4 Auswertungen . . . 27
4.4.1 Dicht besetzt . . . 28
4.4.2 Mehrere Rundgänge . . . 29
4.4.3 Spärliche Mauern . . . 31
4.4.4 Variable Lockerheit . . . 32
4.4.5 Interpretation und Fazit . . . 34

5 Vertiefungsthema: Konvexe Hüllen 35
5.1 Aufgabenstellung . . . 35
5.2 Konkrete Rahmenbedingungen . . . 36
5.3 Algorithmenbeschreibungen . . . 36

5.3.1 Brute Force 36
5.3.2 Jarvis's March 37
5.3.3 Graham's Scan 38
5.3.4 Quickhull 39
5.3.5 Chan's Algorithmus 40
5.3.6 Akl-Toussaint-Heuristik 42
5.4 Testfälle und Auswertungen 43
5.4.1 Zufällig 43
5.4.2 Kreis 44
5.4.3 Quadrat 46
5.4.4 Einsatz der Akl-Toussaint-Heuristik 47
5.4.5 Fazit 48

Abbildungsverzeichnis

1 Vergleich von QSORTR mit QSORTI . . . 5
2 Vergleich von QSORTR mit QSORTRI und QSORTRIM . . . 6
3 Finden der optimalen Trenngröße m für QSORTRIM . . . 7
4 Tests von Quicksort und Heapsort bzgl. der Laufzeit . . . 8
5 Tests von Quicksort und Heapsort bzgl. der Element-Vergleiche . . . 9
6 Künstliche Erhöhung der Vergleichszeit zum Finden von Kreuzungspunkten 10
7 Laufzeiten der Operation INIT . . . 12
8 Laufzeiten der Operation INSERT . . . 13
9 Laufzeiten der Operation GETMIN . . . 14
10 Laufzeiten der Operation DELETEMIN . . . 15
11 Laufzeiten der Operation DECREASEPRIORITY . . . 16
12 Laufzeiten der Operation DELETE . . . 17
13 Konstanten bei INIT . . . 18
14 Konstanten bei INSERT . . . 19
15 Konstanten bei GETMIN . . . 19
16 Konstanten bei DELETEMIN mit Faktor $log\ n$ . . . 20
17 Konstanten bei DELETEMIN mit Faktor $(log\ n)^2$ . . . 21
18 Konstanten bei DELETEMIN mit Faktor $(log\ n)^4$ . . . 21
19 Konstanten bei DECREASEPRIORITY mit Faktoren n und $log\ n$ . . . 22
20 Konstanten bei DECREASEPRIORITY mit Faktoren $(log\ n)^{1/10}$ und $n^{1/20}$ . . 23
21 Konstanten bei DELETE . . . 23
22 Szenario-Auswertung mit Überlegenheit des Linksbaumes . . . 25
23 Szenario-Auswertung mit Überlegenheit des Binomialheaps . . . 26
24 Beispiel-Labyrinth für Lockerheit = 0 (l: Tiefensuche, r: Pledge) . . . 28
25 Laufzeiten bei variabler Labyrinthgröße, Lockerheit = 0 . . . 29
26 Beispiel-Labyrinth für Lockerheit = 5 (l: Tiefensuche, r: Pledge) . . . 30
27 Laufzeiten bei variabler Labyrinthgröße, Lockerheit = 5 . . . 30
28 Beispiel-Labyrinth für Lockerheit = 30 (l: Tiefensuche, r: Pledge) . . . 31
29 Laufzeiten bei variabler Labyrinthgröße, Lockerheit = 30 . . . 32
30 Laufzeiten bei variabler Lockerheit . . . 33
31 Anzahl besuchter Räume bei variabler Lockerheit . . . 34
32 Konvexe Hülle einer Beispiel-Punktemenge . . . 36
33 Beispiel für Jarvis's March . . . 38
34 Erste Schritte eines Beispiels zu Graham's Scan . . . 39

35 Verletzung der Konvexitätsbedingung bei Graham's Scan 39
36 Beispiel für einen Quickhull-Schritt 40
37 Beispiel für Chan's Algorithmus 42
38 Beispiel für die Akl-Toussaint-Heuristik 43
39 Zufällige Punktemenge, Brute-Force im Vergleich 43
40 Zufällige Punktemenge, Vergleich anderer Hüllalgorithmen 44
41 Kreispunkte, Explosion der Laufzeit bei Jarvis's March 45
42 Kreispunkte, Vergleich der anderen Hüllalgorithmen 45
43 Quadrat, Vergleich der Algorithmen 46
44 Zufällige Punktemenge, Vorverarbeitung mit Akl-Toussaint-Heuristik . . . 47

Tabellenverzeichnis

1 Abkürzungen für Variationen der Sortieralgorithmen 5
2 Prozentsätze der Operationen für Vorteile beim Linksbaum 24
3 Prozentsätze der Operationen für Vorteile beim Binomialheap 25

Zusammenfassung
Diese Ausarbeitung wurde zur Anerkennung als prüfungsrelevante Studienleistung im Fach *Algorithm Engineering* von Prof. Weicker an der HTWK Leipzig angefertigt. In dieser Lehrveranstaltung waren aller 2 Wochen Übungsaufgaben zu bewältigen, welche die Implementation verschiedener Algorithmen erforderten. Zudem stand deren Vergleich untereinander im Vordergrund, vor allem hinsichtlich der Laufzeit, allerdings auch andere Operationen betreffend. Drei der Aufgaben waren auszuwählen und werden hier noch einmal zusammenfassend betrachtet: *Quicksort und Heapsort* in Abschnitt 2, *Prioritätswarteschlangen* in Abschnitt 3 und *Labyrinthe* in Abschnitt 4. Weiterhin war eine vertiefende Aufgabe selbst zu wählen, wobei hier die *Berechnung konvexer Hüllen von Punktemengen in 2D* betrachtet wurde (Abschnitt 5). Die implementierten Algorithmen und Testprogramme sowie die konkreten Auswertungsdaten aller Tests stehen unter [3] zum Download bereit.

1 Allgemeine Rahmenbedingungen

An dieser Stelle sollen allgemeine Rahmenbedingungen angeführt werden, die für alle durchgeführten Tests gelten.

Als Rechnerkonfiguration wurde verwendet:

- Prozessor: AMD Athlon XP 2400+ (2000 MHz)
- Arbeitsspeicher: 768 MB DDR333 SDRAM
- Betriebssystem: Windows XP (SP2)
- für Java: JDK 1.6.0, Eclipse 3.3.1 (Europa)
- für C#: Visual Studio 2005, .Net Framework 2.0

Als Programmiersprache für die Tests in den Kapiteln 2 und 3 wurde Java verwendet, bei den Kapiteln 4 und 5 kam aufgrund der dortigen Zusammenarbeit mit meiner Projekt-Partnerin Tanja Weber (06MIM) C# zum Einsatz. Entsprechend unterschiedlich wurden auch die Zeitnahmen für die Laufzeittests realisiert. Auf Profiling wurde verzichtet, da dies bei nebenläufigen Prozessen z. B. aufgrund des Prozessor-Pipelinings nur wenig Vorteile gebracht hätte.

Bei Java kam für die Zeitnahme die Klasse `ThreadMXBean` und daraus speziell die Methode `ThreadMXBean.getCurrentThreadCpuTime()` zum Einsatz (siehe auch [8]). Mit dieser ist es möglich die vom aktuellen Thread verbrauchte CPU-Zeit in Nanosekunden abzugreifen, d. h. die Zeit welche der Thread auf dem Prozessor zugebracht hat. Dies erlaubt ein genaueres Maß als die Messung der vergangenen Systemzeit, ist aber aufgrund von Pipelining-Effekten immernoch abhängig von anderen Prozessen.

Bei C# kann die `DateTime`-Klasse verwendet werden, doch diese konnte die gestellten Anforderungen in Bezug auf Genauigkeit der Zeitmessung aufgrund der geringen Auflösung nicht erfüllen. Stattdessen wurde die Klasse `HiResTimer` von [6] verwendet, welche eine genau messende Windows-API-Funktion aus `kernel32.dll` kapselt. Statt CPU-Zeit wird hier allerdings nur die verstrichene Systemzeit gemessen.

Um den Einfluss weiterer Prozesse auf die Zeitmessung gering zu halten und bei zufälliger Erzeugung von Testdaten die Wahrscheinlichkeit für Extremfälle zu verringern, fand bei allen Tests eine Mittelung über mehrere Durchläufe statt. Die Anzahl der Durchläufe variiert dabei je nach Aufgabe und ist in dem jeweiligen Kapitel unter *Konkrete Rahmenbedingungen* aufgeführt. Wurden mehrere Algorithmen miteinander verglichen, so wurden für jeden Algorithmus dieselben Eingabedaten pro Durchlauf verwendet. Weiterhin wurde für die oft über 10 Stunden dauernden Testläufe der Rechner von allen Computernetzwerken getrennt (*standalone*-Betrieb), nicht benötigte Benutzerprogramme wurden beendet und es fanden keine Interaktionen von Benutzern mit dem Rechner statt.

2 Quicksort und Heapsort

2.1 Aufgabenbeschreibung

Von den in der Vorlesung vorgestellten Sortieralgorithmen waren Quicksort, verbessertes Quicksort und Bottom-Up-Heapsort zu implementieren und miteinander zu vergleichen. Das verbesserte Quicksort war insofern zu tunen, dass eine optimale Größe zum Umsteigen auf Insertion Sort gefunden werden sollte. Weiterhin war die Vergleichszeit zwischen 2 Elementen künstlich zu erhöhen und eine Aussage zu treffen, ab wann Bottom-Up-Heapsort den Quicksort-Algorithmen überlegen ist.

2.2 Konkrete Rahmenbedingungen

Es gelten die im Kapitel 1 dargestellten allgemeinen Rahmenbedingungen. Als Programmiersprache wurde auf Java zurück gegriffen. Pro Versuch wurden 50 Durchläufe durchgeführt, über welche die Messwerte gemittelt wurden.

Als Elementtyp für die zu sortierenden Arrays wurde `int` verwendet. Für alle Tests wurde ein Array der Länge n erzeugt, indem seine Elemente zufällig im Bereich $[0..n \cdot 10]$ generiert wurden. Demnach sind Dopplungen von Elementen möglich, fallen aber gering aus und beeinflussen die Sortieralgorithmen nicht verfälschend.

Die künstliche Erhöhung der Vergleichszeit in Abschnitt 2.4.4 wurde über eine Vergleichsmethode `compare()` für jeweils 2 Elemente realisiert, in der eine einfache Zählschleife eingebaut ist. Bei jedem Vergleich wird dabei einfach nur eine Variable i solange inkrementiert, bis die gewünschte Anzahl an Iterationen $cmpIts$ erreicht ist:

```
private int compare(int a, int b){
  if(cmpIts > 0){
    int i = 0;
    while(i < cmpIts)
      i++;
  }
  return (a<b) ? -1 : (a>b) ? 1 : 0;
}
```

Neben der Messung der Laufzeit wurde in den Tests auch die Anzahl der Element-Vergleiche gezählt. Element-Vergleiche umfassen dabei alle Vergleiche zwischen 2 Elementen des zu sortierenden Arrays, was elegant durch Modifikation der obigen `compare()`-Methode gelöst werden konnte.

2.3 Variationen der Algorithmen

Im Zusammenhang mit der Verbesserung von Quicksort stellte sich die Frage, welche Variationen des Quicksort-Algorithmus am meisten Zeitersparnis bringen und weiter als „verbessertes Quicksort“ zu verwenden waren. Einige davon wurden vom Autor bereits früher betrachtet und können unter [4] nachgelesen werden. Die folgenden Abwandlungen von Quicksort wurden hier mit einbezogen:

Quicksort rekursiv Dies ist das Standard-Quicksort. Es existiert eine rekursive Quicksort-Methode, die ein bestimmtes Teilarray zunächst partitioniert und dann rekursive Aufrufe für die dabei entstehende linke und rechte Partition durchführt.

Quicksort iterativ Es existiert weiterhin die rekursive Quicksort-Methode, welche zunächst das ihr übergebene Teilarray partitioniert. Für die dabei entstehende kleinere Partition findet ein rekursiver Aufruf statt, die größere Partition wird in einer Schleife iterativ behandelt. Sinn dabei ist, die Anzahl der Rekursionen klein zu halten und somit die Arbeit zur Stack-Verwaltung der rekursiven Aufrufe zu sparen.

Insertion Sort Grundlage dieser Variation ist die Beobachtung, dass rekursive Aufrufe in Quicksort solange stattfinden, bis die Größe des zu sortierenden Teilfeldes < 2 ist. Daher ist eine Idee die Laufzeit zu verbessern, für kleine Teilfelder der Größe m Insertion Sort aufzurufen (siehe [7], S. 155f.). Dabei ergibt sich die Frage wie groß m gewählt werden sollte, um die Laufzeit so gering wie möglich zu halten und von welchen Parametern m vielleicht abhängt.

median-of-three Als Pivot-Element für die Partitionierung wird bei Standard-Quicksort z. B. das letzte Element des betrachteten Teilarrays verwendet. Ein Problem, welches sich in der Praxis dadurch ergibt ist, dass so oftmals Teilfelder sehr kleiner Länge gebildet werden, wodurch viele rekursive Aufrufe nötig sind und der Algorithmus entsprechend länger läuft. median-of-three wählt 3 Elemente aus dem Teilarray aus (z. B. das erste, das mittlere und das letzte) und ermittelt das wertmäßig mittlere unter ihnen. Die worst-case-Laufzeitkomplexität von $O(n^2)$ bleibt zwar bestehen, allerdings steigt die Wahrscheinlichkeit ein Element zu finden, welches das Teilarray in annähernd gleich große Teile teilt.

Neben Quicksort lässt sich auch Standard-Heapsort verbessern, was auf der Beobachtung beruht, dass Elemente beim Versickern meist bis weit nach unten in die Nähe der Blattebene durchgereicht werden. Daher kann es von Vorteil sein, zunächst den maximalen Pfad von der Wurzel zur Blattebene zu bestimmen und das zu versickernde Element von unten auf dem Pfad einzufügen. Diese Algorithmus-Variation wird als Bottom-Up-Heapsort bezeichnet.

Folgende Tabelle gibt die Abkürzungen der Algorithmen-Variationen an, die im Nachfolgenden weiter verwendet werden:

Abkürzung	Algorithmus	Variation
QSORTR	Quicksort	Rekursive Variante
QSORTI	Quicksort	Iterative Variante
QSORTRI	Quicksort	Rekursiv + Kombination mit Insertion Sort
QSORTRIM	Quicksort	Rekursiv + Insertion Sort + median-of-three
HSORT	Heapsort	–
HSORTBU	Heapsort	Bottom-Up-Heapsort

Tabelle 1: Abkürzungen für Variationen der Sortieralgorithmen

2.4 Auswertungen

2.4.1 Verbesserungen von Quicksort

Entsprechend der in Abschnitt 2.3 dargestellten Variationen von Quicksort war zunächst zu ermitteln, welche Kombination die größte Verbesserung einbringt. Als erster Test wurde dabei QSORTR mit QSORTI verglichen, was in Abbildung 1 grafisch dargestellt ist.

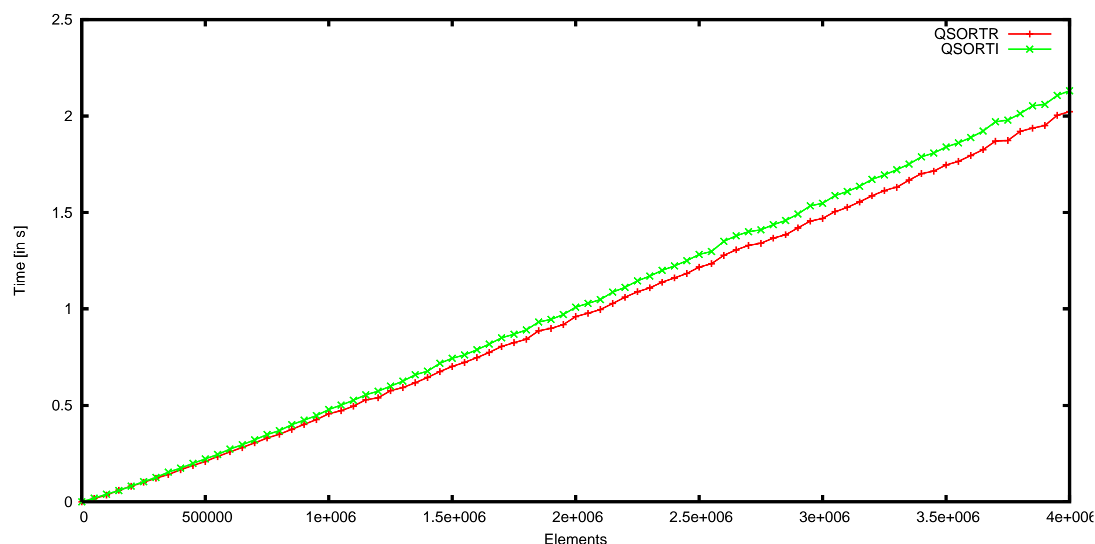

Abbildung 1: Vergleich von QSORTR mit QSORTI

Aus der grafischen Auswertung geht hervor, dass die iterative Variante QSORTI rund 5% langsamer ist als das rekursive Standard-Quicksort QSORTR. Dies widerspricht der eigentlichen Intention, durch die Elimination einer Rekursion die Laufzeit zu verkürzen. Die Theorie ist einfach: rekursive Aufrufe sind langsamer, da der Aufrufstack belegt werden

muss. Die praktischen Auswertungen konnten an dieser Stelle die theoretischen Erwartungen allerdings nicht bestätigen. Der Test wurde auf 3 unterschiedlichen Rechnern unter Windows und Linux durchgeführt, die Ergebnisse sind allerdings überall vergleichbar. Dieses Ergebnis kann hier nicht abschließend begründet werden, da es sich dem Autor jedes logischen Erklärungsversuches entzieht. Auch der Java Bytecode zeigte sich unauffällig, was den Verdacht der Compiler-Optimierung verwarf. Weiterhin werden bei der iterativen Variante zwar ein Vergleich `p<(from+to)/2` und eine Verzweigung zusätzlich benötigt, dies sollte aber nicht den Aufwand übersteigen, der für den rekursiven Aufruf benötigt wird.

Mit dem Wissen, dass das rekursive Quicksort dem iterativen überlegen ist, wurden die weiteren Test mit Kombination von Insertion Sort (mit einer Übergangsgröße zu Insertion Sort von $m = 15$) und median-of-three auf Basis von `QSORTR` gestartet. Ein Vergleich von `QSORTR` mit `QSORTRI` und `QSORTRIM` ist in Abbildung 2 zu sehen.

