Bachelorarbeit

Experimentelle Analyse von Core-Algorithmen zur Lösung des Rucksackproblems

Institut für Informatik

Mathematisch-Naturwissenschaftliche Fakultät

Rheinische Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Danny-Marvin Rademacher

Matrikelnummer 2535707

Erstprüfer	Prof. Dr. Heiko Röglin
Zweitprüfer	Prof. Dr. Norbert Blum
Abgabedatum	9. Juli 2015

Zusammenfassung

Diese Bachelorarbeit beschäftigt sich mit dem Konzept der Core-Algorithmen um einen effizienten Algorithmus zu implementieren, der das Rucksackproblem im Erwartungswert auf uniform zufälligen Nutzen und Gewichten in $\mathcal{O}(n\, polylog(n))$ löst. Dieser Algorithmus beginnt mit der Berechnung des fraktionalen Rucksackproblems und tauscht von der Lösung ausgehend so lange Objekte aus bis eine optimale ganzzahlige Lösung vorliegt.

Der Fokus dieser Arbeit liegt auf der experimentellen Analyse dieses Core-Algorithmus und Algorithmen zur Lösung des fraktionalen Rucksackproblems. Die Arbeit baut auf Ergebnissen von René Beier und Berthold Vöcking auf. Der Hauptteil dieser Arbeit liegt im Verbessern der bisher verwendeten Algorithmen zur Lösung des fraktionalen Rucksackproblems. Hierzu werden neuere Erkenntnisse von ähnlichen Algorithmen verwendet, welche insbesondere für praktische Anwendungen stark verbessert wurden.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt und durch meine Unterschrift, dass die vorliegende Arbeit von mir selbstständig, ohne fremde Hilfe angefertigt worden ist. Inhalte und Passagen, die aus fremden Quellen stammen und direkt oder indirekt übernommen worden sind, wurden als solche kenntlich gemacht. Ferner versichere ich, dass ich keine andere, außer der im Literaturverzeichnis angegebene Literatur verwendet habe. Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Bonn, den 9. Juli 2015

Danny-Marvin Rademacher

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das *Rucksackproblem* ist ein kombinatorisches Optimierungsproblem. Es ist eines der bekannten NP-vollständigen Probleme. Als Anschauung dient ein Rucksack und eine Menge verschiedener Gegenstände. Der Rucksack hat eine beschränkte Kapazität und jeder Gegenstand ein Gewicht und einen Nutzen. Dann muss eine Auswahl getroffen werden unter der alle gewählten Gegenstände in den Rucksack passen und der Nutzen aller ausgewählten Gegenstände maximal ist.

Diese Arbeit ist eine experimentelle Analyse, welche einen Core-Algorithmus zur Lösung des Rucksackproblems untersucht. In der Praxis sind Core-Algorithmen das am häufigsten verwendete Konzept. Diese Algorithmen berechnen erst die optimale Lösung der Instanz in einer Variante, in der ein einziges Objekt fraktal verwendet werden kann. Eine solche Lösung unterscheidet sich nur in wenigen Objekten von der optimalen ganzzahligen Lösung.

In Kapitel 2 wird der von mir implementierte Algorithmus ExpandingCore und die Grundlagen des Core-Algorithmus formal beschrieben. In den Kapiteln 3 und 4 werden die Algorithmen erklärt, die zu dem Core-Algorithmus gehören und von diesem benutzt werden um eine optimale Lösung zu bestimmen. Zudem werden die Stellen erläutert, an denen die Algorithmen modifiziert wurden um schneller und speichereffizienter als gewöhnlich für die besonderen Anforderungen des Core-Algorithmus zu sein. Zum Schluss enthält Kapitel 5 die experimentelle Analyse der besprochenen Algorithmen um das Verhalten in der Praxis zu untersuchen.

1.1 Formale Definition

Eine Instanz des *Rucksackproblems* besteht aus einer Menge von Objekten, welche jeweils durch einen Nutzen und ein Gewicht ausgezeichnet werden, sowie einem Kapazitätslimit. Das Ziel dieses Problems besteht darin, eine Teilmenge der Objekte zu finden, welche die Summe der Nutzen maximiert und deren Summe aller Gewichte nicht größer als die Kapazität ist.

Sei $n \in \mathbb{N}$ die Anzahl der Objekte. Dann bezeichne $p = (p_1, p_2, ..., p_n)^T \in \mathbb{R}^n_{\geq 0}$ die Nutzen und $w = (w_1, w_2, ..., w_n)^T \in \mathbb{R}^n_{\geq 0}$ die Gewichte. $W \in \mathbb{R}_{\geq 0}$ bezeichnet die Kapazität. Dann ist $\{0,1\}^n$ die Menge aller Lösungsvektoren.

$$\begin{aligned} \text{maximiere} \quad & p^T x = \sum_{i=1}^{n} p_i x_i \\ \text{unter} \quad & w^T x = \sum_{i=1}^{n} w_i x_i \leq W \\ \text{mit} \quad & x \in \{0,1\}^n \end{aligned}$$

Es gibt eine Abwandlung des Rucksackproblems, in dem die Einschränkungen des Lösungsvektors weniger stark sind. Diese wird fraktionales Rucksackproblem genannt. Ein Eintrag des Lösungsvektors x_i darf in diesem fraktionalen Rucksackproblem statt den Werten 0 oder 1 nun einen beliebigen reellen Wert aus dem Intervall $(0, 1]$ annehmen. Zusätzlich wird in dieser Arbeit die Bedingung gestellt, dass alle anderen Einträge entweder 0 oder 1 sind.

1.2 Bisherige Arbeit

Die Grundlagen dieser Bachelorarbeit finden sich in dem Paper „*Probabilistic analysis of knapsack core algorithms*“ von René Beier und Berthold Vöcking aus dem Jahr 2004 [BV04]. Zwei Jahre später haben die beiden Autoren eine experimentelle Analyse zu diesen Algorithmen veröffentlicht [BV06].

Das ursprüngliche Konzept der Core-Algorithmen stammt von Balas und Zemel. Das Konzept basiert auf der Erkenntnis, dass in vielen Problemen eine Approximation bereits ein sehr gutes Ergebnis liefert und sich nur geringfügig von der optimalen Lösung unterscheidet. Ein weiterer wichtiger Beitrag stammt von Goldberg und Marchetti-Spaccamela, welche die Definition des Core in Abschnitt 2.2 eingeführt haben. Die erfolgreiche Implementation von Beier und Vöcking entstand aus dieser Arbeit.

Zusätzlich wird ein möglichst effizienter Algorithmus zur Lösung des fraktionalen Rucksackproblems benötigt. Um für meine Arbeit einen solchen zu konstruieren, habe ich auf die Ergebnisse von Yaroslavskiy zurückgegriffen [Yar09]. Er hat eine Implementation von Quicksort vorgestellt, welche deutlich performanter als alle bereits verwendeten und hochgradig optimierten Sortieralgorithmen ist. Diese verwendet statt üblicherweise einem nun zwei Pivotelemente im Partitionierungsschritt. Die Grundidee von Hoares Quicksort sowie die Implementation von Yaroslavskiy wurden verwendet um eine effiziente Implementation zum Bestimmen der fraktionalen Lösung des Rucksackproblems zu schreiben.

Kapitel 2

Core-Algorithmus

In diesem Kapitel wird das Paper von Beier und Vöcking zusammengefasst, in dem der Core-Algorithmus beschrieben wird. Das Konzept der Core-Algorithmen wurde erstmals von Balas und Zemel vorgestellt [BV04].

Die Idee hinter dem Core-Konzept besteht darin mit einer optimalen Lösung für das fraktionale Rucksackproblem zu beginnen. Die fraktionale Variante unterscheidet sich in dem Punkt, dass Objekte auch nur teilweise zu den Lösungen gezählt werden dürfen. Ein Eintrag im Lösungsvektor $x_k \in \{0, 1\}$ darf stattdessen im Intervall $[0, 1]$ liegen. Das Objekt (p_k, w_k) wird im folgenden als Teilungsobjekt bezeichnet und die fraktionale Lösung ohne das Teilungsobjekt als Teilungslösung. Näheres dazu findet sich in Kapitel 3.

In der Teilungslösung werden dann Objekte ausgetauscht bis eine optimale ganzzahlige Lösung gefunden ist. Die Menge der interessanten Tauschobjekte, also Kandidaten für die optimale Lösung, wird als Core bezeichnet. Der Algorithmus baut darauf, dass der Core bei gewöhnlichen Instanzen nur wenige Objekte enthält. Der Core umfasst die Objekte, deren Effizienz relativ nah an der des Teilungsobjekts liegt. Insgesamt ergeben sich drei Mengen von Objekten, neben dem Core noch die Menge M_1 mit wertvollen Objekten, die sicher zur Lösung gehören und die Menge M_2 mit den wertlosen Objekten, die nicht zur Lösung gehören können.

Die optimale Lösung für die Eingabeinstanz besteht demnach aus allen Objekten der Menge M_1 und der optimalen Lösung der Probleminstanz, die alle Objekte des Core umfasst und eine Kapazität von $W - \sum_{x \in M_1} w(x)$ besitzt. Die Laufzeit des Algorithmus ist dann direkt abhängig vom Bestimmen des Core und dem Finden einer optimalen Lösung für das Teilproblem. Das Teilproblem kann mit allen möglichen Algorithmen gelöst werden, häufig wird ein typischer *Divide and Conquer* Ansatz gewählt. Der Algorithmus nutzt jedoch die Struktur der Objekte aus, welche sich in einem rekursiven Aufruf nicht hinreichend ändern würden. Es ist zwar möglich, dass das Teilungsobjekt ein anderes ist und der Core sich verkleinert, aber vollständig lösen können wird man das Problem nicht. Daher wird stattdessen der Algorithmus Nemhauser-Ullmann aus Kapitel 4 zur Lösungsbestimmung im Core verwendet. Der Algorithmus vertraut

darauf, dass die relative Größe des Core gering ist. Dann lässt sich das Gesamtproblem effizient lösen.

2.1 Geometrische Anschauung

Als Anschauung dient ein Graph, welcher jedes Objekt i als Punkt (w_i, p_i) in einem kartesichen Koordinatensystem abbildet. Der Algorithmus zur Bestimmung des Teilungsobjekts arbeitet derart, dass man mit einer senkrechten Halbgeraden vom Ursprung startet. Diese wird dann im Uhrzeigersinn von der Y-Achse in die Richtung der X-Achse gedreht. Jedes Objekt, das sich geometrisch über der Halbgeraden befindet, stellt ein Objekt dar, welches in den Rucksack gepackt wird. Die Steigung von einer Linie, die vom Ursprung aus zu einem Punkt läuft, entspricht der Effizienz des Objekts. Sobald die Halbgerade auf ein oder mehrere Objekte trifft, die nicht mehr vollständig in den Rucksack passen, wird diese nicht mehr weiter gedreht und hält an.