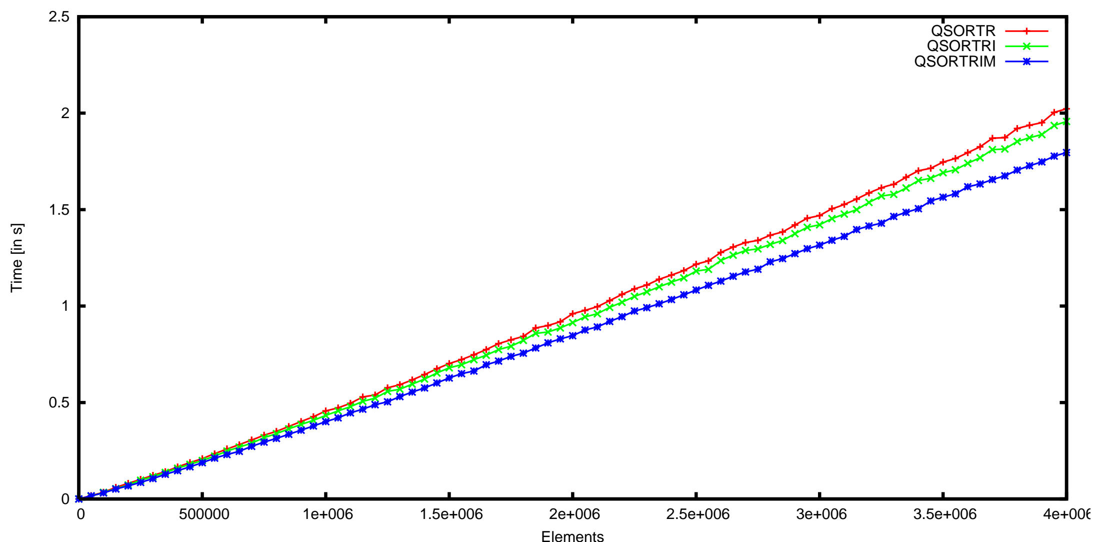

Abbildung 2: Vergleich von `QSORTR` mit `QSORTRI` und `QSORTRIM`

Die Auswertung ergibt, dass `QSORTRI` im Vergleich zu `QSORTR` über 3% weniger Laufzeit benötigt. Die besten Ergebnisse werden aber von `QSORTRIM` durch Kombination des rekursiven Quicksort mit Insertion Sort und median-of-three erreicht. Hier ergibt sich eine Laufzeitersparnis gegenüber `QSORTR` von ca. 11%. Demzufolge wird für weitere Vergleichstests `QSORTRIM` als Referenz für verbessertes Quicksort heran gezogen.

2.4.2 „Tuning“ des verbesserten Quicksort

`QSORTRIM` bezieht Insertion Sort für kleine Teilfelder mit ein. Es stellte sich dabei die Frage, wie groß die Feldgröße m gewählt werden muss, ab der von Quicksort auf Insertion Sort gewechselt wird, damit die besten Ergebnisse erreicht werden konnten. Außerdem war herauszufinden, ob m von weiteren Parametern wie der Größe des zu sortierenden Arrays abhängt.

Im folgenden Experiment wurde `QSORTRIM` für verschiedene Arraylängen durchgeführt. Dabei wurde die Laufzeit der Sortierung über steigende Trenngrößen m gemessen. Abbildung 3 zeigt das Ergebnis.

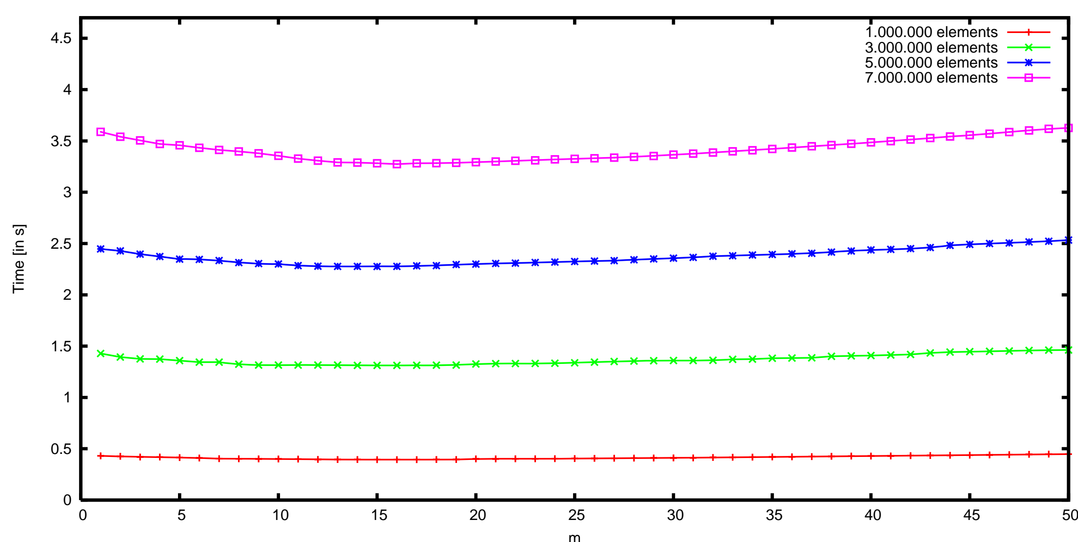

Abbildung 3: Finden der optimalen Trenngröße m für `QSORTRIM`

Der Test ergibt eine optimale Wechselgröße auf Insertion Sort im Bereich $m = [14..17]$, wobei $m = 15$ im Mittel die besten Ergebnisse hervorbingt. Die Laufzeitersparnis im Vergleich zu $m = 0$ liegt hier bei ca. 8,5%. Daher wurde $m = 15$ für weitere Tests mit `QSORTRIM` als Standard verwendet. Von der Größe des zu sortierenden Arrays hängt m den Tests folgend nicht ab, es kann daher für beliebige Eingabegrößen fest gewählt werden.

2.4.3 Direkter Vergleich der Algorithmen

Im nächsten Test wurden Quicksort, Heapsort und ihre jeweiligen Verbesserungen von der Laufzeit her betrachtet und miteinander verglichen. Abbildung 4 stellt das Ergebnis

grafisch dar.

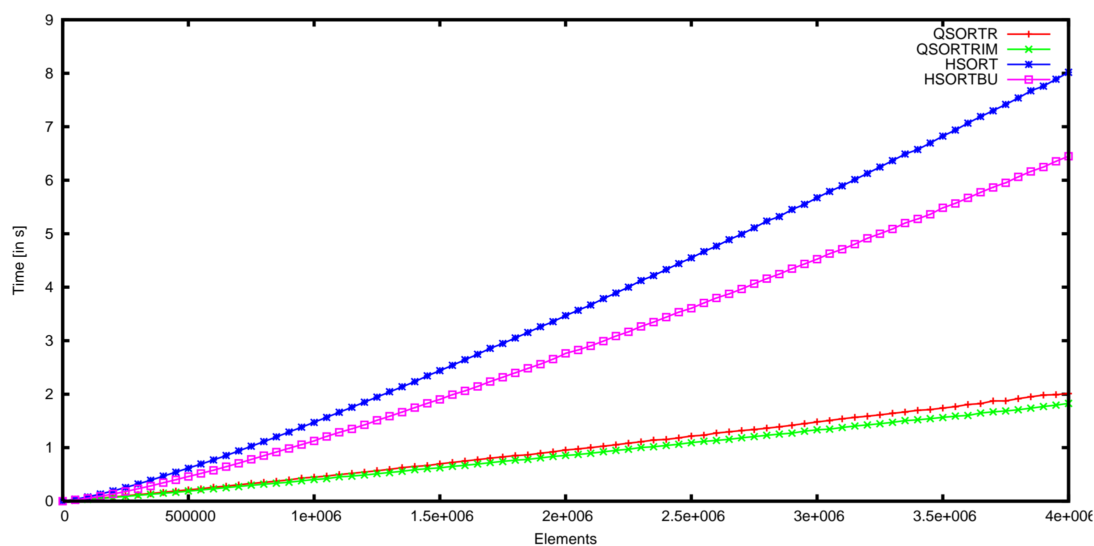

Abbildung 4: Tests von Quicksort und Heapsort bzgl. der Laufzeit

`QSORTRIM` ist der beste der betrachteten Algorithmen, im Vergleich zu `QSORTR` wird hier eine Laufzeitersparnis von ca. 11% erreicht. Interessant ist das sehr schlechte Abschneiden des normalen Heapsort `HSORT`. Gegenüber `QSORTR` wird hier fast 4 mal soviel Rechenzeit benötigt. Bottom-Up-Heapsort `HSORTBU` als Verbesserung des normalen Heapsort sortiert zwar um ca. 25% schneller als `HSORT`, ist allerdings immernoch über 3 mal langsamer als `QSORTR`.

2.4.4 Künstliche Erhöhung der Vergleichszeit

Dass die Heapsort-Algorithmen derart schlechte Ergebnisse liefern, ist nur für den Spezialfall der Verwendung von `int` als Elementtyp eine gültige Aussage. Interessant in diesem Zusammenhang ist Abbildung 5, welche die Anzahl der benötigten Vergleiche für die 4 Algorithmen darstellt.

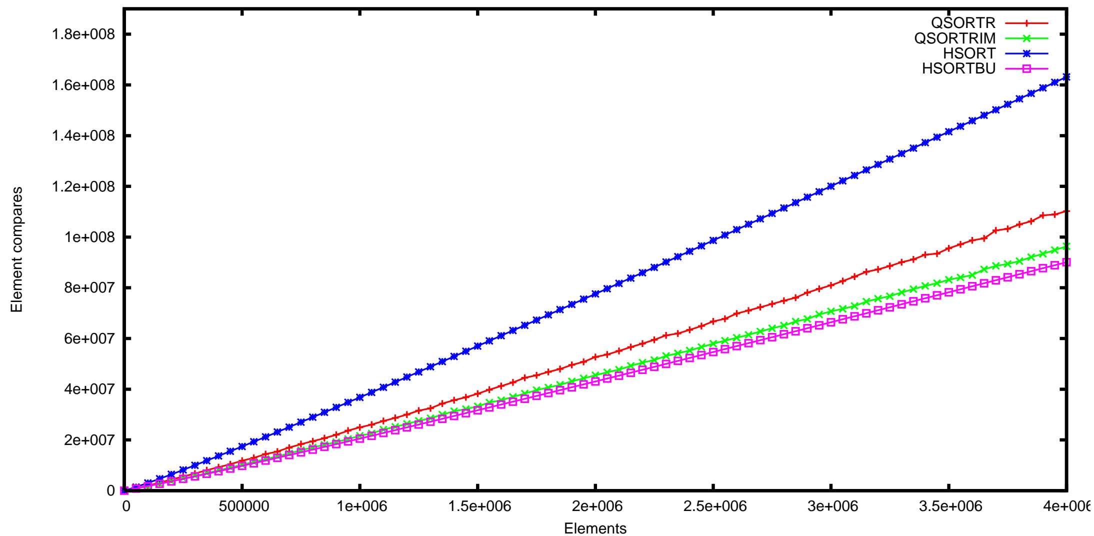

Abbildung 5: Tests von Quicksort und Heapsort bzgl. der Element-Vergleiche

Es stellt sich heraus, dass `HSORT` immernoch ca. 50% mehr Vergleichsoperationen benötigt als `QSORTR`. Den interessanteren Aspekt liefert allerdings `HSORTBU`, welches von den 4 Algorithmen die wenigsten Vergleiche aufwendet. Im Vergleich zu `QSORTR` werden hier ca. 18% weniger Vergleiche benötigt, verglichen mit `QSORTRIM` sind es immerhin noch 6,5%. Das bedeutet, dass bei einer Erhöhung der Vergleichszeit zwischen 2 Elementen Bottom-Up-Heapsort ab einem gewissen Punkt schneller ist als die betrachteten Quicksort-Varianten, was es somit für praktische Probleme interessant macht, bei denen oftmals der Vergleich zweier Elemente nicht-trivial ist.

Im nachfolgenden Experiment wurde die im Abschnitt 2.2 angegebene Methode `compare()` verwendet, um die Vergleichszeit künstlich zu erhöhen. Dabei wurde der End-Index *cmpIts* der Zählschleife sukzessive erhöht und es wurde ermittelt, wann eine Kreuzung der Laufzeiten zwischen den Quicksort-Varianten und Bottom-Up-Heapsort auftrat. D. h. es wurden bei einer Laufzeit $f(x)$ mit x=Anzahl der Iterationen die kleinsten Werte a und b ermittelt, für die gilt: $\forall x \geq a : f_{HSORTBU}(x) < f_{QSORTR}(x)$ und $\forall x \geq b : f_{HSORTBU}(x) < f_{QSORTRIM}(x)$. Als Größe der zu sortierenden Arrays wurden n=1.000.000 Elemente verwendet. Abbildung 6 zeigt die Ergebnisse.

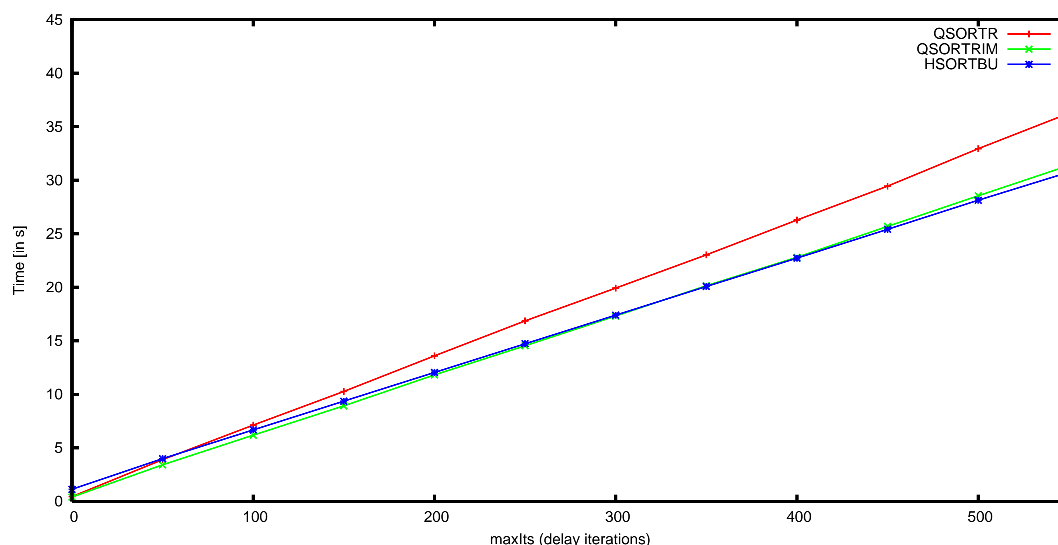

Abbildung 6: Künstliche Erhöhung der Vergleichszeit zum Finden von Kreuzungspunkten

Die Grafik zeigt einen annähernd linearen Anstieg der Laufzeiten für alle 3 Algorithmen in Abhängigkeit von der Anzahl an Iterationen $maxIts$. Die Konstante von `HSORTBU` ist dabei am kleinsten, sodass die zugehörige Gerade flacher ist als bei `QSORTR` und `QSORTRIM`. Der Kreuzungspunkt von `HSORTBU` mit `QSORTR` ergibt sich für $a = 59$ Iterationen, der Kreuzungspunkt mit `QSORTRIM` ergibt sich für $b = 328$ Iterationen. Ab dieser Anzahl von Iterationen ist Bottom-Up-Heapsort dauerhaft schneller. Für die Praxis ist dies besonders deswegen interessant, da eine einfache Zählschleife ohne weitere Operationen keinen großen Aufwand darstellt und a, b klein sind. Weitere Testfälle für konkrete Datentypen wie `string` wären hier interessant, waren aber nicht Bestandteil der hier durchgeführten Untersuchungen.

3 Prioritätswarteschlangen

3.1 Aufgabenbeschreibung

Im Rahmen dieser Aufgabe waren zwei der in der Vorlesung vorgestellten Techniken für Prioritätswarteschlangen zu implementieren und es war ein Vergleich bzgl. der einzelnen Operationen hinsichtlich der Laufzeit durchzuführen. Weiterhin waren für die einzelnen Operationen die theoretischen Laufzeiten zu bestätigen und die dabei involvierten Konstanten zu ermitteln.

Zuletzt sollte durch prozentuale Zusammensetzung der einzelnen Operationen je ein Szenario gefunden werden, für das angenommen wird, dass der eine Algorithmus dem anderen überlegen ist. Diese Annahmen waren experimentell zu bestätigen.

3.2 Konkrete Rahmenbedingungen und Algorithmenauswahl

Es gelten die im Kapitel 1 dargestellten allgemeinen Rahmenbedingungen. Als Programmiersprache wurde auf Java zurück gegriffen. Pro Versuch wurden 100 Durchläufe durchgeführt, über welche die Messwerte gemittelt wurden.

Ein Element einer Prioritätswarteschlange beinhaltet neben der ganzzahligen Priorität noch einen Schlüssel und einen Wert. Der Schlüssel ist vom Typ `int` und identifiziert das Element eindeutig, der Wert stellt den eigentlichen Inhalt eines Elements dar und wurde hier der Einfachheit halber mit dem Schlüssel gleich gesetzt. Der Schlüssel wird für die Zugriffsdatenstruktur benötigt, aus der ein Element bei bekanntem Schlüssel schnell zurück gegeben werden kann. In der vorliegenden Implementierung wurde die Java `Hashtable` als Zugriffsdatenstruktur verwendet.

Daten für die Prioritätswarteschlangen wurden in allen Tests zufällig erzeugt. Bei Warteschlangen der Länge n erhält das i. Element den Schlüssel und den Wert i, seine Priorität wird im Bereich $[1..n \cdot 10]$ zufällig gewählt. Demnach sind Dopplungen von Prioritäten verschiedener Elemente möglich, was die Algorithmen der Prioritätswarteschlangen allerdings nicht verfälschend beeinflusst. Bei $n \gg 300000$ kam es beim Einfügen aufgrund der rekursiven Aufrufe innerhalb der Algorithmen zu einem Stack Overflow, wodurch dieser Wert als Maximum für n verwendet wurde.

Bei den beiden zu implementierenden Algorithmen wurden der *Linksbaum* (*Leftist Tree*) und der *binomiale Heap* (*Binomial Heap*, siehe auch [2], S. 400ff.) verwendet, vor allem aufgrund der äquivalenten Laufzeitkomplexitäten in über der Hälfte der Operationen. Dies machte es spannend zu sehen, welcher Algorithmus bei praktischer Anwendung eine geringere Laufzeit aufweist.

3.3 Testfälle und Auswertungen

3.3.1 Betrachtung der einzelnen Operationen

Operation `INIT` Hier wurden jeweils n Elemente erzeugt und die Zeit gemessen, die benötigt wurde, um eine anfangs leere Prioritätswarteschlange mit diesen Elementen zu füllen. Abbildung 7 stellt dies grafisch dar.

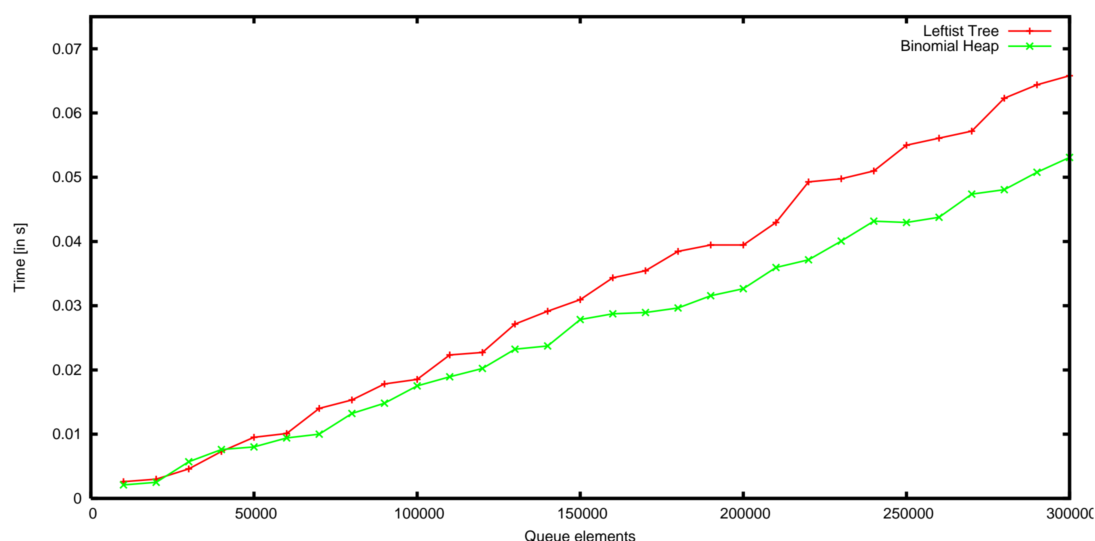

Abbildung 7: Laufzeiten der Operation `INIT`

Als Ergebnis lässt sich festhalten, dass der Binomialheap dem Linksbaum überlegen ist, auch wenn die theoretischen Komplexitäten mit $O(n \cdot log\ n)$ übereinstimmen. Im Vergleich ist die Laufzeit ca. 20% geringer. Da `INIT` bei beiden Datenstrukturen auf `MERGE` zurück geführt wird, lässt sich das Resultat mit den 2 Schritten des Linksbaumes begründen, zunächst rekursiv in den rechten Teilbaum abzusteigen, das Element einzufügen und dann in die andere Richtung die Linksbaumeigenschaft wiederherzustellen. Dabei sind viele Element-Vergleiche notwendig und da in jedem Schritt immer nur ein Element eingefügt wird, kommt es zu vielen Tausch-Operationen um die Linksbaumeigenschaft zu gewährleisten.

Operation `INSERT` In diesem Testfall wurden $m = 30000$ neue Elemente in bestehende Prioritätswarteschlangen der Größe n eingefügt. Über steigende Werte von n wurde die Zeit gemessen, die zum Einfügen der neuen Elemente benötigt wird. Das Ergebnis dieser Zeitmessung wird in Abbildung 8 dargestellt.

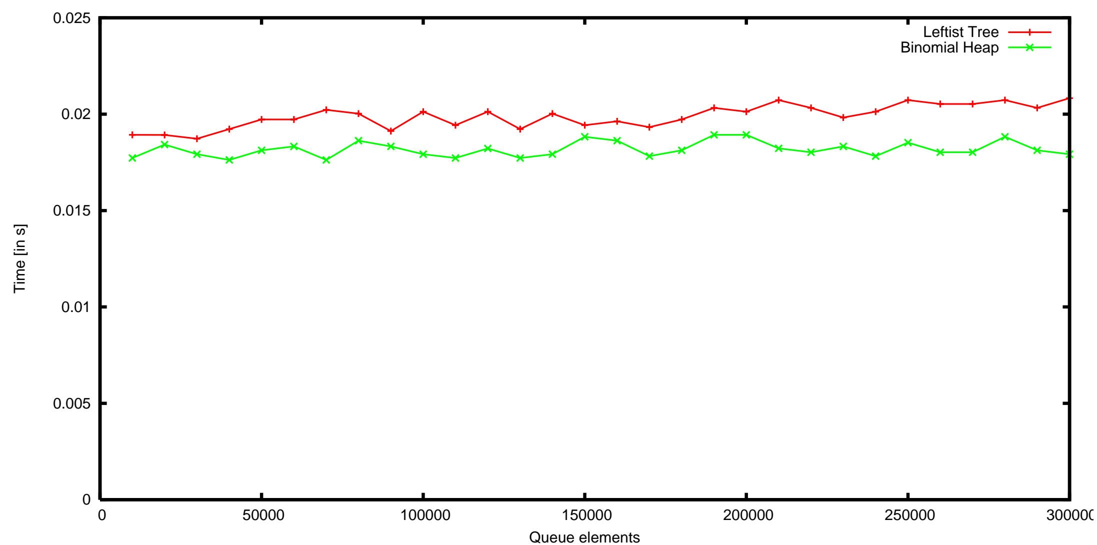

Abbildung 8: Laufzeiten der Operation `INSERT`

Die theoretische Laufzeitkomplexität von $O(log\ n)$ bedingt das schwache Ansteigen beider Kurven. Wie schon bei `INIT` schlägt auch hier der Binomialheap den Linksbaum aus denselben Gründen wie dort angeführt.

Operation `GETMIN` In diesem Testfall wurde die `GETMIN`-Operation $m = 30000$ auf bestehenden Prioritätswarteschlangen der Größe n ausgeführt. Über steigende Werte von n wurde die Zeit gemessen, die zum Erhalten des minimalen Prioritätswertes benötigt wurde. Dabei hat die `GETMIN`-Operation das „Problem", dass sie sehr schnell abläuft. Selbst bei einer Mittelung über 100 Durchläufe konnten keine stetigen Graphen erzeugt werden, wie in der Ergebnis-Ausgabe von Abbildung 9 dargestellt wird.

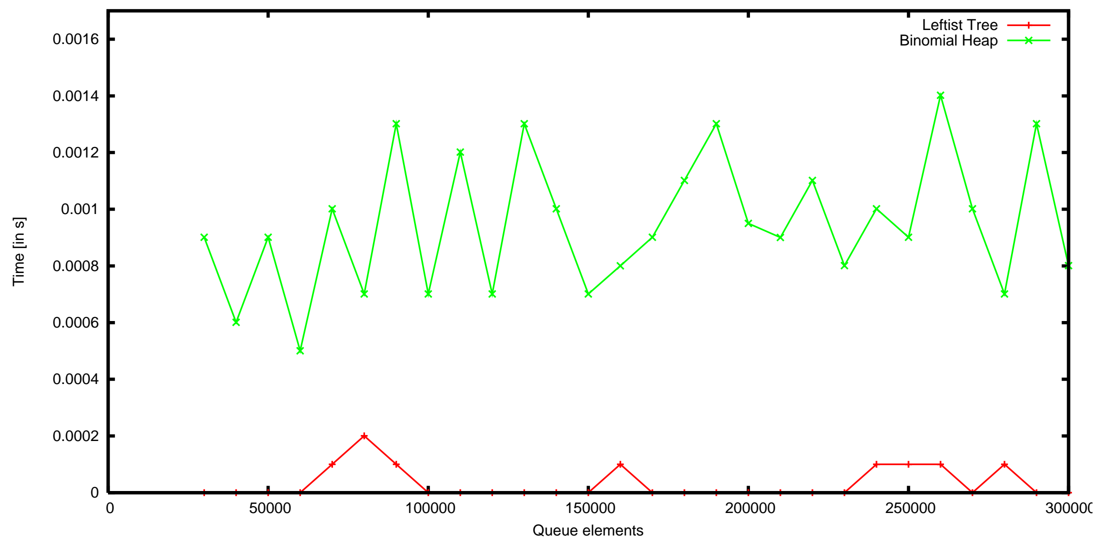

Abbildung 9: Laufzeiten der Operation `GETMIN`

Das Ergebnis entspricht den Erwartungen. Die Laufzeitkomplexität $O(1)$ des Linksbaumes wird durch die praktisch nicht vorhandene Laufzeit widergespiegelt, was dadurch bedingt ist, dass immer nur das Wurzelelement des Baumes zurück gegeben werden muss. Die Laufzeitkomplexität $O(log\ n)$ des Binomialheaps ist mit dem Durchlaufen der Topliste zu begründen und wird auch durch die gemessene Laufzeit bestätigt.

Operation `DELETEMIN` In diesem Testfall wurde die `DELETEMIN`-Operation $m = 30000$ hintereinander auf bestehenden Prioritätswarteschlangen der Größe n ausgeführt. Über steigende Werte von n wurde die Zeit gemessen, die zum Löschen der minimalen Prioritätswerte benötigt wurde. Abbildung 10 zeigt die dabei ermittelten Resultate.

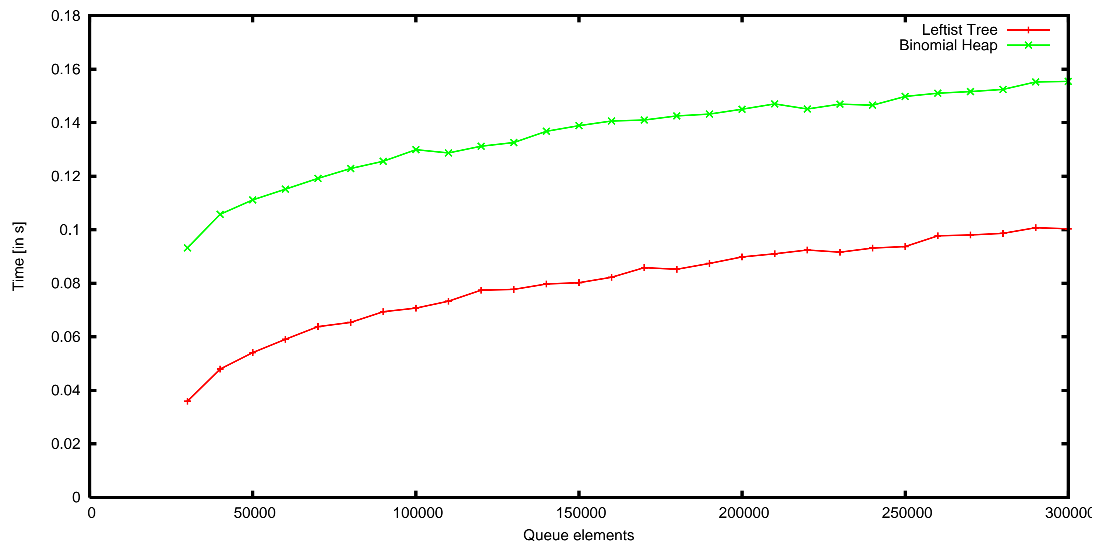

Abbildung 10: Laufzeiten der Operation `DELETEMIN`

Die Abbildung lässt die theoretische logarithmische Laufzeitkomplexität beider Algorithmen erkennen, der Linksbaum ist jedoch ca. 35% schneller als der Binomialheap. Begründet werden kann dieses Ergebnis mit der aufwändigen Toplisten-Vereinigung des Binomialheaps bei einem Merge größerer Bäume, was bei `DELETEMIN` stattfindet. Dadurch muss der ganze Heap neu geordnet werden, wohingegen beim Linksbaum nur einige Tauschoperationen notwendig sind, um die Linksbaumeigenschaft wieder herzustellen.

Operation `DECREASEPRIORITY` Hier wurde die `DECREASEPRIORITY`-Operation $m = 30000$ mal hintereinander auf zufällige Elemente bestehender Prioritätswarteschlangen der Größe n angewendet. Dabei wurden die initialen Prioritäten so weit angehoben, dass das Verringern der Priorität nicht zu einer Priorität ≤ 0 führen konnte. Über steigende Werte von n wurde die Zeit gemessen, die zum Verringern der Prioritäten der zufälligen Elemente benötigt wurde. Das Ergebnis stellt Abbildung 11 grafisch dar.

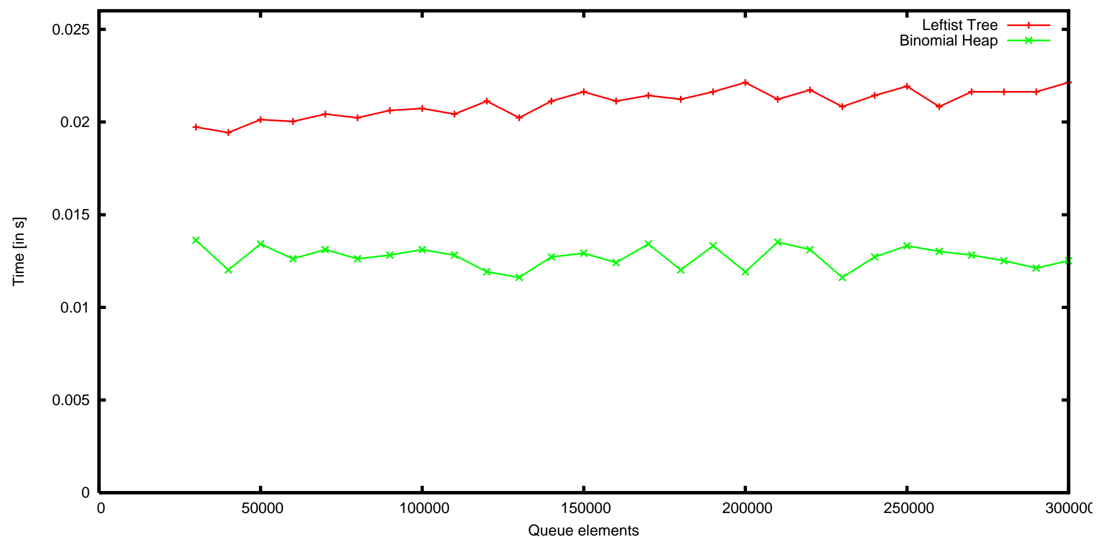

Abbildung 11: Laufzeiten der Operation `DECREASEPRIORITY`

Der Binomialheap benötigt für diesen Testfall ca. 40% weniger Laufzeit als der Linksbaum. Dies deckt sich auch mit dem erwarteten Ergebnis, da die asymptotische Laufzeit des Linksbaumes mit $O(n)$ über der Komplexität $O(log\ n)$ des Binomialheaps liegt.

Operation `DELETE` In diesem Testfall wurde die `DELETE`-Operation $m = 30000$ mal hintereinander auf zufällige Elemente bestehender Prioritätswarteschlangen der Größe n angewendet. Über steigende Werte von n wurde die Zeit gemessen, die für das Löschen der zufälligen Elemente benötigt wurde. In Abbildung 12 ist die Auswertung dieses Testfalls dargestellt.

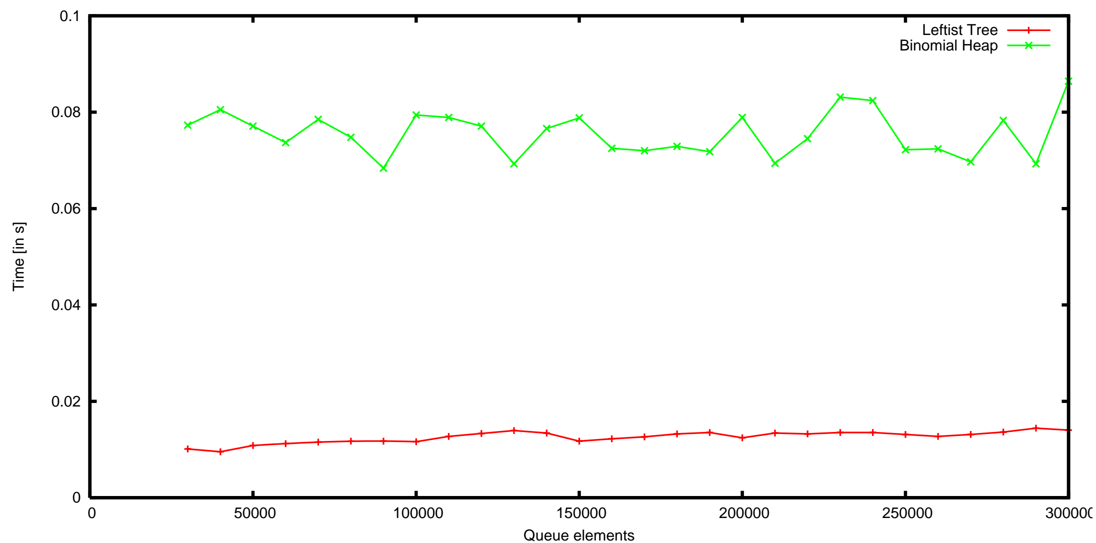

Abbildung 12: Laufzeiten der Operation `DELETE`

Die theoretischen Komplexitäten der Algorithmen betragen $O(n)$ für den Linksbaum und $O(log\ n)$ für den Binomialheap. Trotzdem liegen die praktischen Laufzeiten des Linksbaumes im betrachteten Bereich ca. 5 mal unter den Laufzeiten des binomialen Heaps. Dies zeugt von einer großen Konstante bei der Komplexität des Binomialheaps und verdeutlicht den nur bedingt praktischen Nutzen der O-Notation.

Operation `MERGE` Für die `MERGE`-Operation betragen die theoretischen Komplexitäten der beiden Algorithmen jeweils $O(log\ n)$. Bei der praktischen Zeitmessung gab es jedoch das Problem, dass keine relevanten Laufzeiten ermittelt werden konnten, da sie zu gering ausfielen. Dies war dann der Fall, wenn Operationen auf der Zugriffsdatenstruktur Hashtable ausgeschaltet wurden (nur hier wurde signifikante Laufzeit verbraucht). Es war angedacht, zufällig zwei Prioritätswarteschlangen zu erzeugen, deren Größen in der Summe $n = 300000$ ergeben. Mit diesen sollte dann ein `MERGE` durchgeführt und die Zeit dabei gemessen werden. Bei diesem einen `MERGE` konnte allerdings noch keine Laufzeit gemessen werden, da die Messung immer 0 ns hervorbrachte. Mehrere Durchläufe hatten das Problem, dass immer wieder neue Prioritätswarteschlangen hätten erzeugt werden müssen und somit die Zeitnahme unterbrochen worden wäre. $x \cdot 0$ ist immernoch 0, wodurch sich keine Verbesserung der Testergebnisse ergeben hätte. Demzufolge können für `MERGE` keine praktischen Aussagen hinsichlich der Laufzeit getroffen werden.

3.3.2 Involvierte Konstanten

Zur Ermittlung der Konstanten wurde jede Operation durch ihre theoretische Laufzeitkomplexität geteilt. Dabei wurde o. B. d. A. bei Logarithmen der log_{10} (Java: `Math.log()`) zur Berechnung verwendet. Die sich dabei ergebenden Resultate sind im Nachfolgenden dargestellt.

Operation `INIT` Beide Datenstrukturen haben bei `INIT` eine Komplexität von $O(n \cdot log\ n)$, durch das dividiert wurde. Abbildung 13 zeigt die Ergebnisse.

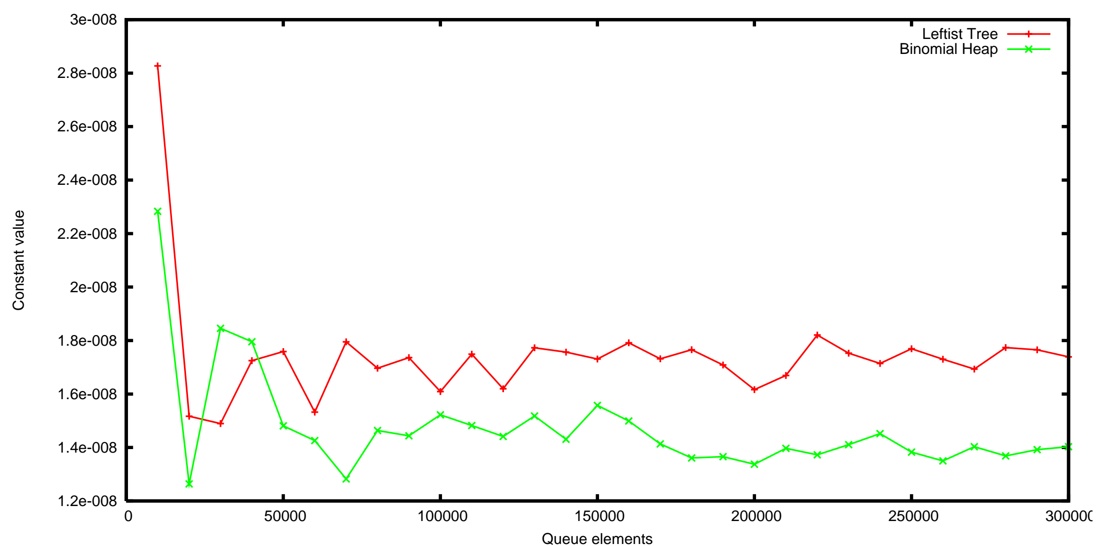

Abbildung 13: Konstanten bei `INIT`

Erkennbar ist eine annähernde Konstantheit beider Kurven (ausgenommen der erste Messwert), wenn eine Ausgleichsgerade durch die Punkte gelegt wird. Die Konstante für den Linksbaum ergibt sich als $c_1 = 1.71 \cdot 10^{-8}$ und für den Binomialheap mit $c_2 = 1.44 \cdot 10^{-8}$, was die geringere Laufzeit des Binomialheaps bestätigt.

Operation `INSERT` `INSERT` benötigt bei beiden Datenstrukturen eine Laufzeit von $O(log\ n)$. In Abbildung 14 sind die Ergebnisse der Konstantenermittlung dargestellt.