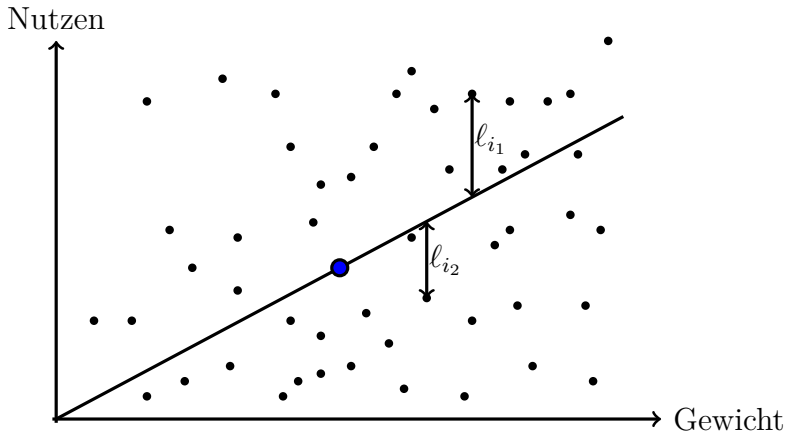

Abbildung 2.1: Ein jedes Objekt i ist visualisiert durch einen Punkt (w_i, p_i). Das Teilungsobjekt liegt auf dem Danzig-Strahl. Der Verlust eines Objekts ist durch den vertikalen Abstand zum Danzig-Strahl für zwei Objekte dargestellt.[Rö15]

Diese Halbgerade wird Danzig-Strahl genannt. Wenn auf dem Danzig-Strahl nur ein Punkt liegt, ist dieser das Teilungsobjekt. Sollten mehrere Punkte darauf liegen, haben diese alle dieselbe Effizienz und es ist nicht festgelegt, ob ein Objekt fraktional, ganzzahlig oder gar nicht eingepackt wird. Es macht keinen Unterschied, welcher Punkt zuerst entdeckt wird und priorisiert in den Rucksack gepackt wird. Im Abschnitt 2.2 wird ausgeführt, dass der Core aus allen Objekten besteht, deren vertikaler Abstand kleiner als ein bestimmter Wert ist. Der Bereich wird durch zwei Halbgeraden abgegrenzt, die parallel zum Danzig-Strahl liegen und deren vertikale Abstände gleich sind.

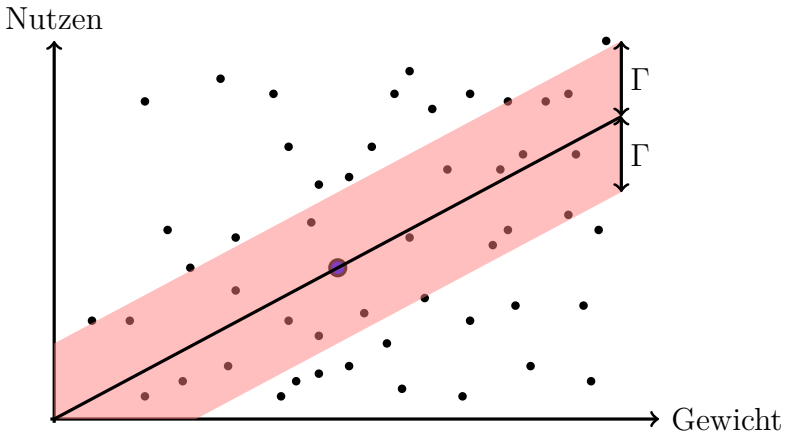

Abbildung 2.2: Der Core besteht allen Objekten mit Verlust kleiner gleich Γ.[Rö15]

2.2 Formale Definition des Core

Es gibt mehrere Möglichkeiten, um den Core und die darin enthaltenen Objekte zu bestimmen. Das oben angegebene Paper befasst sich mit den Definitionen von Goldberg und Marchetti-Spaccamela und den daraus resultierenden Algorithmen.

Die optimale Lösung des fraktionalen Rucksackprobems der aktuellen Instanz wird $\bar{x}$ genannt. Diese Lösung hat nur ein fraktionales Objekt, d.h. die Struktur von $\bar{x}^T$ ist $(1, 1, ..., 1, f, 0, 0, ..., 0)$. f nimmt einen Wert aus dem Intervall $(0, 1]$ an und hat den Index k. Alle in Kapitel 3 beschriebenen Algorithmen liefern eine solche Lösung. Im folgenden ist die Effizienz des Teilungsobjekts $r := \frac{p_k}{w_k}$. Objekte mit einer Effizienz größer r werden als hochwertig bezeichnet, Objekte mit einer Effizienz kleiner r als minderwertig.

Definiere $\Delta(x) = \{i \in [n] \mid \bar{x}_i \neq x_i\}$ für jede gültige Lösung des ganzzahligen Rucksackproblems x als die Menge aller Objekte, in denen sich die Lösungen $\bar{x}$ und x unterscheiden.

Wenn ein Objekt aus der Teilungslösung entfernt wird, kann es durch ein oder mehrere Objekte ersetzt werden, die nicht zur Teilungslösung gehören und eine geringere Effizienz als das entfernte Objekt aufweisen. Alle Objekte der Teilungslösung haben eine Effizienz von zumindest r, während die Effizienz aller anderen Objekte höchstens r ist. Ein Unterschied in der Effizienz zwischen zwei Objekten resultiert in einem Verlust an Nutzen. Sobald hochwertige Objekte durch minderwertige ausgetauscht werden, ergibt sich ein Verlust. Dieser Verlust kann dadurch ausgeglichen werden, dass die Restkapazität $W - w^T\bar{x}$ von den neuen Objekten in Anspruch genommen wird und die Gesamtkapazität W dadurch besser ausgeschöpft wird.