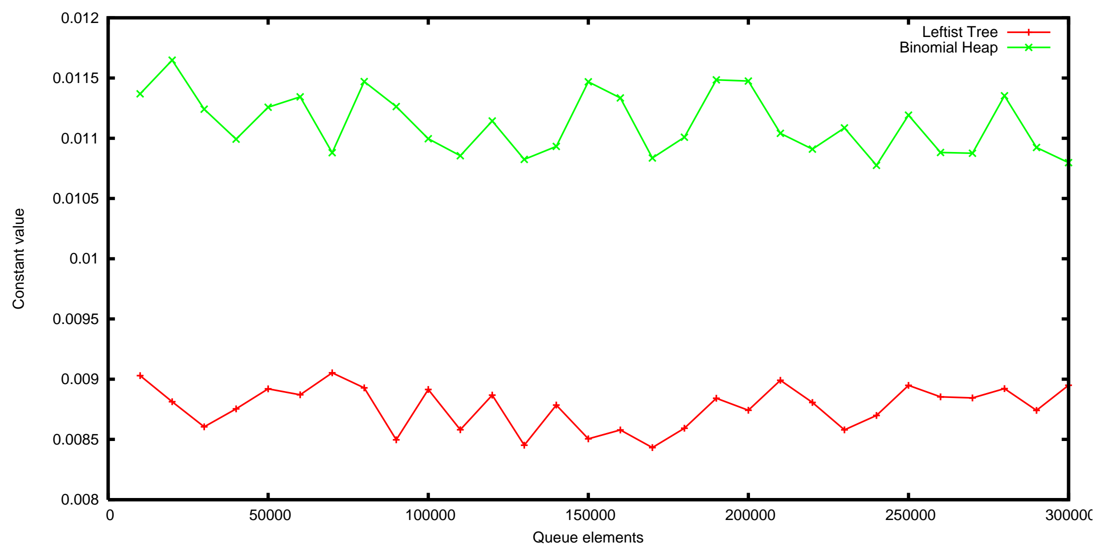

Abbildung 14: Konstanten bei `INSERT`

Auch hier ergibt sich für beide Kurven ein annähernd konstanter Wert. Die Konstante beim Linksbaum beträgt dabei nach Mittelung $c_1 = 0.00877$, beim Binomialheap ergibt sich $c_2 = 0.01112$, was wieder die bessere Laufzeit des Binomialheaps verdeutlicht.

Operation `GETMIN` Bei `GETMIN` hat der Linksbaum eine Komplexität von $O(1)$, während der Binomialheap $O(log\ n)$ benötigt. Durch diese Komplexitäten wurde dividiert, die Ergebnisse sind in Abbildung 15 dargestellt.

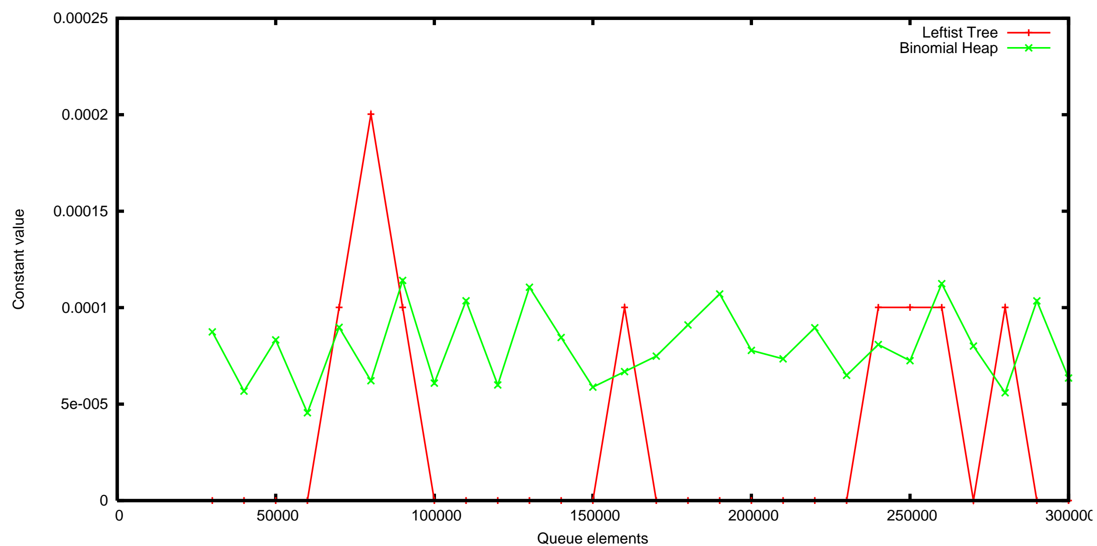

Abbildung 15: Konstanten bei `GETMIN`

Da beim Linksbaum nur unzuverlässig geringe Laufzeiten gemessen werden konnten, kann hier auch keine zuverlässige Aussage über die Konstante getroffen werden. Beim Binomialheap ergibt sich eine mittlere Konstantheit mit der Konstanten $c_2 = 7.9696 \cdot 10^{-5}$.

Operation `DELETEMIN` Zunächst wurde versucht, die theoretische Laufzeit von $O(log\ n)$ für beide Datenstrukturen zu bestätigen. Dazu wurde durch $log\ n$ dividiert und die Ergebnisse in Abbildung 16 dargestellt.

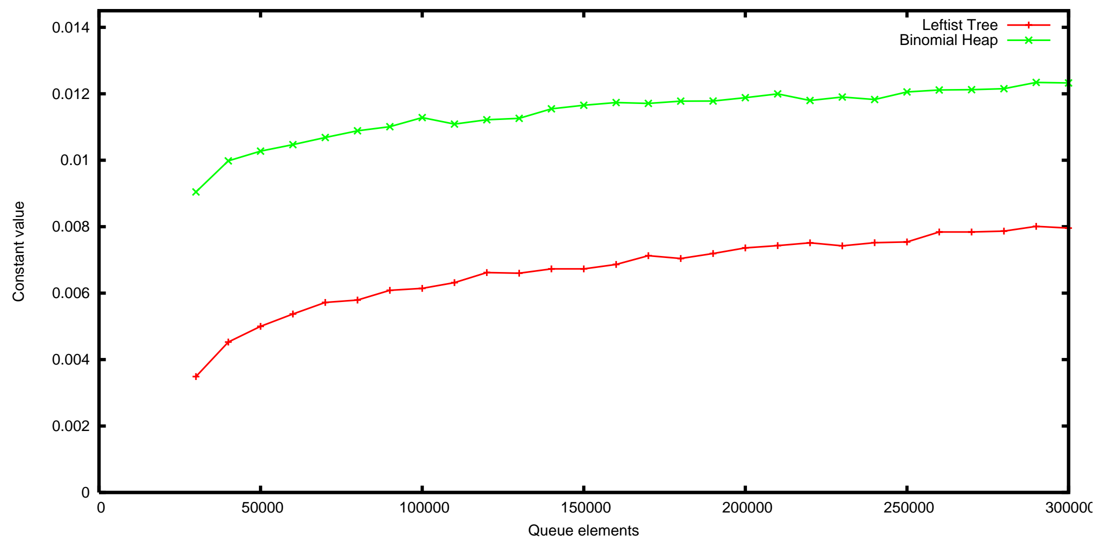

Abbildung 16: Konstanten bei `DELETEMIN` mit Faktor $log\ n$

Es ist erkennbar, dass die dargestellten Kurven keineswegs konstant sind, sondern immernoch logarithmische Form haben. Das bedeutet, dass die gemessenen Laufzeiten asymptotisch über dem $O(log\ n)$ liegen, welches von der Theorie vorgegeben wird.

Um dieses Phänomen weiter zu untersuchen wurde statt durch $log\ n$ durch $(log\ n)^i$ mit $i > 1$ dividiert, bis die Kurven annähernd konstant waren. Für den Binomialheap ergab sich dies (abgesehen von den ersten Werten) bei $i = 2$ mit einer Konstanten von $c_2 = 9.7 \cdot 10^{-4}$, für den Linksbaum bei $i = 4$ mit der Konstanten $c_1 = 6.79 \cdot 10^{-6}$, wie die Abbildungen 17 und 18 zeigen.

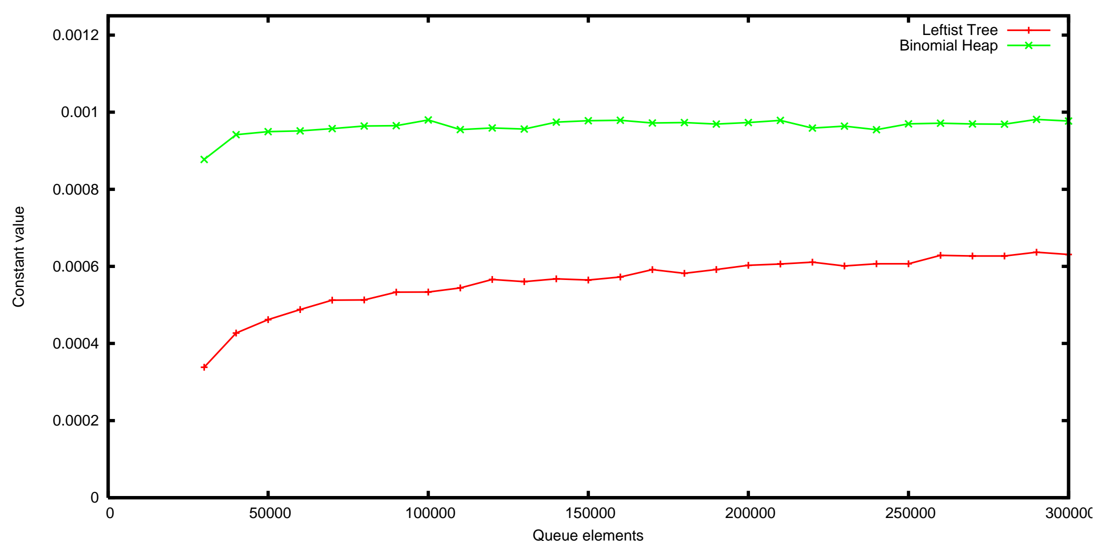

Abbildung 17: Konstanten bei DELETEMIN mit Faktor $(log\ n)^2$

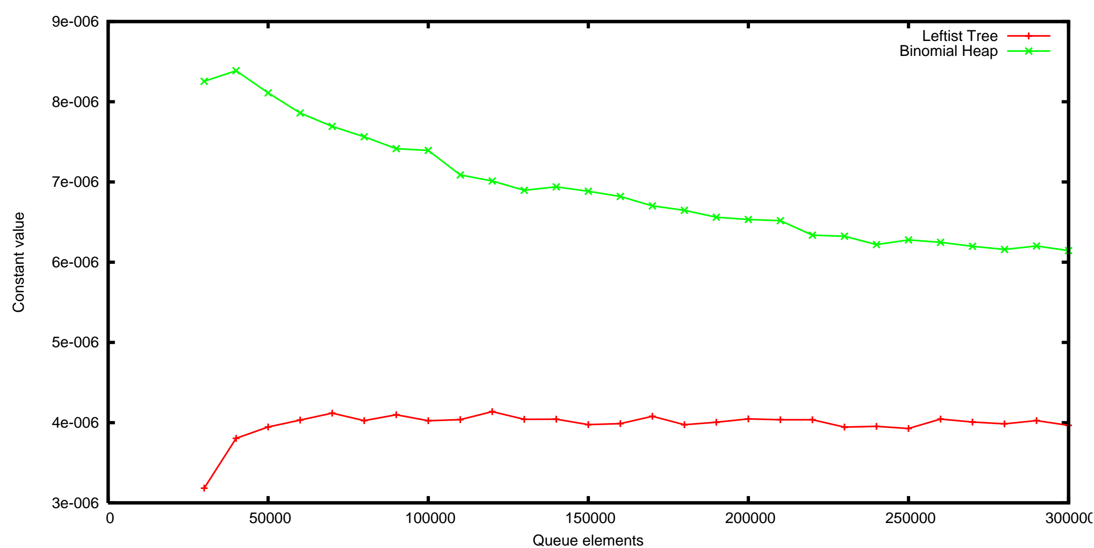

Abbildung 18: Konstanten bei DELETEMIN mit Faktor $(log\ n)^4$

Dass die Laufzeiten „über-logarithmisch“ sind, kann mit den notwendigen Operationen der Zugriffsdatenstruktur `Hashtable` erklärt werden, welche zusätzliche Zeit in Anspruch nehmen. Eine genauere Untersuchung von deren Beeinflussung wurde hier nicht durchgeführt.

Operation DECREASEPRIORITY Als theoretische Laufzeiten sind bei DECREASEPRIORITY für den Linksbaum $O(n)$ und für den Binomialheap $O(log\ n)$ angegeben. Abbildung 19

ergibt sich bei Division durch diese Terme.

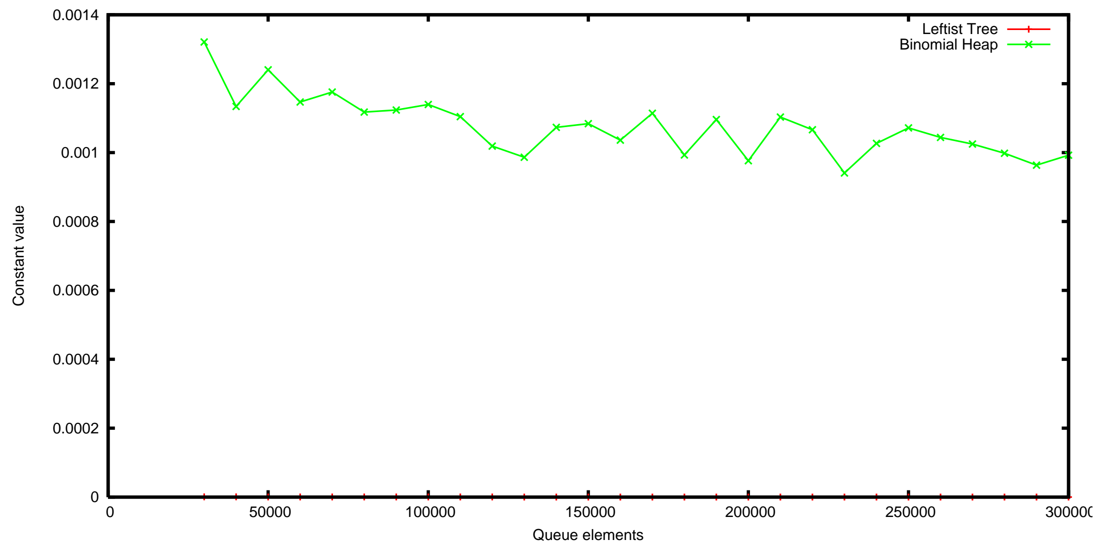

Abbildung 19: Konstanten bei `DECREASEPRIORITY` mit Faktoren n und $log\ n$

Wie bei `DELETEMIN` ist zumindest für den Binomialheap auch hier erkennbar, dass sich keine Konstante anlegen lässt. Im Gegensatz zu dieser Operation fällt die Kurve hier jedoch ab, anstatt zu steigen. Die Laufzeit ist somit bei der Messung besser gewesen als die Theorie aussagt. Für den Linksbaum ergibt sich (hier aufgrund der geringen Werte nicht zu sehen) ein ähnliches Bild, auch dessen Kurve fällt ab.

Entsprechend wurde auch hier nicht durch die eigentlichen Komplexitäten dividiert, sondern durch $n^{1/20}$ für den Linksbaum und $(log\ n)^{1/10}$ für den Binomialheap. Dadurch ergeben sich die Kurven in Abbildung 20.

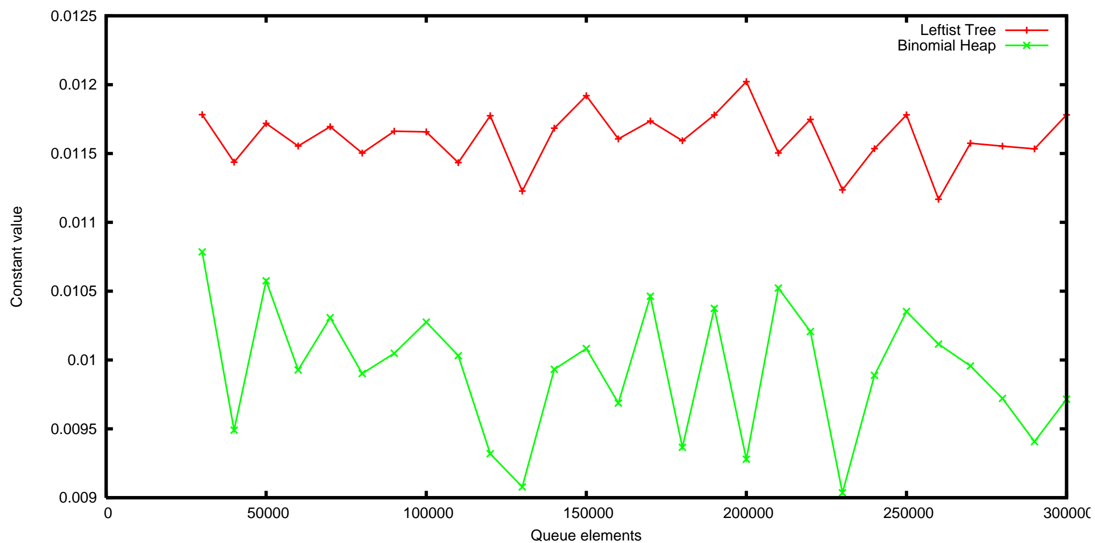

Abbildung 20: Konstanten bei `DECREASEPRIORITY` mit Faktoren $(log\ n)^{1/10}$ und $n^{1/20}$

Diese Kurven können als annähernd konstant angesehen werden mit den Konstanten $c_1 = 0.0116$ für den Linksbaum und $c_2 = 0.0099$ für den Binomialheap. Warum die Laufzeiten besser sind als von der Theorie angenommen kann an dieser Stelle nicht begründet werden, da keine logische Grundlage dafür gefunden werden konnte.

Operation `DELETE` Als theoretische Laufzeiten sind bei `DELETE` für den Linksbaum $O(n)$ und für den Binomialheap $O(log\ n)$ angegeben. Bei Division durch diese Terme ergibt sich die in Abbildung 21 dargestellte Grafik.

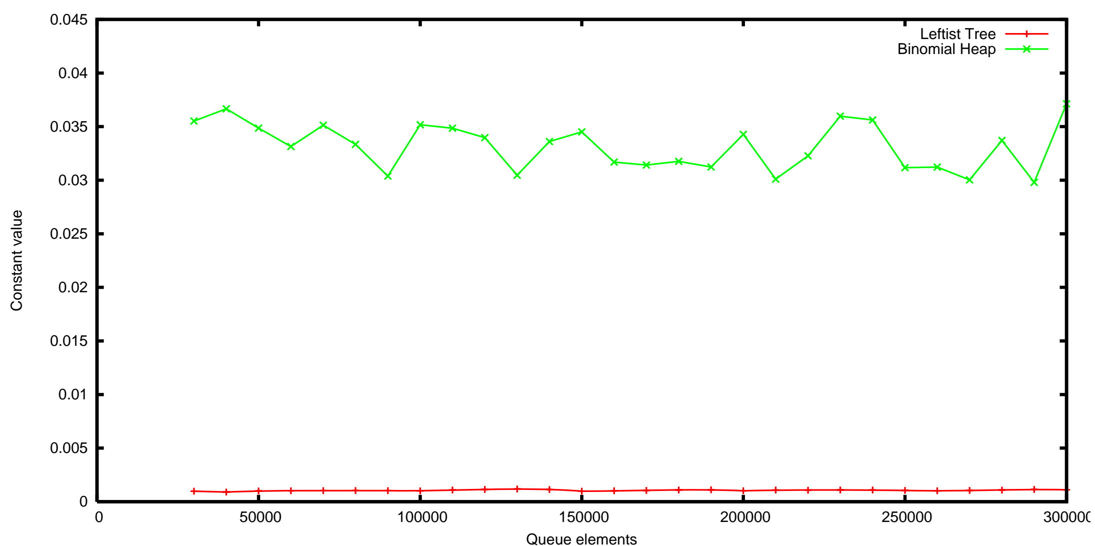

Abbildung 21: Konstanten bei `DELETE`

Die beiden Kurven sind bei Mittelung annähernd konstant mit der Konstanten $c_1 = 0.001$ für den Linksbaum und $c_2 = 0.059$ für den Binomialheap.