Der Verlust, der auftrifft, wenn ein Objekt ausgetauscht wird, lässt sich mit $l_i := |p_i - rw_i|$ quantifizieren. Der Verlust lässt sich durch den vertikalen Abstand des Objekts veranschaulichen, welcher in dem Graphen der Objekte von dem Punkt (Nutzen, Gewicht) bis zum Danzig-Strahl läuft. Das Lemma 2.1 wurde erstmals von Goldberg und Marchetti-Spaccamela bewiesen. Es besagt, dass die Differenz der Nutzen von

$\bar{x}$ und einer beliebigen Lösung x gleich der Summe aller Verluste $\sum_{i\in\Delta(x)} l_i$ und den Nutzen des nicht verwendeten Gewichts, den das frakale Objekte ausnutzen kann, $r(W - w^T x)$ ist. Dieser Term wird minimal, wenn $x = x^\star$ die optimale ganzzahlige Lösung ist. Diese Differenz wird als ganzzahliger Nutzenabstand Γ bezeichnet.

Lemma 2.1. *[Rö15, Seite 31] Sei $x \in \{0,1\}^n$ eine beliebige Lösung. Dann gilt*

$$p^T\bar{x} - p^T x = r(W - w^T x) + \sum_{i\in\Delta(x)} l_i$$

Beweis. Definiere $\Delta_0(x) := \{i \in [n] \mid i < i^\star \text{und } x_i = 0\}$ sowie $\Delta_1(x) := \{i \in [n] \mid i > i^\star \text{und } x_i = 1\}$. Dann gilt $\Delta(x) = \Delta_0(x) \cup \Delta_1(x) \cup \{i^\star\}$. Für alle Objekte $i \in \Delta_0(x)$

$$p_i\bar{x}_i - p_i x_i = p_i = rw_i + l_i$$

und für alle Objekte $i \in \Delta_1(x)$ gilt

$$p_i\bar{x}_i - p_i x_i = -p_i = -(rw_i - l_i) = -rw_i + l_i.$$

Außerdem gilt

$$p_{i^\star}\bar{x_{i^\star}} - p_{i^\star}x_{i^\star} = rw_{i^\star}(\bar{x_{i^\star}} - x_{i^\star}).$$

Der Verlust des Teilungsobjekts k beträgt $|p_k - rw_k| = |p_k - \frac{p_k}{w_k}w_k| = 0$.

Insgesamt folgt

$$\begin{aligned}
p^T\bar{x} - p^T x &= \sum_{i=1}^{n} p_i\bar{x}_i - p_i x_i \\
&= \left(\sum_{i\in\Delta_0(x)} rw_i + l_i\right) + \left(\sum_{i\in\Delta_1(x)} -rw_i + l_i\right) + rw_{i^\star}(\bar{x}_{i^\star} - xi^\star) \\
&= r\left(\sum_{i\in\Delta_0(x)} w_i + \sum_{i\in\Delta_1(x)} w_i + w_{i^\star}(\bar{x}_{i^\star} - xi^\star)\right) + \sum_{i\in\Delta_0(x)\cup\Delta_1(x)} l_i \\
&= r(w^T\bar{x} - w^T x) + \sum_{i\in\Delta(x)} l_i \\
&= r(W - w^T x) + \sum_{i\in\Delta(x)} l_i
\end{aligned}$$

□

Der Wert Γ ist eine obere Abschätzung der Summe aller Verluste und damit auch jedes Verlustes l_i von Objekten in $\Delta(x^\star)$. Alle Objekte in $\Delta(x^\star)$ liegen höchstens Γ vom Danzig-Strahl entfernt. Damit kann nun der Core als Menge aller Objekte mit höchstens einem Verlust Γ definiert werden. Da Γ unbekannt ist und nur aus einer optimalen Lösung bestimmt werden kann, braucht man ein Verfahren um auch ohne genaue Kenntnis von Γ den Core zu bestimmen.

2.3 Algorithmus ExpandingCore

Ein Verfahren um den Core zu bestimmen, besteht darin, dass man Γ mit der Variable γ abschätzt und die Genauigkeit im Verlauf des Algorithmus verbessert. γ ist dabei die beste bekannte obere Schranke für Γ. Die Güte der Annäherung von γ hat ausschließlich Auswirkungen auf die Laufzeit, aber nicht auf die Korrektheit des Algorithmus. Der Wert von Γ ist $p^T\bar{x} - p^T x^\star$. Da $x^\star$ eine optimale Lösung darstellt, ist der Nutzen jeder anderen Lösung höchstens genau so groß. Damit stellt der Term $p^T\bar{x} - p^T x$ für alle Lösungen x eine obere Schranke dar.

Der Algorithmus ExpandingCore setzt γ zu Beginn auf den Nutzenabstand zwischen der optimalen fraktionalen Lösung $\bar{x}$ und der Teilungslösung Da die Teilungslösung sich nur im Eintrag k unterscheidet, ist der initiale Wert $p_k\bar{x}_k$. Mit jeder Iteration findet der Algorithmus neue Lösungen. Da in Zeile 7 immer die nutzengrößte Lösung gewählt wird, reduziert sich der Wert γ in jeder Iteration in der er geändert wird. Der Name stammt daher, dass dies Algorithmus mit einem leeren Core startet und diesen jeder Iteration um ein Objekt erweitert wird. Das neue Objekt hat unter allen Objekten außerhalb des Core den geringsten Verlust l. Damit steigt das Minimum der Verluste von Objekten außerhalb des Core in jedem Schritt an. Dadurch kann der Verlust l_i des in der nächsten Iteration betrachteten Objekts nicht sinken. Demzufolge muss der Wert γ überschritten werden, da jener gleich bleibt oder sinkt.

$$\forall i \in \text{Core} : \gamma = p^T\bar{x} - p^T x \geq p^T\bar{x} - p^T x^\star = r(W - w^T x) + \sum_{i \in \Delta(x)} l_i \geq l_i$$

Die Ungleichung gilt für alle i, für die das zugehörige Objekt im Core liegt. Sollte die Gleichung nicht mehr stimmen, liegt das Objekt i außerhalb. Sobald der Verlust eines Objekts einmal γ überschreitet, gilt das für alle weiteren Objekte, da die Verluste aller weiteren Objekte mindestens genau so groß sind. Der Algorithmus kann aufhören, da alle dem Core zugehörigen Objekte bereits abgearbeitet wurden. Da eine optimale Lösung $x^\star$ bereits gefunden ist und diese auch die nutzengrößte gültige Lösung ist, wird diese der Variablen x in Zeile 7 zugewiesen werden. Dann entspricht γ dem ganzzahligen Nutzenabstand Γ.

Die Lösung x wird aus der Teilungslösung heraus konstruiert. Die Teilungslösung enthält genau die Objekte, deren Effizienz größer als die Effizienz des Teilungsobjekts ist. x entspricht in dem Algorithmus immer nur einer leichten Modifikation der Teilungslösung. Hier werden höchstens die Objekte, die bereits im Core liegen, ausgetauscht und die Summe der ausgetauschten Objekten ist höchsten γ. Die Größe des Core beträgt nur einen Bruchteil der Anzahl der Objekte, sodass der Aufwand für die Berechung von x nicht sonderlich hoch ist. Von der Lösung x wird nicht der Lösungsvektor benötigt, sondern bloß der Nutzen und das Gewicht $p^T x$ und $w^T x$ sowie der Gesamtverlust aller ausgetauschten Objekte $\sum_{i \in \Delta(x)} l_i \leq \gamma$.

Jede Lösung wird aus der Teilungslösung konstruiert. Daher braucht man deren Nutzen und Gewicht nicht zu berücksichtigen. Im Vergleich mit der fraktionalen Lösung wird außer dem Teilungsobjekt, welches einen Verlust von 0 hat, kein Objekt ausgetauscht. Diese Lösung hat folglich keinen Verlust. Daher muss im Algorithmus nur

die aus dem Core heraus bestimmte Lösung verwaltet werden, welche nur Objekte innerhalb des Core berücksichtigt. Anschließend kann man diese Lösung zusammen mit der Teilungslösung kombinieren, um eine optimale Lösung für die Eingabeinstanz zu erhalten. Dies funktioniert uneingeschränkt, da jedes Gewicht einer Lösung aus den Objekten im Core nicht die Gesamtkapazität überschreitet und die beiden Nutzen nur addiert werden müssen.

Algorithmus 1 ExpandingCore

```
1: Berechne die optimale fraktionale Lösung x̄
2: γ ← p_k x̄_k
3: repeat
4:     i ← Index eines nicht betrachteten Objekts mit minimalem Verlust
5:     Erweitere Core um Objekt i
6:     x ← nutzengrößte gültige Lösung aus dem Core hervorgehend
7:     γ ← p^T x̄ − p^T x
8: until l_i ≥ γ
9: return x
```

Die Korrektheit des Algorithmus folgt aus der Reduktion des Problems. Alle Objekte, die nicht innerhalb des Core liegen, müssen genau so gewählt werden wie in der Teilungslösung. D.h., dass alle Objekte über dem Core in die Lösung genommen werden und alle Objekte unter dem Core aus der Lösung ausgeschlossen werden. Sollte es ein Objekt i geben, sodass sich $\bar{x}$ und x unterscheiden, dann liegt es nach Definition in $\Delta(x)$. Da alle Lösungen außerhalb des Core einen Verlust $l_i > \Gamma$ haben, folgt der Widerspruch

$$\Gamma = p^T\bar{x} - p^T x^\star \geq l_i > \Gamma$$

Kapitel 3

Bestimmung des Teilungsobjekts

Die Algorithmen zur Partitionierung und die damit zusammenhängenden Informationen finden sich in der Masterarbeit von S. Wild [Wil15]. Die weiteren Zitate aus anderen Quellen sind gesondert gekennzeichnet.

In diesem Kapitel wird hauptsächlich das fraktionale Rucksackproblem betrachtet. Das Objekt k, das einen fraktionalen Eintrag im Lösungsvektor hat, wird im folgenden Teilungsobjekt genannt. Für den Core-Algorithmus wurde bereits in Kapitel 2 erläutert, dass besonders das Teilungsobjekt interessant ist und die daraus resultierende Teilungslösung. Die Teilungslösung ist unterscheidet sich von der optimalen fraktionalen Lösung nur um das Teilungsobjekt, welches aus der Teilungslösung entfernt wird. Die Teilungslösung stellt die Initiallösung für den Core-Algorithmus dar, der jene im weiteren Verlauf stetig verbessert. Aus diesem Grund ist sie besonders interessant für dieses Kapitel. Der Core-Algorithmus kann sehr viel Laufzeit einsparen, da keine Objekte außerhalb des Core betrachtet werden müssen, um eine optimale Lösung zu konstruieren.

Wie später zu sehen ist, liefern alle Algorithmen zusätzlich zum Teilungsobjekt auch eine leicht zugängliche Teilungslösung. Der Aufwand zu deren Berechnung fällt direkt beim effizienten Bestimmen des Teilungsobjekts an, sodass insgesamt Arbeit im Core-Algorithmus gespart wird. Die vorgestellten Algorithmen sind BreakitemA, ein Greedy Algorithmus der auf einer sortierten Eingabe arbeitet und eine $\mathcal{O}(n \log(n))$ Laufzeit hat, und die beiden Linearzeitalgorithmen BreakitemB und BreakitemC, welche beide eine Abwandlung einer Quicksort Implementation darstellen und sich hauptsächlich in dem Vorgehen zur Partitionierung der Eingabe unterscheiden.