Operation `MERGE` Aufgrund der nicht aussagekräftigen Laufzeittests bei `MERGE` können hier keine Konstanten angegeben werden.

3.3.3 Untersuchtes Szenario

Weiterhin galt es ein Szenario zu finden, bei welchem durch unterschiedliche prozentuale Zusammensetzungen die eine Datenstruktur der anderen jeweils überlegen ist. Als Basis für dieses Szenario wurden folgende Operationen hintereinander ausgeführt `INIT` → `DELETE` → `INSERT` → `DECREASEPRIORITY` → `DELETEMIN`. Die summierte Anzahl aller Operationen betrug dabei n und die Laufzeit wurde über steigendes n betrachtet.

Szenarioteil 1 - Vorteile beim Linksbaum Für diese Untersuchung wurde folgende prozentuale Zusammensetzung der einzelnen Operationen verwendet:

Operation	Prozentsatz
`INIT`	40%
`DELETE`	25%
`INSERT`	10%
`DECREASEPRIORITY`	10%
`DELETEMIN`	15%

Tabelle 2: Prozentsätze der Operationen für Vorteile beim Linksbaum

Es wird behauptet, dass der Linksbaum dem Binomialheap überlegen ist, da er bei `DELETE` und `DELETEMIN` große Vorteile besitzt, deren Prozentsätze hoch angesetzt wurden. Abbildung 22 bestätigt diese Behauptung. Trotz der 60% an Operationen, wo der Binomialheap Vorteile hat (`INIT`, `INSERT` und `DECREASEPRIORITY`), ist der Linksbaum hier um ca. 30% schneller.

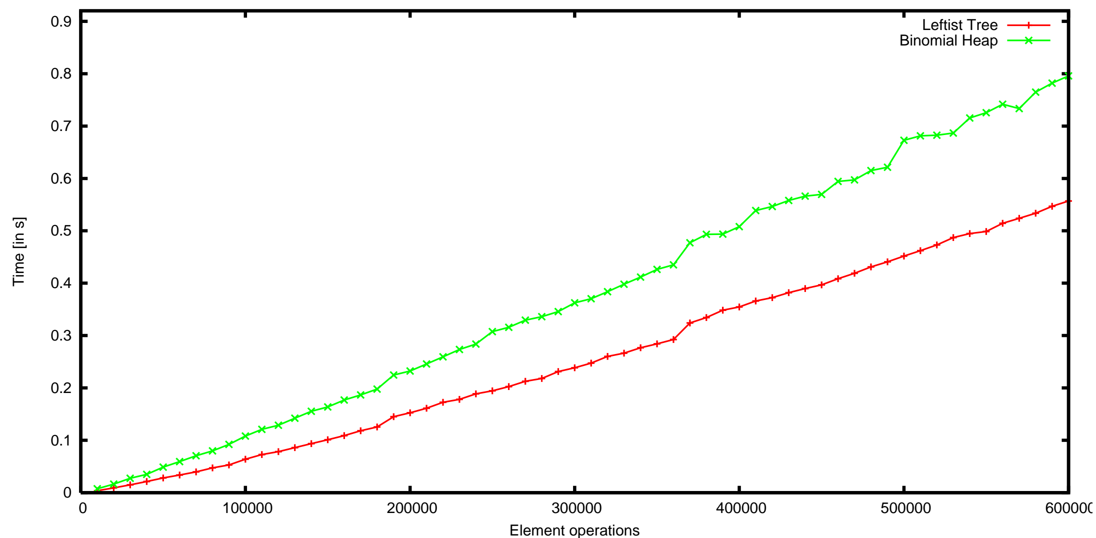

Abbildung 22: Szenario-Auswertung mit Überlegenheit des Linksbaumes

Szenarioteil 2 - Vorteile beim binomialen Heap Für diese Untersuchung wurde folgende prozentuale Zusammensetzung der einzelnen Operationen verwendet:

Operation	Prozentsatz
INIT	45%
DELETE	1%
INSERT	8%
DECREASEPRIORITY	45%
DELETEMIN	1%

Tabelle 3: Prozentsätze der Operationen für Vorteile beim Binomialheap

Es wird behauptet, dass der Binomialheap schneller ist als der Linksbaum, da er Vorteile bei `INIT`, `INSERT` und `DECREASEPRIORITY` besitzt, die hier 98% aller Operationen ausmachen. Abbildung 23 bestätigt diese Behauptung. Der Binomialheap ist hier ca. 20% schneller als der Linksbaum. Relativiert wird dieses Ergebnis dadurch, dass die Anzahl der Operationen mit Vorteilen auf Seiten des Binomialheaps mit 98% sehr hoch gewählt werden musste. Wurden ausgeglichenere Prozentsätze angesetzt, so war der Linksbaum dem Binomialheap überlegen. So lässt sich als Fazit angeben, dass die Wahl der konkreten Datenstruktur für Prioritätswarteschlangen zwar sehr wohl vom Anwendungsgebiet abhängt, im Fall von Linksbaum und Binomialheap aber in vielen Fällen der Linksbaum

verwendet werden soll, da seine Überlegenheit bei einigen Operationen stärker wiegt als die Unterlegenheit bei den anderen.

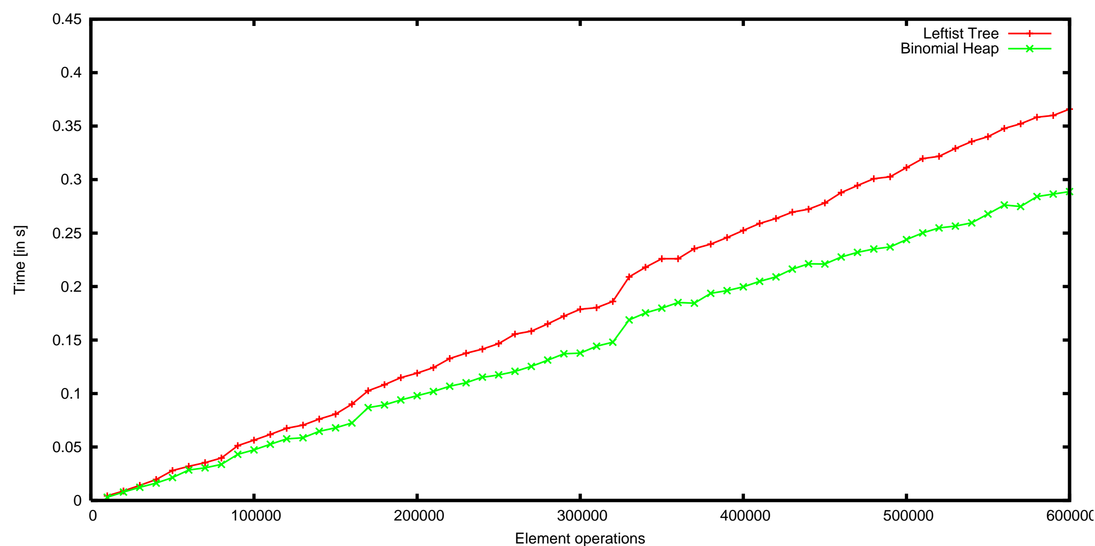

Abbildung 23: Szenario-Auswertung mit Überlegenheit des Binomialheaps

4 Labyrinthe

4.1 Aufgabenbeschreibung

Mit dem auf dem Übungsblatt angegebenen Algorithmus waren unter Zuhilfenahme der Union-Find-Struktur Labyrinthe der Größe $n \times n$ zu erzeugen. Auf diesen Labyrinthen musste nachfolgend der Weg vom Start (linke obere Ecke) zum Ziel (rechte untere Ecke) gefunden werden. Zur Wegfindung kamen die Tiefensuche und der Pledge-Algorithmus zum Einsatz, die miteinander hinsichtlich ihrer Laufzeit zu vergleichen waren. Dabei waren diese Verfahren auf 3 Arten von Labyrinthen anzuwenden, die sich in der Dichte der inneren Mauern voneinander unterschieden.

4.2 Konkrete Rahmenbedingungen

Es gelten die im Kapitel 1 dargestellten allgemeinen Rahmenbedingungen. Als Programmiersprache wurde auf C# zurück gegriffen. Pro Versuch wurden 50 Durchläufe mit unterschiedlichen Labyrinthen durchgeführt, über welche die Messwerte gemittelt wurden.

Zusätzlich wurden pro Labyrinth 10 Wiederholungen durchgeführt, wodurch sich eine Gesamt-Mittelung über 500 Messwerte ergab.

Implementiert wurde das Labyrinth als eine 2D-Matrix von „Räumen“. Jeder Raum wird dabei durch seine Position in der Matrix und 4 boolesche Variablen beschrieben die angeben, welche Wände zu den Nachbarräumen besetzt sind. Zudem wird in einem Raum eine Marke gespeichert die festhält, ob der Raum bereits betrachtet wurde oder nicht. Dies ist vor allem für die Tiefensuche notwendig um zu entscheiden, ob ein Raum noch betrachtet werden muss oder nicht.

Der Algorithmus für die Tiefensuche wurde iterativ unter Zuhilfenahme eines Stacks implementiert, zudem gelten folgende Richtungspräzedenzen (mit absteigender Priorität): Ost, Süd, West, Nord. Die Startrichtung des Pledge-Algorithmus ist Osten.

4.3 Testfälle

Die unterschiedlich dichte Besetzung der Labyrinthe mit Wänden kann dadurch erreicht werden, dass ein gewisser Prozentsatz der $2 \cdot (n-1) \cdot n$ inneren Mauern trotz eines schon existierenden Pfades vom Start zum Ziel entfernt wird. Dieser Prozentsatz wird im Folgenden „Lockerheit“ genannt. Daraus ergeben sich die folgenden Testfälle:

1. Dicht besetzt (Nur ein Weg durch das Labyrinth): Lockerheit = 0
2. Mehrere Rundgänge: Lockerheit = 5
3. Spärliche Mauern: Lockerheit = 30

Es war zu untersuchen, wie sich die Lockerheit auf die beiden Wegfindungsalgorithmen auswirkt und ob ein Algorithmus dem anderen nur bei einer bestimmten Lockerheit überlegen ist.

4.4 Auswertungen

Zunächst werden die einzelnen Auswertungen vorgestellt. Deren Interpretation findet anschließend zusammen gefasst in den Abschnitten 4.4.4 und 4.4.5 statt.

4.4.1 Dicht besetzt

Bei einer Lockerheit von 0 gibt es nur einen Weg durch das Labyrinth. Abbildung 24 zeigt einen typischen Durchlauf der Tiefensuche und des Pledge-Algorithmus. Dabei gibt olivgrün bei der Tiefensuche an, welche Räume betrachtet wurden, aber nicht zum gefundenen Pfad durch das Labyrinth gehören. Rot markierte Räume stehen bei beiden Algorithmen für „besucht".

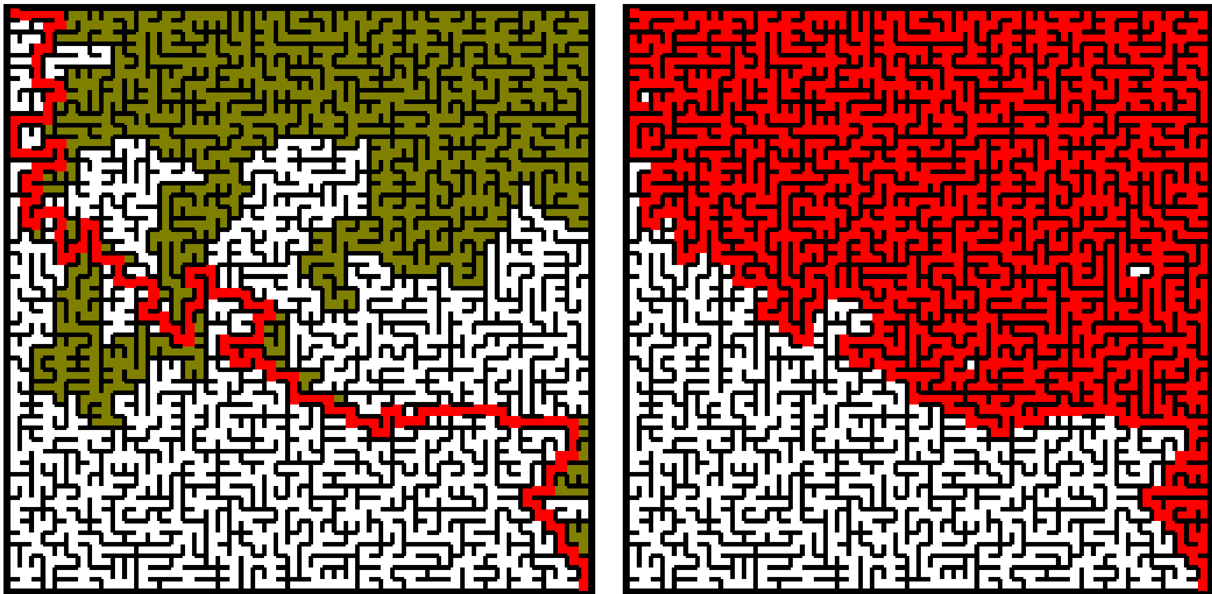

Abbildung 24: Beispiel-Labyrinth für Lockerheit = 0 (l: Tiefensuche, r: Pledge)

In Abhängigkeit von der Seitenlänge n der erzeugten $n \times n$-Labyrinthe ergab sich die in Abbildung 25 dargestellte Grafik.

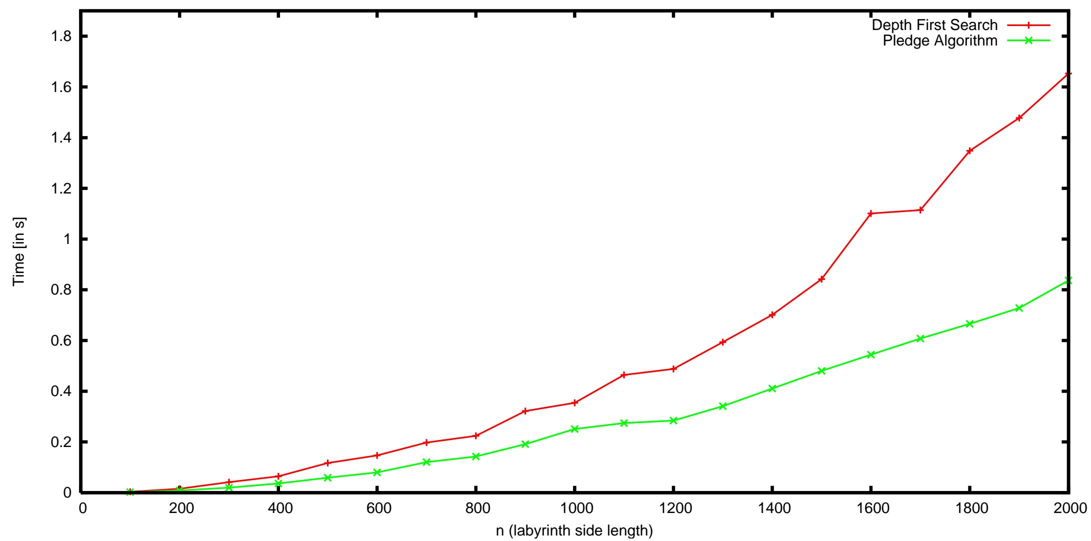

Abbildung 25: Laufzeiten bei variabler Labyrinthgröße, Lockerheit = 0

Aus der Abbildung geht hervor, dass der Pledge-Algorithmus teilweise nur die Hälfte der Zeit von der Tiefensuche benötigt und damit eindeutig überlegen ist.

4.4.2 Mehrere Rundgänge

Bei einer Lockerheit von 5 gibt es neben dem eigentlichen Pfad vom Start zum Ziel mehrere Rundgänge durch das Labyrinth. Ein typisches Beispiel-Labyrinth und die Abarbeitung von Tiefensuche und Pledge-Algorithmus wird durch Abbildung 26 gegeben.

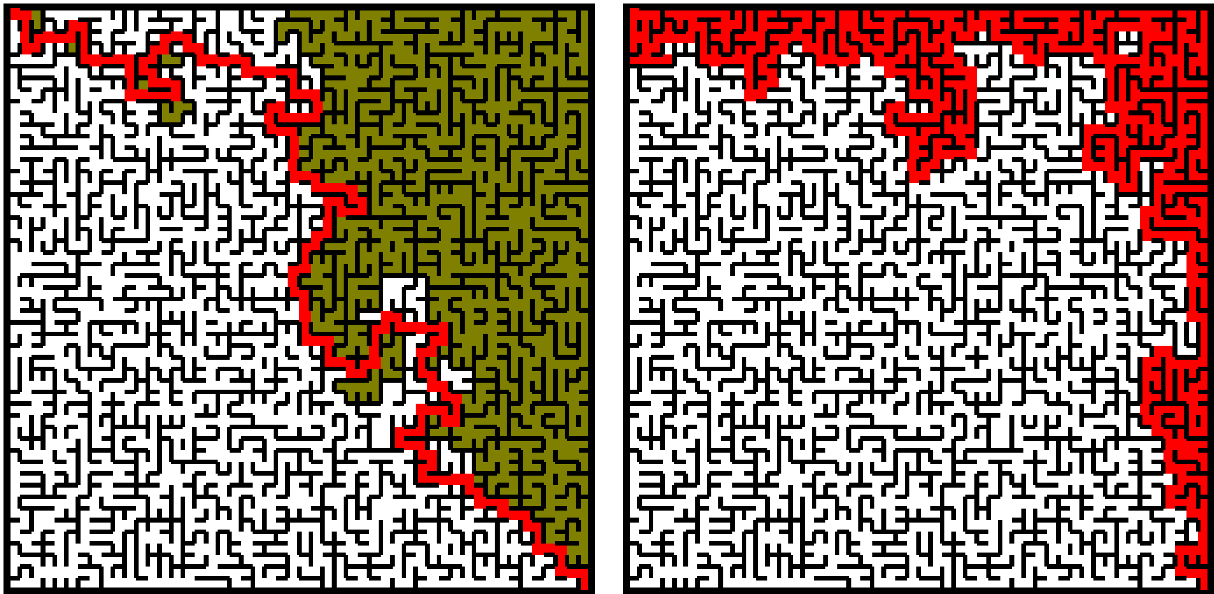

Abbildung 26: Beispiel-Labyrinth für Lockerheit = 5 (l: Tiefensuche, r: Pledge)

In Abhängigkeit von der Seitenlänge n wurde wieder die Laufzeit gemessen und es ergab sich die in Abbildung 27 dargestellte Grafik.

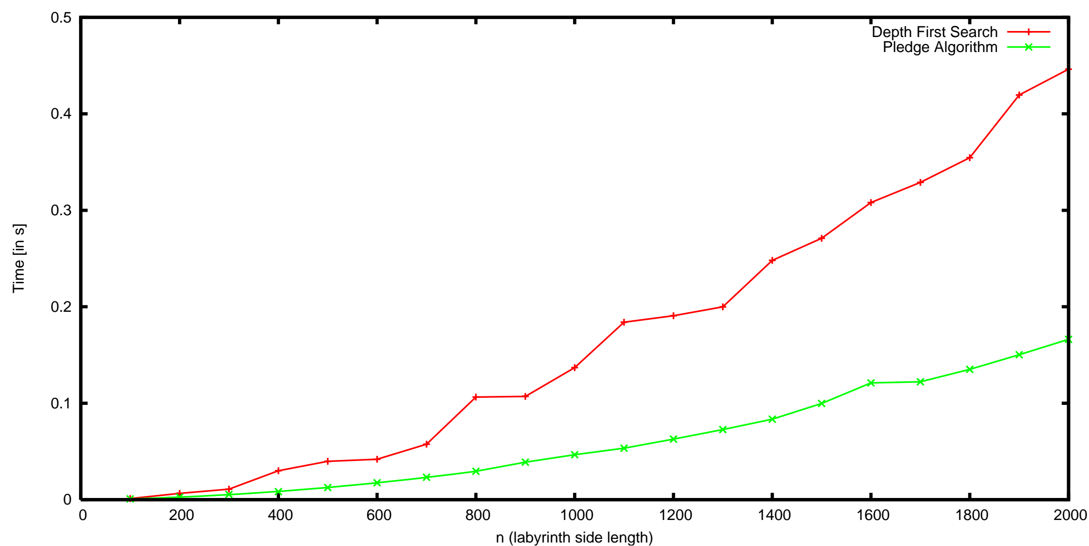

Abbildung 27: Laufzeiten bei variabler Labyrinthgröße, Lockerheit = 5

Aus der Abbildung geht hervor, dass der Pledge-Algorithmus hier sogar weniger als die Hälfte der Zeit von der Tiefensuche benötigt und damit noch bessere Ergebnisse liefert

als bei einer Lockerheit von 0. Weiterhin sind beide Algorithmen nun ca. 4 mal schneller, da der Weg zum Ziel durch die zusätzlichen Rundgänge schneller gefunden werden kann.