3.1 Algorithmus BreakitemA

Ein naheliegender Greedy-Algorithmus sortiert die Objekte anhand ihrer Effizienz, d.h. dem Quotienten aus Nutzen und Gewicht. Dann startet er mit einem Lösungsvektor x, dessen Einträge alle 0 sind, und erweitert die Lösung solange um das nächste Objekt bis ein weiteres Objekt das Gesamtgewicht größer als die Kapazität werden lässt. Das

nächsteffizienteste Objekt k, welches nicht mehr in den Rucksack passt, kann dann zum Teil eingepackt werden, sodass die Residualkapazität zu der ganzzahligen Lösung ausgeschöpft wird.

Algorithmus 2 BreakitemA(A)

Ensure: Alle x_i sind mit 0 initialisiert
1: Sortiere die Objekte absteigend nach ihrer Effizienz
Require: $\forall i \in \{1, 2, ..., n-1\} : p_i/w_i \geq p_{i+1}/w_{i+1}$
2: $t \leftarrow W$
3: **for** i = 1, 2, ..., n **do**
4: **if** $w_i \leq t$ **then**
5: $x_i \leftarrow 1$
6: $t \leftarrow t - w_i$
7: **else**
8: $x_i \leftarrow t/w_i$
9: $t \leftarrow 0$
10: **return** i
11: **return** n+1 ▷ triaviale Lösung

Dieser Algorithmus bestimmt das Teilungsobjekt. In der Ausgabe des Algorithmus gilt folgendes:

$$\forall i \in \{1, 2, ..., k-1\} : \frac{p_i}{w_i} \geq \frac{p_k}{w_k} \tag{3.1}$$

$$\forall i \in \{k+1, k+2, ..., n\} : \frac{p_i}{w_i} \leq \frac{p_k}{w_k} \tag{3.2}$$

Im dem Core-Algorithmus werden die Objekte anhand ihres Verlusts beurteilt und der Position relativ zum Danzig Strahl. Daher soll dieser Algorithmus und alle Optimierungen die Eigenschaft behalten, dass es nach der Ausführung zwei Teilarrays gibt für die Objekte, die über respektive unter dem Danzig Strahl liegen.

Die Laufzeit beträgt $\mathcal{O}(n \log(n))$ für das Sortieren und $\mathcal{O}(n)$ für das Durchlaufen der Schleife, insgesamt ergibt sich eine Laufzeit von $\mathcal{O}(n \log(n))$.

Theorem 3.1. *[Rö14, Seite 27 ff] Der oben formulierte Algorithmus* BreakitemA *berechnet eine optimale Lösung des fraktionalen Rucksackproblems. Diese Lösung benutzt höchstens ein fraktionales Objekt, alle anderen Objekte sind entweder vollständig enthalten oder gar nicht.*

Beweis. Es wird angenommen, dass die im folgenden betrachtete Instanz ihre Objekte bereits absteigend nach Effizienz sortiert hat. Diese Eigenschaft wird in Zeile 1 hergestellt. Der von BreakitemA bestimmte Rückgabeindex wird mit k und der induzierte Lösungsvektor wird mit x bezeichnet.

Fall 1: $\sum_{i=1}^{n} w_i \leq W$
Der Algorithmus liefert als Ergebnis $n+1$ und die induzierte Lösung $x = 1^n$ ist trivial. Es werden alle x_i in der Schleife auf 1 gesetzt. Da der Nutzen dieser Lösung in dieser Probleminstanz maximal ist, kann es keine Lösung mit größerem Nutzen geben und x ist optimal. Die Lösung verzichtet auf die Verwendung eines fraktionalen Objekts.

Fall 2: $\sum_{i=1}^{n} w_i > W$
Sei $x^\star$ eine beliebige, aber feste optimale Lösung.
x hat die Struktur $x_1 = x_2 = ... = x_{k-1} = 1$, $x_k < 1$ und $x_{k+1} = x_{k+2} = ... = x_n = 0$, da die Zählschleife in BreakitemA beendet wird sobald einmal der Fall in Zeile 7 erfüllt ist. In allen vorherigen Iterationen der Schleife wird ein neuer Eintrag $x_{j<k}$ auf 1 gesetzt. Die Lösung x benutzt nur ein fraktionales Objekt x_k.
Es gilt $x^T w = W$ nach Konstruktion des Algorithmus. Ebenso gilt $(x^\star)^T w = W$, da eine Lösung, die nicht den Rucksack ausfüllt, nicht den maximal erreichbaren Nutzen hat und daher nicht optimal sein kann.
Es wird jetzt der kleinste Index i mit $x_i^\star < 1$ betrachtet.

Fall 2.1: $\forall j > i : x_j^\star = 0$
Da i der kleinste Index mit der Eigenschaft $x_i^\star < 1$ ist, gilt, dass alle $x_{j<i}^\star = 1$ sind. Da zudem gilt, dass alle $x_{j>i}^\star = 0$ sind und $x_i^\star < 1$ ist, muss die Struktur von $x^\star$ die gleich der sein, die schon der Lösung x nachgewiesen wurde.
Offen bleibt die Frage, ob $i = k$ ist. Sollte $i < k$ gelten, gilt ebenso dass $(x^\star)^T w < x^T w = W$, was ein Widerspruch zu der festgestellten Aussage ist, dass die optimale Lösung die gesamte Kapazität ausnutzt. Sollte $i > k$ gelten, gilt dann $(x^\star)^T w > x^T w = W$, was ein Widerspruch zur Gültigkeit der Lösung $x^\star$ darstellt. Also gilt $i = k$ und damit auch $x = x^\star$, was die Optimalität der Lösung x beweist.