4.4.3 Spärliche Mauern

Eine Lockerheit von 30 führt zu spärlich besetzten Labyrinthen, in denen bereits kleine „Lichtungen“ existieren und wo es viele Wege vom Start zum Ziel gibt. Abbildung 26 zeigt ein Beispiel-Labyrinth und die Wegfindung der beiden Algorithmen.

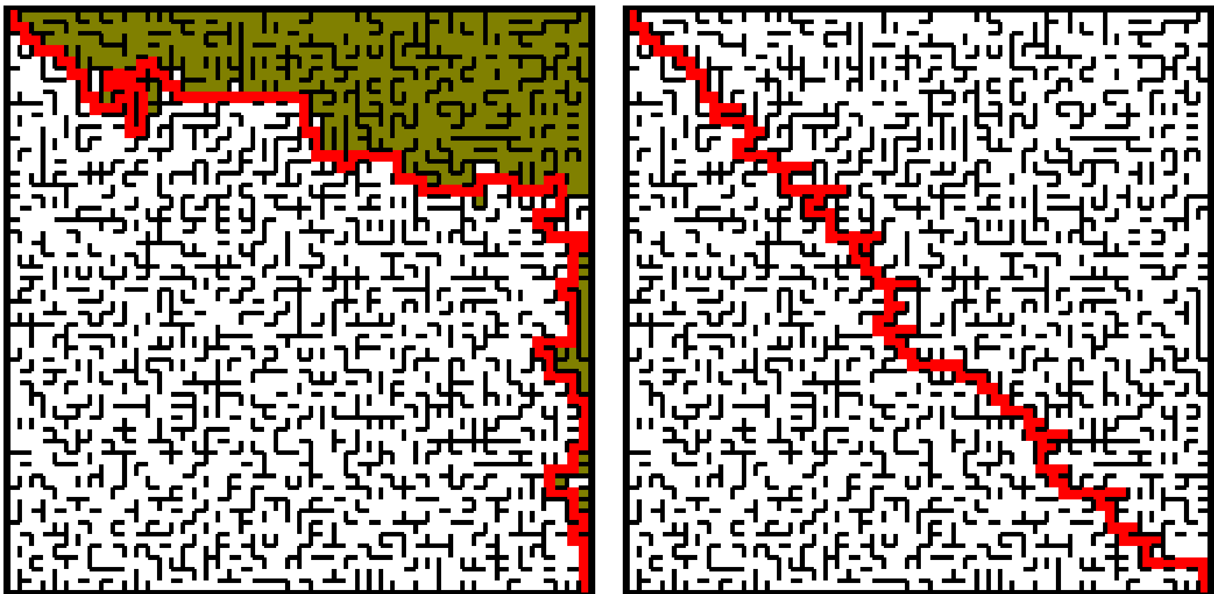

Abbildung 28: Beispiel-Labyrinth für Lockerheit = 30 (l: Tiefensuche, r: Pledge)

Die Laufzeit wurde wiederum in Abhängigkeit von der Seitenlänge n gemessen und im Diagramm in Abbildung 29 für die beiden Algorithmen aufgetragen.

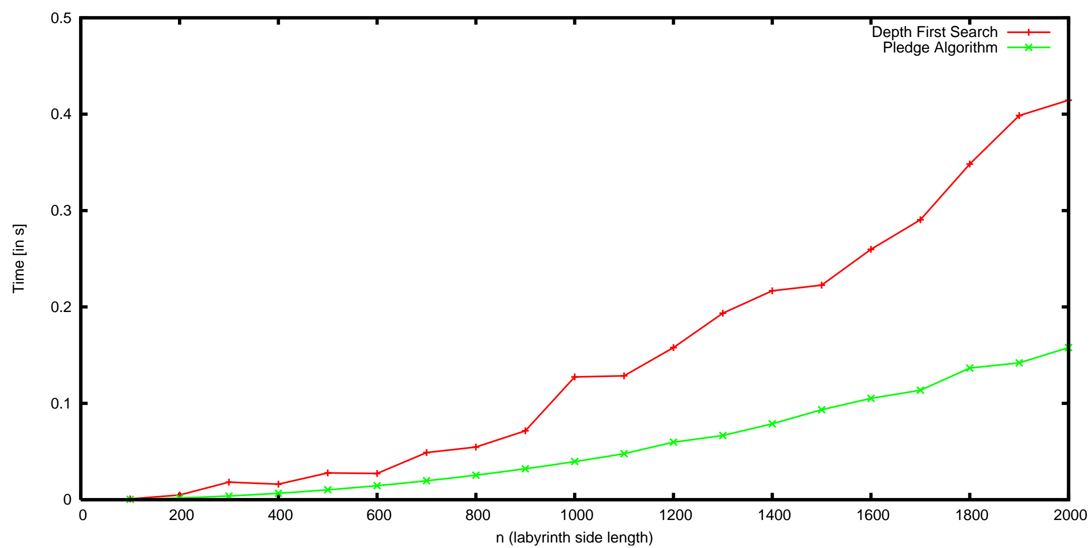

Abbildung 29: Laufzeiten bei variabler Labyrinthgröße, Lockerheit = 30

Verglichen mit einer Lockerheit von 5 ergeben sich hier nur geringe Unterschiede die Laufzeit betreffend. Der Pledge-Algorithmus ist der Tiefensuche weiterhin überlegen.

4.4.4 Variable Lockerheit

Als weiterer interessanter Testfall wurde eine feste Labyrinthseitenlänge von $n = 1000$ angenommen und die Lockerheit variabel gehalten. In dieser Kosntellation wurden die Laufzeiten beider Algorithmen gemessen. Das Ergebnis dieses Versuchs ist in Abbildung 30 dargestellt.

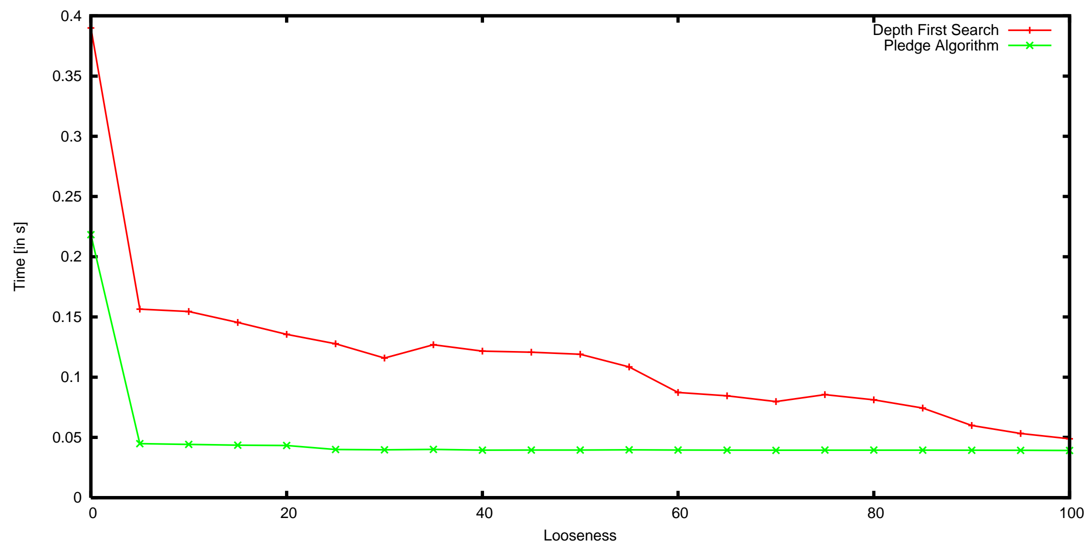

Abbildung 30: Laufzeiten bei variabler Lockerheit

Es ist offensichtlich, dass der Pledge-Algorithmus stets weniger Zeit benötigt als die Tiefensuche, unabhängig von der verwendeten Lockerheit der zugrunde liegenden Labyrinthe. Dabei ist die Laufzeit von Pledge ab einer Lockerheit von 5 nahezu konstant bzw. nimmt nur noch geringfügig ab. Die Differenz in der Laufzeit zur Tiefensuche ist an dieser Stelle auch am größten und wird geringer für steigende Lockerheit. Das starke Abfallen der Laufzeit von Lockerheit = 0 zu Lockerheit = 5 erklärt sich direkt durch die hinzukommenden Rundgänge. Dadurch werden neue Wege vom Start zum Ziel angelegt und die Algorithmen können schnell einen Pfad durch das Labyrinth finden. Größere Lockerheiten haben hier einen deutlich geringeren Einfluss.

Interessant in diesem Zusammenhang ist auch, wie viele Räume von den beiden Algorithmen eigentlich besucht werden müssen, bis das Ziel erreicht ist. Abbildung 31 stellt diesen Zusammenhang dar.

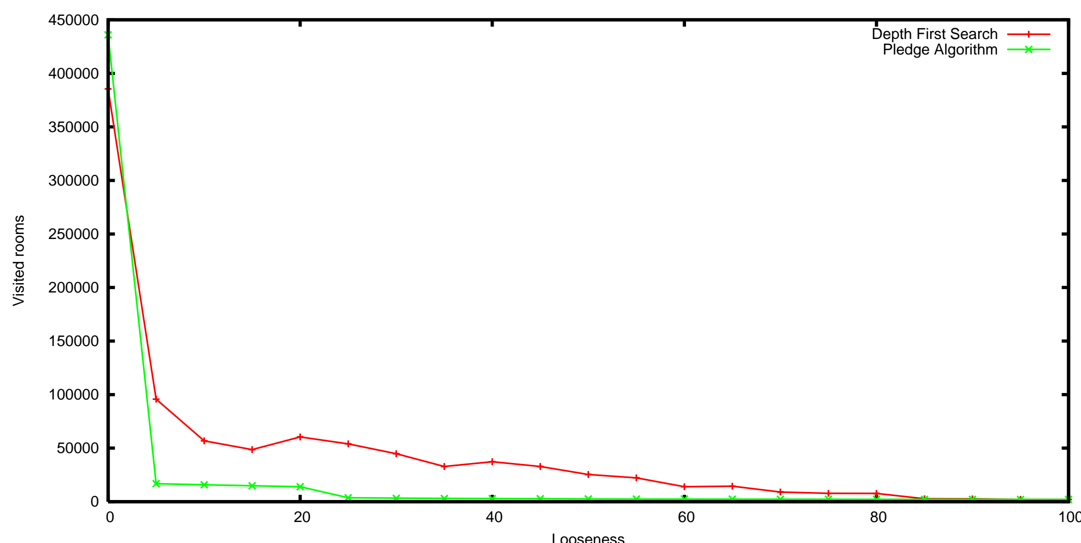

Abbildung 31: Anzahl besuchter Räume bei variabler Lockerheit

Hier ist erkennbar, dass der Pledge-Algorithmus für eine Lockerheit von 0 mehr Räume betrachtet als die Tiefensuche. Trotzdem ist die Laufzeit von Pledge bereits hier geringer, da weniger Vergleichs-Operationen in einem Raum durchgeführt werden müssen. Beim Pledge-Algorithmus werden anders als bei der Tiefensuche keine „Brotkrumen“ ausgestreut, nach denen erst immer geschaut werden muss, wenn man einen Raum betritt. Zudem findet kein Backtracking statt, wodurch bereits einmal besuchte Räume wiederholt besucht werden, wenn noch kein Weg gefunden wurde. Für größere Lockerheiten muss der Pledge-Algorithmus deutlich weniger Räume betrachten, um zum Ziel zu kommen, wobei die Differenz zur Tiefensuche stetig abnimmt.

4.4.5 Interpretation und Fazit

Bei den betrachteten Testfällen mussten nur für eine Lockerheit von 0 beim Pledge-Algorithmus mehr Räume betrachtet werden als bei der Tiefensuche. In diesem Fall folgt Pledge langen Wandketten und benötigt dementsprechend viel Zeit, um den Weg zu finden. Bei der Tiefensuche ist dies nicht der Fall und mit dem dort stattfindenden Backtracking wird das Ziel mit weniger zu betrachtenden Räumen gefunden. Die Laufzeit von Pledge ist jedoch auch hier schon geringer, da das Backtracking viele Vergleichsoperationen mit sich bringt, die beim Pledge-Algorithmus mit seiner einfachen Arithmetik entfallen.

Bereits bei einer Wegnahme von 5% der inneren Mauern (Lockerheit = 5) und weiter für größere Lockerheiten ist der Pledge-Algorithmus der Tiefensuche meist deutlich überlegen, wobei die Differenz zwischen beiden für hohe Lockerheiten wieder abnimmt. Werden neben dem eigentlichen Weg von Start zu Ziel mit einer Lockerheit > 0 auch noch weitere Wege ermöglicht, so findet Pledge mit seiner Wandverfolgung schnell durch das Labyrinth. Die Tiefensuche ist in diesem Fall mit den statischen Präzedenzen zu starr und es werden im Backtracking zu viele unnötige Wege betrachtet.

Neben der geringeren Anzahl zu besuchender Räume hat der Pledge-Algorithmus gegenüber der Tiefensuche weitere praktische Vorteile. Steht man in einem Labyrinth und sucht den Ausgang, so muss man bei der Tiefensuche in jedem Raum einen Brotkrumen legen um zu erkennen, in welchen Räumen man bereits war und so vor allem Rundgänge zu vermeiden. Dumm nur, wenn Vögel die Brotkrumen stehlen oder man die Brotkrumen mangels Licht nicht sieht. Den Pledge-Algorithmus kann man auch in der dunkelsten Höhle anwenden, benötigt keine Brotkrumen und findet in endlicher Zeit den Ausgang.

5 Vertiefungsthema: Konvexe Hüllen

5.1 Aufgabenstellung

Als selbst gewähltes Vertiefungsthema sollen verschiedene Algorithmen zur Berechnung der konvexen Hülle einer Punktemenge in 2D miteinander bzgl. der Laufzeit verglichen werden. Dabei sind verschiedene Testfälle zu untersuchen für die anhand ihrer Laufzeitkomplexität angenommen wird, dass einige Algorithmen besser arbeiten als andere.

Die konvexe Hülle $conv(X)$ einer Punktemenge X ist definiert als kleinste konvexe Menge, die X vollständig enthält. Bezeichnet $conv_i(X)$ mit $i \in [1 \ldots h]$ und $h = |conv(X)|$ den i-ten Punkt einer konvexen Hülle entgegen des Uhrzeigersinns, so gilt:
$\forall x \in X : x$ liegt links von oder auf $\overline{conv_i(X)\ conv_{i+1}(X)}$, wobei $conv_{h+1}(X) = conv_1(X)$. Anschaulich lässt sich X mit einem Nagelbrett vergleichen. Wird ein Gummi über alle Nägel gespannt, so bilden die Nägel welche der Gummi berührt die konvexe Hülle der Nägelmenge. Abbildung 32 zeigt beispielhaft die konvexe Hülle einer Punktemenge.

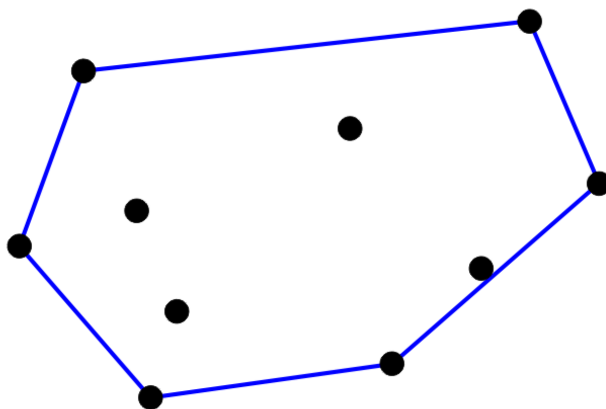

Abbildung 32: Konvexe Hülle einer Beispiel-Punktemenge

5.2 Konkrete Rahmenbedingungen

Es gelten die im Kapitel 1 dargestellten allgemeinen Rahmenbedingungen. Als Programmiersprache wurde auf C# zurück gegriffen. Pro Versuch wurden 50 Durchläufe durchgeführt, über welche die Messwerte gemittelt wurden.

Wurden Punkte erzeugt, so lagen diese stets im Bereich $\left[0 \ldots \sqrt{\texttt{Int32.MaxValue}}\,\right]$, um Integer-Überläufe bei den Berechnungen zu vermeiden.

5.3 Algorithmenbeschreibungen

Bevor auf die Auswertungen eingegangen wird, werden in diesem Abschnitt die einzelnen Algorithmen kurz vorgestellt, die implementiert und miteinander verglichen wurden. Diese Auswahl deckt dabei eine hohe Bandbreite an unterschiedlichen Laufzeitkomplexitäten ab, was den Vergleich der einzelnen Verfahren interessant macht. Die Anzahl von Algorithmen zur Berechnung konvexer Hüllen ist dabei sehr groß und reicht von inkrementellen Verfahren über divide-and-conquer-Ansätze bis hin zu prune-and-search-Algorithmen (siehe [2], S. 898 ff.). Nach eigenen Recherchen des Autors gibt es ca. 20 „bekannte“ Hüll-Algorithmen, die tatsächliche Anzahl dürfte jedoch noch höher liegen.

5.3.1 Brute Force

Ein Brute-Force-Ansatz zum Finden der konvexe Hülle $conv(X)$ einer Punktemenge X ist folgender: für jedes mögliche Paar (x_1, x_2) zweier Punkte aus X wird geprüft, ob alle

Punkte aus X auf der linken Seite von oder auf $\overline{x_1\ x_2}$ liegen. Ist dies der Fall, so sind die beiden Punkte Teil von $conv(X)$.

Zum Durchlaufen aller möglichen Paare zweier Punkte wird $O(n^2)$ benötigt, die Prüfung ob alle Punkte nicht auf der rechten Seite liegen bringt einen weiteren Faktor $O(n)$ mit ein. Die Laufzeitkomplexität dieses Ansatzes beträgt somit $O(n^3)$.

5.3.2 Jarvis's March

Dieser Algorithmus wurde 1973 von R. A. Jarvis veröffentlicht. Er wird in der Literatur (siehe z. B. [2], S. 901 ff.) auch oft als „gift wrapping algorithm" bezeichnet und das zu Recht. Es handelt sich um einen inkrementellen Algorithmus, bei dem in jedem Schritt ein weiterer Hüllpunkt ermittelt wird. Zunächst wird der Punkt mit der kleinsten x-Koordinate ermittelt, wofür eine Laufzeit von $O(n)$ benötigt wird. Dieser Punkt P_1 ist der erste Punkt der konvexen Hülle. Von P_1 aus wird entgegen des Uhrzeigersinns der nächste Hüllpunkt P_2 gesucht (der am weitesten „rechts" liegende, d. h. es gibt keinen Punkt der rechts von $\overline{P_1\ P_2}$ liegt), was in $O(n)$ durchgeführt werden kann. Diese Prozedur wird wiederholt aufgerufen, bis wieder P_1 erreicht wird, was die Hülle abschließt. Der ganze Ablauf ähnelt dem Verpacken eines Geschenks, was dem Algorithmus seinen beiläufigen Namen gab. Abbildung 33 veranschaulicht den Vorgang für die ersten 4 Punkte einer Beispielmenge.

Für jeden der h Hüllpunkte wird somit eine Laufzeit von $O(n)$ benötigt, was in einer Gesamtlaufzeit für Jarvis's March von $O(n \cdot h)$ resultiert. Dadurch handelt es sich bei diesem Verfahren um einen *ausgabesensitiven Algorithmus*, was bedeutet, dass die Laufzeit neben der Eingabegröße n auch von der Größe h der Resultatmenge abhängt. Mit Chan's Algorithmus wird in Abschnitt 5.3.5 noch ein weiterer ausgabesensitiver Algorithmus vorgestellt, von dem sogar bewiesen werden kann, dass er in seiner Laufzeitkomplexität optimal ist. Im worst case liegen alle n Punkte auf der Hülle der Punktemenge, womit Jarvis's March eine Komplexität von $O(n^2)$ in diesem Fall erhält.

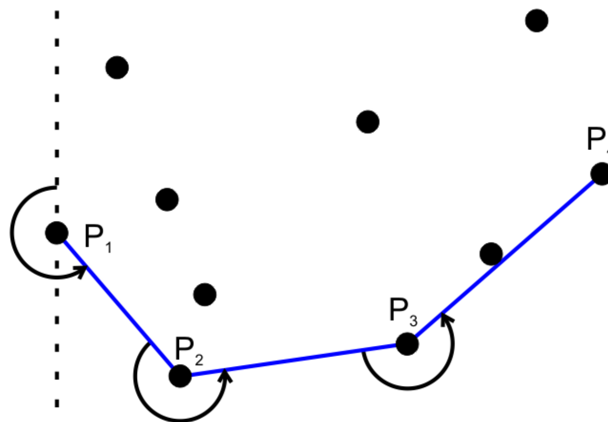

Abbildung 33: Beispiel für Jarvis's March

5.3.3 Graham's Scan

Dieser Algorithmus (siehe z. B. [2], S. 899 ff.) wurde 1972 von Ronald Graham publiziert und ist wie Jarvis's March inkrementell, da sukzessive neue Hüllpunkte gesucht und hinzugefügt werden. In einem ersten Schritt wird der Punkt mit der niedrigsten y-Koordinate gesucht. Dieser Punkt P_1 stellt den ersten Hüllpunkt dar. Im nächsten Schritt wird die Menge der übrigen Hüllpunkte aufsteigend nach dem Winkel sortiert, den die Punkte mit P_1 und der x-Achse aufspannen. Im finalen Schritt werden alle Punkte dieser sortierten Folge der Reihe nach betrachtet. Angenommen es wird als nächster Punkt P_i herangezogen (wie in Abbildung 34 für $i = 3$, also P_3, dargestellt). Weiterhin wird angenommen, dass alle P_j mit $j < i$ Punkte der bisher berechneten Hülle sind. P_i wird zur bisherigen Hülle hinzugefügt und es wird überprüft, ob die neue Teilhülle noch konvex ist. Liegt P_i links von $\overline{P_{i-2}\ P_{i-1}}$, so besteht weiterhin Konvexität und es kann mit der Betrachtung des nächsten Punktes fortgefahren werden. Ansonsten muss P_{i-1} aus der bisherigen konvexen Hülle entfernt werden. Die Bedingung ist weiter für Punkte vor P_{i-1} zu prüfen und es sind solange Punkte zu entfernen, bis die Bedingung erfüllt ist. Diese Abarbeitung wird wiederholt, bis P_1 wieder erreicht wird. Abbildung 35 zeigt ein Beispiel, bei dem die Konvexitäts-Bedingung für P_3 nicht erfüllt ist, da P_4 rechts von $\overline{P_2\ P_3}$ liegt. P_3 wird aus der bisherigen Hülle entfernt und es wird mit dem nächsten Punkt fortgefahren, da für P_4 in Verbindung mit $\overline{P_1\ P_2}$ Konvexität vorliegt.

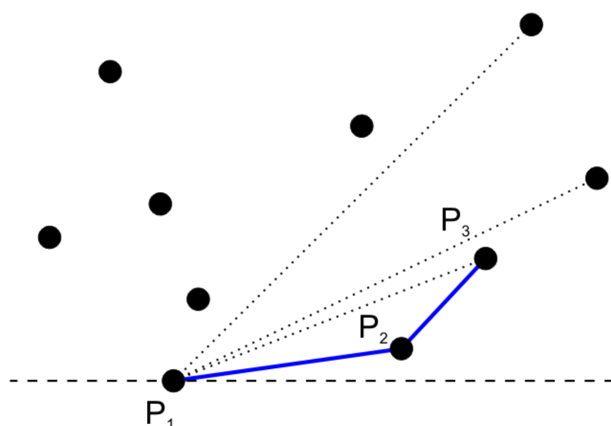

Abbildung 34: Erste Schritte eines Beispiels zu Graham's Scan

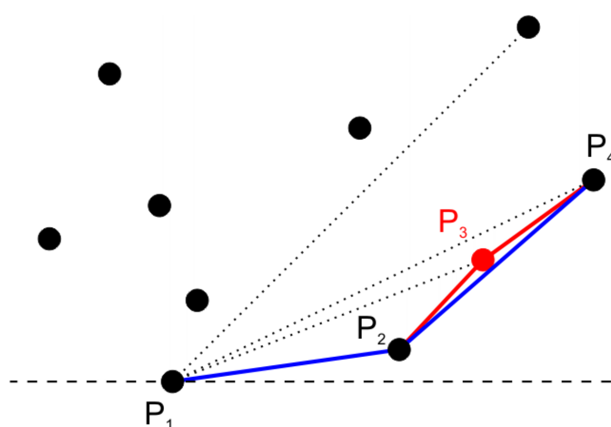

Abbildung 35: Verletzung der Konvexitätsbedingung bei Graham's Scan

Die Suche nach dem Punkt mit der kleinsten y-Koordinate benötigt eine Laufzeit von $O(n)$, die Sortierung $O(n \cdot log\ n)$. Das schlussendliche Finden der Hüllpunkte ist in $O(n)$ beendet. Die Gesamtlaufzeit von Graham's Scan wird also durch das Sortieren beschränkt und beträgt somit $O(n \cdot log\ n)$.

5.3.4 Quickhull

Quickhull wurde 1977 von W. F. Eddy vorgestellt und gehört zu den divide-and-conquer-Algorithmen, wobei nicht nur vom Namen her große Ähnlichkeit mit dem Sortieralgorithmus Quicksort besteht. Im initialen Schritt werden die Punkte P_1 mit der kleinsten und P_2 mit der größten x-Koordinate gefunden. Diese Punkte gehören auf jeden Fall zur kovexen Hülle. Die Linie $\overline{P_1\ P_2}$ wird als initiale Trennlinie für die rekursive Quickhull-Prozedur

verwendet und die ursprüngliche Punktemenge X wird in zwei Mengen X_1 und X_2 zerlegt (oberhalb bzw. unterhalb der Trennlinie). Die rekursive Prozedur arbeitet wie folgt (und ist sowohl für X_1 als auch X_2 anwendbar). Es werden ihr eine Trennlinie $\overline{P_1\ P_2}$ und die zu betrachtende Teil-Punktemenge Y übergeben, die oberhalb der Trennlinie liegen. Zunächst wird der Punkt Q aus Y bestimmt, der am weitesten von der Trennlinie entfernt liegt. Die Punkte P_1, P_2 und Q spannen dann ein Dreieck auf und Q ist aufgrund der Konvexität Teil der finalen konvexen Hülle. Alle Punkte innerhalb des Dreiecks können nicht zur konvexen Hülle gehören und werden verworfen. Abbildung 36 stellt diesen Vorgang dar. Die Rest-Punktemenge wird in Y_1 (alle Punkte links von $\overline{P_1\ Q}$) und Y_2 ((alle Punkte links von $\overline{Q\ P_2}$) zerlegt und die rekursive Prozedur wird für sie aufgerufen. Sind keine Punkte mehr in einer Teilmenge enthalten, so finden keine weiteren rekursiven Aufrufe statt. Bei Rückkehr der rekursiven Prozeduren wird die Teilhülle von Y_2 mit Q und der Teilhülle von Y_1 verbunden und diese größere Teilhülle zurück gegeben. Im obersten Aufruf wird die Teilhülle von X_1 mit P_1, der Teilhülle von X_2 und P_2 verbunden, wodurch sich die Ergebnishülle ergibt.

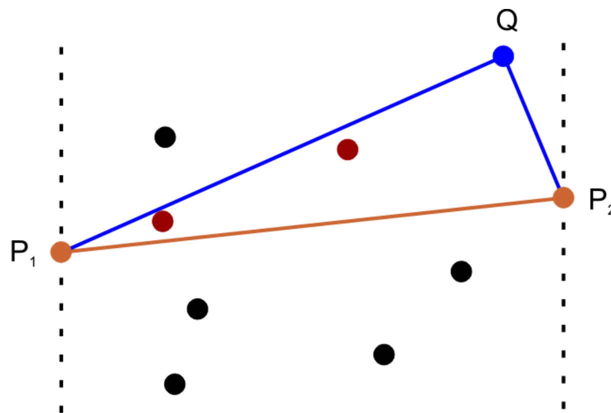

Abbildung 36: Beispiel für einen Quickhull-Schritt

Durch das frühzeitige Verwerfen irrelevanter Punkte arbeitet Quickhull sehr schnell. Seine Laufzeitkomplexität beträgt (wieder wie bei Quicksort) im Durchschnitt $O(n \cdot log\ n)$ und im schlechtesten Fall wird $O(n^2)$ benötigt (wenn alle Punkte auf einem Kreis/einer Ellipse liegen), was den Haken an dem Algorithmus darstellt.

5.3.5 Chan's Algorithmus

Dieser Algorihtmus wurde 1996 von Timothy M. Chan vorgestellt (siehe [1]). Es handelt sich um einen prune-and-search-Algorithmus, der dadurch interessant wird, dass seine

Laufzeit $O(n \cdot log\ h)$ beträgt. Er ist durch die Abhängigkeit von h ausgabesensitiv, weiterhin kann Optimalität für diese Komplexität gezeigt werden. Die Ausgabegröße h wird als Parameter mit in die Laufzeit einbezogen, da durch Transformation des Sortierproblems auf das Problem der Berechnung konvexer Hüllen in linearer Zeit gezeigt werden kann, dass $O(n \cdot log\ n)$ die bestmögliche Komplexität ist, die nur von n abhängt. Ist h klein, so können mit ausgabesensitiven Algorithmen gute Ergebnisse erreicht werden.

Chan's Algorithmus ist komplexer als die anderen hier beschriebenen Algorithmen, wodurch nicht auf jede Feinheit eingegangen wird. Angenommen die Ausgabegröße h sei bekannt (was i. Allg. natürlich nicht der Fall ist, doch diese Bedingung wird später fallen gelassen). Zunächst wird die Punktemenge X (mit $|X| = n$) zufällig in n/h Teilmengen R_i mit maximaler Größe h unterteilt. Von jeder dieser R_i wird die konvexe Hülle C_i mit einem $O(n \cdot log\ n)$ Algorithmus berechnet, z. B. mit Graham's Scan. Im zweiten Schritt wird Jarvis's March ausgeführt, um die konvexe Hülle von X zu berechnen. Dabei wird ausgenutzt, dass die konvexen Hüllen C_i bekannt sind. Soll von einem Hüllpunkt P_j der nächste Punkt mit Jarvis's March gefunden werden, so wird in jedem C_i eine binäre Suche nach dem am weitesten rechts von P_j liegenden Punkt Q_i durchgeführt (d. h. es gibt in C_i keinen Punkt rechts von $\overline{P_j\ Q_i}$). Abbildung 37 veranschaulicht dieses Vorgehen. Die binäre Suche ist dabei aufgrund der Konvexität möglich, wenn die Hüllpunkte C_i in geordneter Reihenfolge abgespeichert werden. Schließlich wird aus den gefundenen Punkten Q_i der „rechteste" ausgewählt, welcher dann den nächsten Hüllpunkt darstellt. Dieser Vorgang wird für alle weiteren Hüllpunkte wiederholt, bis der Startpunkt wieder erreicht wird.

Die Bedingung, dass h bereits zu Beginn bekannt ist, stellt ein großes Hindernis dar. Abhilfe schafft dabei einfach ein h^* festzulegen, welches eine Schätzung für h darstellt. Ist $h > h^*$, so kann Jarvis's March nach h^* Schritten abgebrochen werden. Ist die Hülle dann noch nicht vollständig, so wurde h^* zu klein gewählt. Ist $h \leq h^*$, so wird die Hülle vom Algorithmus vollständig berechnet. In [1] schlägt Chan vor, h^* als Folge von Zahlen 2^{2^t} mit $t = 1, \ldots$ zu wählen, um schnell zum Ziel zu gelangen. Mit den so entstehenden mehrfachen Durchläufen des eigentlichen Algorithmus ändert sich die Laufzeitkomplexität nicht:

$$\sum_{t=0}^{O(log\ log\ h)} O\left(n \cdot log\left(2^{2^t}\right)\right) = n \cdot \sum_{t=0}^{O(log\ log\ h)} O\left(2^t\right) = O\left(n \cdot log\ h\right)$$

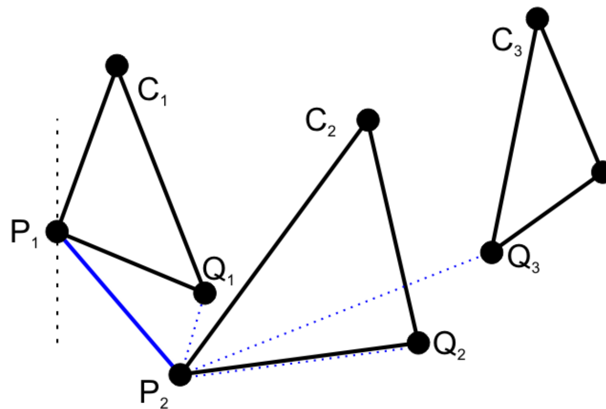

Abbildung 37: Beispiel für Chan's Algorithmus

Chan's Algorithmus ist ein prune-and-search Verfahren, da mit Graham's Scan über alle R_i Punkte verworfen werden, die nicht zur konvexen Hülle gehören können („*prune*") und dann mit Jarvis's March in der übrigen Menge nach Hüllpunkten gesucht wird („*search*"). Die Laufzeit von Graham's Scan für die n/h Teilmengen R_i der Größe h beträgt $O(n/h * h \cdot log\ h) = O(n \cdot log\ h)$. Bei Jarvis's March wird dann für jeden der h Hüllpunkte in jedem C_i der am weitesten rechts liegende Punkt mit binärer Suche gesucht, was eine Laufzeit von $O(n/h \cdot log\ h)$ ergibt, wodurch sich die Gesamtlaufzeit $O(n \cdot log\ h)$ nicht ändert.

5.3.6 Akl-Toussaint-Heuristik

Bei dieser Heuristik (siehe z. B. [9]) handelt es sich nicht um einen Algorithmus zur Berechnung konvexer Hüllen, sondern um einen Vorverarbeitungsschritt, durch den möglichst viele Punkte bereits vor der Hüllbildung eliminiert werden sollen. Grundlage dafür ist die Beobachtung, dass alle Punkte, die innerhalb einer Teilmenge der konvexen Hülle liegen, nicht zu dieser gehören können. Es werden bei dieser Heuristik also zunächst die 4 Extrema der Punktemenge in beiden Achsenrichtungen gesucht. Diese Punkte gehören auf jeden Fall zur konvexen Hülle. In einem zweiten Durchlauf können alle diejenigen Punkte eliminiert werden, die innerhalb dieses Vierecks liegen. Die Rest-Punktemenge wird dann an den eigentlichen Hüllalgorithmus zur Verarbeitung übergeben. Abbildung 38 zeigt ein Beispiel für die Anwendung der Akl-Toussaint-Heuristik. Die Laufzeit der Heuristik beträgt $O(n)$.

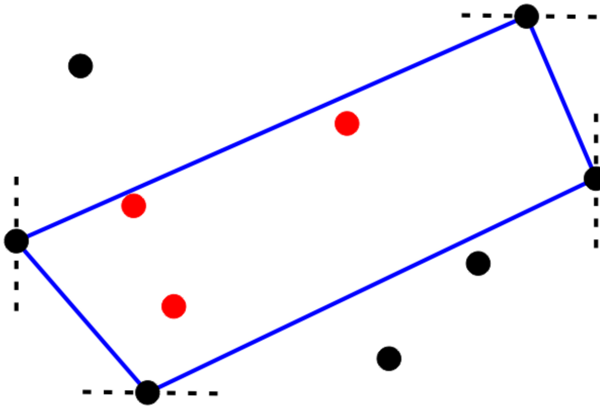

Abbildung 38: Beispiel für die Akl-Toussaint-Heuristik

5.4 Testfälle und Auswertungen

5.4.1 Zufällig

In diesem Testfall wurden Punkte zufällig im vorgegebenen Wertebereich erzeugt. Dabei fand eine Messung der Laufzeit über die Anzahl der erzeugten Punkte statt. Zunächst wurde der Brute-Force-Ansatz mit Jarvis's March und Quickhull verglichen. Das Ergebnis dieses Tests zeigt Abblidung 39.

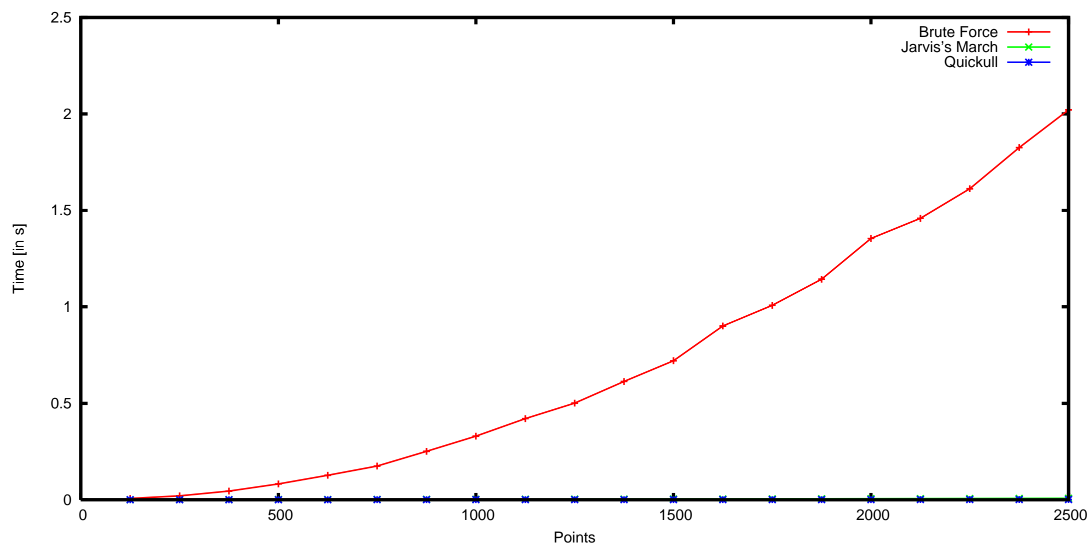

Abbildung 39: Zufällige Punktemenge, Brute-Force im Vergleich

Es ist zu erkennen, dass der $O(n^3)$-Ansatz von Brute Force undiskutabel gegenüber den anderen beiden Algorithmen ist, deren Laufzeiten bei diesen kleinen Punktemengen kaum

sichtbar sind. Aus diesem Grund war dies der einzige Test mit Brute Force im Vergleich zu anderen Algorithmen und es bleibt festzuhalten, dass Brute Force nie eine gute Wahl ist.