Fall 2.2: $\exists j > i : x_j^\star > 0$
Sei j der größtmögliche solche Index für die Erfüllung der obigen Bedingung.
Aus $j > i$ folgt direkt, dass die Effizienz von i größer als die von j ist. Da $i < 1$ und $j > 0$ ist, kann ein gewisser Teil von j nach i verschoben werden bis $j = 0$ oder $i = 1$ ist. Die Menge der Verschiebung ε beträgt $min\{(1 - x_i^\star)w_i, x_j^\star w_j\}$.

Aus Fall 2 der Fallunterscheidung resultiert die konstruierte Lösung $\bar{x}$. Die Gültigkeit dieses Lösungsvektors folgt aus der Gültigkeit von $x^\star$, sowie

$$\bar{x}_i = x_i^\star + \frac{\varepsilon}{w_i} \leq x_i^\star + \frac{(1 - x_i^\star)w_i}{w_i} = x_i^\star + 1 - x_i^\star = 1$$

und

$$\bar{x}_j = x_j^\star - \frac{\varepsilon}{w_j} \geq x_j^\star - \frac{x_j^\star w_j}{w_j} = x_j^\star - x_j^\star = 0$$

Die Gültigkeit der Lösung folgt aus

$$\bar{x}^T w = (x^\star)^T w + \frac{\varepsilon}{w_i} w_i - \frac{\varepsilon}{w_j} w_j = (x^\star)^T w + \varepsilon - \varepsilon = (x^\star)^T w = W$$

sowie

$$\bar{x}^T p = (x^\star)^T p + \frac{\varepsilon}{w_i} p_i - \frac{\varepsilon}{w_j} p_j = (x^\star)^T p + \varepsilon \cdot (\frac{p_i}{w_i} - \frac{p_j}{w_j}) \geq (x^\star)^T p$$

Der Vorgang, bei dem von einem Index j aus zu einem kleineren Index i ein Anteil verschoben wird, kann beliebig oft wiederholt werden. Bei jeder dieser Transformation wird entweder der Index i auf 1 gesetzt oder der Index j auf 0. Dadurch nähern sich diese Indices an, bis die Bedingung für den Fall 2.2 nicht mehr erfüllt ist. Dann kann man auf das aus der letzten Transformation hervorgehende $\bar{x}$ die Argumente aus Fall 2.1 anwenden um zu zeigen, dass x eine optimale Lösung ist. □

3.2 Algorithmus BreakitemB

Die Bestimmung des Teilungsobjekts wird hier formal definiert. Zu einer gegebenen Instanz wird ein k gesucht, sodass $0 \leq W - \sum_{i=1}^{k-1} w_i \leq w_k$ und die Gleichungen 3.1 und 3.2 gelten.

Der Algorithmus BreakitemA zeigt gut, dass man diese Bedingungen leicht auf einer nach Effizienz sortierten Menge realisieren kann. Wenn man das Teilungsobjekt schneller als in $\mathcal{O}(n \log(n))$ bestimmen möchte, wird man auf das vorherige Sortieren der Menge verzichten müssen.

Um die Gleichungen 3.1 und 3.2 zu lösen reicht Linearzeit aus. Ein bekannter Algorithmus der genau das leistet heißt Partition und wurde von dem Informatiker und Turing-Award Preisträger C. A. R. Hoare im Jahr 1960 entworfen. Auf Partition aufbauend entwarf er zudem die beiden Algorithmen Quicksort und Quickselect, welche beide die Grundlagen für die in der Praxis schnellsten Algorithmen für das Sortierproblem und das Finden des k.-kleinsten Elements bilden. Das Sortierproblem ist eines der am besten untersuchten Probleme der Algorithmik, wodurch in vielen Forschungsbeiträgen einige Verbesserung am Algorithmus Partition vorgenommen wurden. Ebenso gibt es Erweiterungen und Optimierungen für Quickselect.

Die angesprochenen Algorithmen sind alle asymptotisch optimal, die erzielten Verbesserungen sind aber in der Praxis durchaus relevant. Die wenigsten großen Softwareprojekte kommen ohne einen Sortieralgorithmus aus. Da die Datenmengen mitunter riesige Ausmaße annehmen, spielen auch sehr kleine Verbesserungen eine große Rolle. Diese Fortschritte und Erkenntnisse fließen hier in den Algorithmus zur Bestimmung des Teilungsobjekts ein, um dieses so effizient wie möglich zu implementieren. Die fraktionale Lösung wird von der Eingabe der Länge n bestimmt, während der Core-Algorithmus nur auf einer vergleichsweise kleinen Teilmenge der Objekte arbeitet. Da dieser Aufwand die Laufzeit signifikant beeinträchtigt, ist es sinnvoll sich alle möglichen Optimierungsmöglichkeiten im Folgenden anzuschauen [Wil15].

3.2.1 Algorithmus Partition

Algorithmus 3 PARTITION(A, left, right)

Require: $left < right$
Require: $\forall i \in \{left, ..., right\} : A[left - 1] \leq A[i]$

```
 1: p ← A[right]                                   ▷ wähle Pivot
 2: i ← left − 1
 3: j ← right
 4: repeat
 5:     while A[i] ≤ p do
 6:         i ← i + 1
 7:     while A[j] > p do
 8:         j ← j + 1
 9:     if j > i then
10:         Tausche A[i] und A[j]
11:         i ← i + 1
12:         j ← j + 1
13: until j > i
14: Tausche A[i] und A[right]                      ▷ Pivot auf Endposition setzen
15: return i
```

Die Idee von PARTITION ist so simpel wie elegant. Zuerst wird das sogenannte Pivotelement gewählt. Das Array aus der Eingabe soll in zwei Teilarrays unterteilt werden, sodass alle Elemente eines Teilarrays größer bzw. kleiner als das Pivot sind. In diesem Fall soll rechts das Teilarray der größeren Elemente und links das Teilarray der kleineren Elemente stehen.