Der zweite Test über die zufällige Punktemenge fand als Vergleich zwischen den anderen 4 in Abschnitt 5.3 vorgestellten Hüllalgorithmen statt. Es wird ein gutes Abschneiden von Quickhull und Chan's Algorithmus erwartet. Graham's Scan sollte zudem Jarvis's March überlegen sein. Die Ergebnisse lassen sich in Abbildung 40 ablesen.

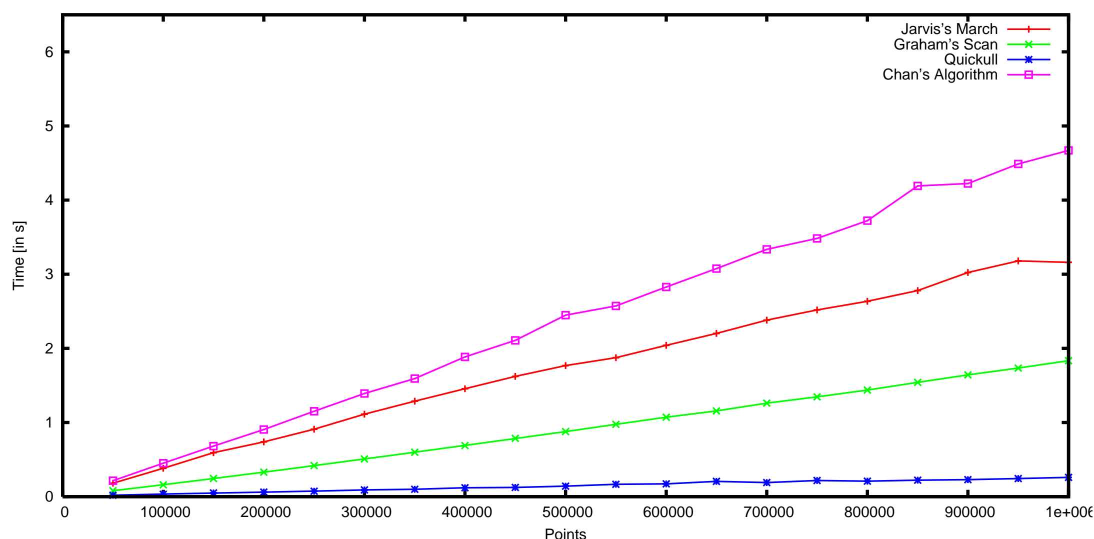

Abbildung 40: Zufällige Punktemenge, Vergleich anderer Hüllalgorithmen

Quickhull (Laufzeit 100%) ist eindeutig der beste Algorithmus in diesem Testfall, gefolgt von Graham's Scan (Laufzeit 706%), Jarvis's March (Laufzeit 1218%) und Chan's Algorithmus (Laufzeit 1800%) als Schlusslicht. Das Ergebnis entspricht ungefähr den Erwartungen bei einer zufälligen Punktemenge. Bei Jarvis's March müssen viele Hüllpunkte betrachtet werden und entsprechend hoch ist die Laufzeit. Nur das schlechte Abschneiden von Chan's Algorithmus verwundert, darauf wird in Abschnitt 5.4.5 noch eingegangen.

5.4.2 Kreis

In diesem Testfall wurden n Punkte gleichmäßig auf einem Kreis verteilt, dessen Mittelpunkt mit dem des Wertebereichs übereinstimmt und dessen Radius der Hälfte des Wertebereichs entspricht. Somit liegen alle Punkte der Menge auf der konvexen Hülle. Dies stellt den schlechtesten Fall für Quickhull, Jarvis's March und Chan's Algorithmus

dar, deren Laufzeit hier zu $O(n^2)$ (Quickhull, Jarvis) bzw. $O(n \cdot log\ n)$ (Chan) entartet und für die somit erwartet wird, dass sie schlecht abschneiden. Zumindest für Jarvis's March trifft das zunächst einmal zu. Seine Laufzeit explodiert bereits für kleine Punktemengen, wie Abbildung 41 zeigt.

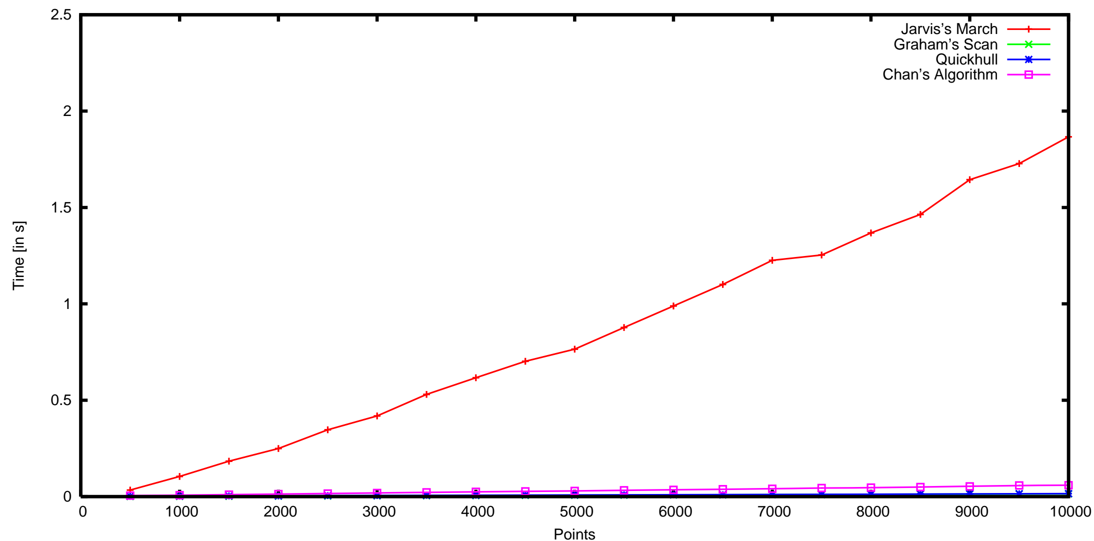

Abbildung 41: Kreispunkte, Explosion der Laufzeit bei Jarvis's March

Die anderen 3 Algorithmen wurden noch einmal gesondert untersucht und miteinander verglichen, was in Abbildung 42 dargestellt ist.

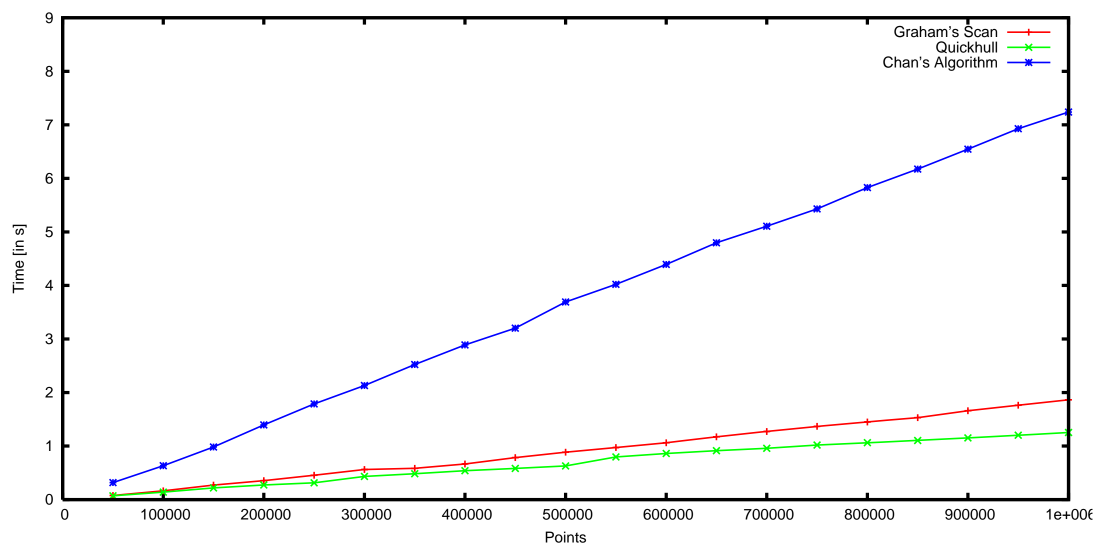

Abbildung 42: Kreispunkte, Vergleich der anderen Hüllalgorithmen

Graham's Scan (Laufzeit 100%) liefert wie vorhergesagt ein gutes Ergebnis, kann Quick-

hull (Laufzeit 67%) aber nicht in der Laufzeit unterbieten. Chan's Algorithmus (Laufzeit 388%) liefert aufgrund der $h = n$ Hüllpunkte wie erwartet schlechte Ergebnisse. Werden die Laufzeitkomplexitäten betrachtet, so gibt es ein großes n_0, ab dem Graham's Scan bessere Ergebnisse liefert als Quickhull und ein noch größeres n_1, ab dem Chan's Algorithmus weniger Laufzeit benötigt als Quickhull. Im hier betrachteten Intervall für die Anzahl der Punkte n wurden diese Schnittpunkte allerdings nicht annähernd erreicht und es ist zu bezweifeln, dass dieser Fall für praktische Punktemengen eintritt.

5.4.3 Quadrat

Dieser Testfall stimmt mit dem zufälligen Fall aus Abschnitt 5.4.1 überein, besitzt aber einen relevanten Unterschied: die 4 Eckpunkte des Wertebereichs der Punkte sind mit in der erzeugten Punktemenge enthalten. Das bedeutet, dass die konvexe Hülle sehr klein ist, sie besteht nur aus diesen 4 Punkten (wenn so programmiert wurde, dass bei mehreren Punkten auf einer Linie die Extrema genommen werden, was hier aber der Fall ist). Es wird erwartet, dass die ausgabesensitiven Algorithmen Jarvis's March und Chan's Algorithmus gute Ergebnisse liefern und sich die Laufzeiten denen von Graham's Scan und Quickhull annähern bzw. diese übertreffen. Abbildung 43 zeigt die Auswertung dieses Testfalls.

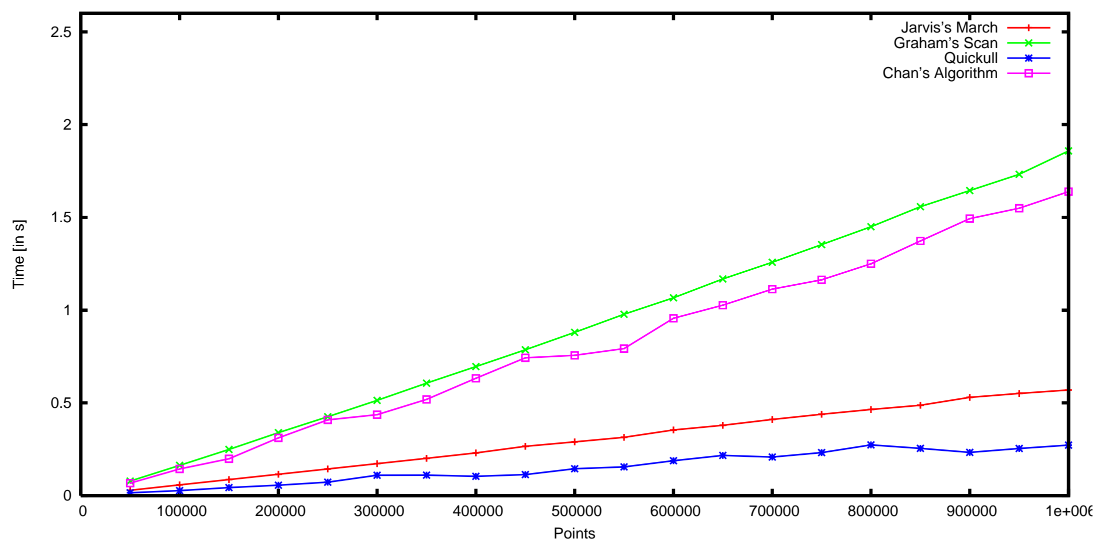

Abbildung 43: Quadrat, Vergleich der Algorithmen

Es ist erkennbar, dass Quickhull (Laufzeit 100%) immernoch der beste Algorithmus ist, Jarvis's March (Laufzeit 209%) diesmal jedoch an zweiter Stelle folgt. Chan's Algorithmus

(Laufzeit 603%) ist zumindest noch besser als Graham's Scan (Laufzeit 683%), was die Erwartungen bestätigt. Die Theorie besagt auch hier, dass Chan's Algorithmus und Jarvis's March ab sehr großen n_0 und n_1 besser sind als Quickhull. Im betrachteten praktischen Intervall konnten diese Werte jedoch auch hier nicht annähernd erreicht werden.

5.4.4 Einsatz der Akl-Toussaint-Heuristik

Die Akl-Toussaint-Heuristik wird vor allem dort Sinn machen, wo innerhalb des Vierecks, welches durch die Extrempunkte aufgespannt wird, viele Punkte vorhanden sind, z. B. bei zufälligen Punktemengen. Abbildung 44 zeigt die Laufzeiten bei zufälliger Punkteverteilung entsprechend Abschnitt 5.4.1 unter Vorverarbeitung mit der Akl-Toussaint-Heuristik.

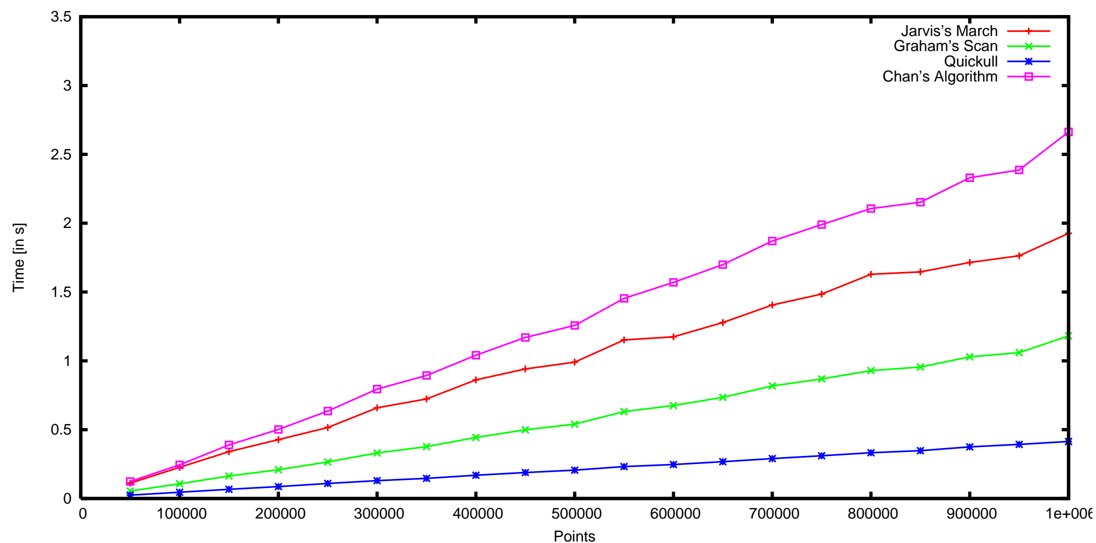

Abbildung 44: Zufällige Punktemenge, Vorverarbeitung mit Akl-Toussaint-Heuristik

Im direkten Vergleich der Laufzeiten mit denen aus Abbildung 40 ohne Akl-Toussaint-Heuristik ergibt sich hier für Jarvis's March eine Verbesserung der Laufzeit von 39%, für Graham's Scan von 36% und für Chan's Algorithmus von 57%. Interessant ist das Verhalten von Quickhull. Dieses ist ca. 60% langsamer als ohne Verwendung der Heuristik. Das ist mit der Abarbeitung von Quickhull zu begründen. Dort findet ein der Heuristik sehr ähnlicher Prozess statt, indem Punkte innerhalb der sich aufspannenden Dreiecke frühzeitig verworfen und nicht weiter betrachtet werden. Daher macht die Heuristik mehr Mühe als sie Nutzen bringt und hilft nur im ersten Schritt, schneller die Punkte mit minimaler/maximaler x-Koordinate zu finden, wesentlich.

Im Fall der Punkteverteilung auf einem Kreis wie in Abschnitt 5.4.2 muss die Laufzeit der Akl-Toussaint-Heuristik vollständig auf die Laufzeiten der Algorithmen addiert werden, da durch sie keine Punkte eliminiert werden können. Es ergeben sich im Mittel 11% schlechtere Laufzeiten. Dieser Fall ist eindeutig und somit an dieser Stelle uninteressant für weitere Betrachtungen.

5.4.5 Fazit

Es lässt sich festhalten, dass Quickhull in allen Fällen am wenigsten Laufzeit benötigt hat, selbst dort, wo ein schlechtes Abschneiden zu erwarten war (Abschnitt 5.4.2). Daher ist Quickhull für die Hüllberechnung uneingeschränkt zu empfehlen. Danach bietet sich Graham's Scan an, welches im zufälligen Fall (Abschnitt 5.4.1) und vor allem bei prozentual vielen Hüllpunkten (Abschnitt 5.4.2) gut arbeitet. Nur beim speziellen Fall des Quadrats in Abschnitt 5.4.3 konnte Jarvis's March trumpfen.

Das durchweg schlechte Abschneiden von Chan's Algorithmus verwundert zunächst. Es muss allerdings dazu gesagt werden, dass bei der Implementierung nicht auf Code-Optimalität Wert gelegt wurde und sich hier sicher noch ein Teil der Laufzeit hätte einsparen lassen können. Trotzdem wurde ein optimaler ausgabesensitiver Algorithmus implementiert, was einmal mehr die nur beschränkte Praxis-Relevanz der O-Notation unterstreicht. Durch die vielen durchzuführenden Operationen und die notwendige wiederholte Abarbeitung des Algorithmus fällt die praktische Laufzeit so hoch aus.

Die Akl-Toussaint-Heuristik stellt vor allem für zufällig verteilte Punktemengen eine gute und einfache Möglichkeit dar, viele Punkte bereits vor der eigentlichen Hüllberechnung zu eliminieren und dadurch Rechenzeit zu sparen. Nicht empfohlen werden kann die Heuristik bei Einsatz von Quickhull aus den in Abschnitt 5.4.4 dargestellten Gründen. Die Akl-Toussaint-Heuristik sollte außerdem nicht eingesetzt werden, wenn nur wenige Punkte in dem von der Heuristik aufgespannten Viereck liegen, was in der Praxis z. B. bei der Berechnung der konvexen Hülle einer Objektkontur häufig vorkommt.

Literatur

[1] Chan, Timothy M.: Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions. In: *Geometry: Discrete and Computational Geometry* 16 (1996). http://citeseer.ist.psu.edu/181154.html

[2] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L.: *Intoduction to Algorithms*. The MIT Press, 1990. – ISBN 0–262–03141–8

[3] Jauernig, Matthias: *Quelltexte und Auswertungsdaten*. http://www.linux-related.de/studium/algeng/Quellen_Jauernig.zip, letzter Zugriff: 15.01.2008

[4] Jauernig, Matthias: *Verbesserungen von Quicksort*. http://linux-related.de/coding/sort/sort_quick.htm#verbesserung, letzter Zugriff: 13.01.2008

[5] Klein, Rolf ; Kamphans, Tom: *Der Pledge-Algorithmus*. http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo6.php, letzter Zugriff: 12.01.2008

[6] Microsoft: *Gewusst wie: Verwenden des hochauflösenden Zeitgebers*. http://msdn2.microsoft.com/de-de/library/aa964692(VS.80).aspx, letzter Zugriff: 13.01.2008

[7] Sedgewick, Robert: *Algorithmen in C*. Addison-Wesley, 1992. – ISBN 3–89319–376–6

[8] Sun Microsystems, Inc.: *ThreadMXBean (Java Platform SE 6)*. http://java.sun.com/javase/6/docs/api/java/lang/management/ThreadMXBean.html, letzter Zugriff: 13.01.2008

[9] Wikipedia: *Akl-Toussaint heuristics*. http://en.wikipedia.org/wiki/Convex_hull_algorithms#Akl-Toussaint_heuristics, letzter Zugriff: 14.01.2008