Dann beginnt man mit zwei Zeigern, die jeweils am linken und rechten Rand starten, wobei das am weitesten rechts stehende Element dem Pivotelement entspricht. Beide Zeiger werden im Laufe der äußeren Schleife solange aufeinander zu bewegt bis sie sich überkreuzen. Falls der Zeiger in der linken Hälfte auf eine Zahl kommt, die größer als das Pivotelement ist, bleibt er stehen. Für den rechten Zeiger gilt dies analog für Zahlen kleiner dem Pivot. Wenn beide Zeiger angehalten haben, werden die beiden Elemente getauscht. Da diese Elemente vorher in dem falschen Teilarray standen, sind nun beide richtig eingeordnet. Wenn die Zeiger sich überkreuzt haben, befindet sich der rechte Zeiger im linken Teilarray und umgekehrt. Dann wird der Vorgang abgebrochen und das Pivot, welches rechts außerhalb des Betrachungsraums gespeichert ist, in die mittlere Position gerückt. Eine grafische Anschauung bietet Abbildung 3.1, in der i und j die Laufindices sind und *right* der Index, an dem das Pivot gespeichert ist. Mit x ist ein Element aus dem Array A gemeint, sodass der linke und rechte Block im nachfolgenden Korrektheitsbeweis die beiden Teile der Invariante darstellen. Das Feld *unbekannt* ist die Menge aller Elemente, die noch nicht betrachtet wurden und daher die Zugehörigkeit des Teilarrays noch nicht bestimmt wurde [Wil15, Seite 29 ff].

Um die Laufzeit der Schleife zu bestimmen, reicht es, sich die Summe aller Operationen anzusehen. Da mit jeder Tauschoperation 2 Elemente richtig eingeordnet werden, reichen $\frac{n}{2} = \mathcal{O}(n)$ Vertauschungen aus. Die Zeiger haben initial einen Abstand von

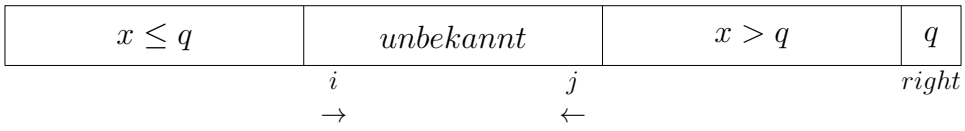

Abbildung 3.1: PARTITION mit einem Pivot

$n-1$ und mit jedem Aufruf der Rechenoperationen verringert er sich um 1. Nach n vielen Aufrufen gilt die Bedingung für die äußere Schleife nicht mehr. Insgesamt lässt sich ein Aufruf in Linearzeit bearbeiten.

Theorem 3.2. PARTITION *ist korrekt.*

Beweis. Die Korrektheit wird mit Hilfe von vollständiger Induktion über eine Invariante bewiesen. Diese wird unterteilt in

(i) $\forall x \in \{left, 2, ..., i-1\} : A[x] \leq Pivot$
(ii) $\forall x \in \{j+1, j+2, ..., right-1\} : A[x] > Pivot$

Im Induktionsanfang sind die Indexzeiger $i = left - 1$ und $j = right$. $A[i]$ ist nach Voraussetzung höchstens so groß wie das Minimum des Arrays A und damit auch nicht größer als das Pivot. Das ist nötig für das erste Ausführen von Zeile 5, da sonst ein nicht vorhandenes Element in das Array gesetzt werden würde. Die Invarianten (i) und (ii) sind erfüllt, da der Allquantor x nicht quantifizieren kann, um die Aussage zu widerlegen.

Die Induktionsvoraussetzung ist, dass die Invarianten (i) und (ii) immer zu Anfang der Iteration der äußeren Schleife gelten.

Im Induktionsschritt steht die Ausführung einer Iteration der äußeren Schleife an. Es gilt die Induktionsvoraussetzung. Nun werden die beiden inneren Schleifen ausgeführt. Diese Schleifen fügen weitere Elemente zu den in den Invarianten betrachteten Mengen hinzu. Sollte ein Element betrachtet werden, dass bei Erhöhen von i oder Verringern von j die Gültigkeit der Invariante beeinträchtigt, wird die Schleife abgebrochen. Also gilt die Invariante zu Beginn von Zeile 9 immer noch.

Die Bedingung in Zeile 9 prüft, ob die Indices i und j sich noch nicht gekreuzt haben. Sollte dies zutreffen, so ist die Menge noch nicht vollständig in die geforderten Teilarrays partitioniert. Da bereits festgestellt ist, dass beide Elemente in das jeweilig andere Teilarray gehören, reicht eine Tauschoperation aus um beide Elemente richtig zu positionieren. Da die Elemente $A[i]$ und $A[j]$ vollständig abgearbeitet sind, können die Indices angepasst werden und die Invariante gilt auch zum Ende der Iteration respektive zu Beginn der nächsten.

In dem Fall, dass die Bedingung nicht gilt, wird an dem Zustand aller Variablen nichts geändert und die äußere Schleife wird beendet. Damit gilt die Invariante nach dem Abarbeiten aller Iterationen.

Um aus der Korrektheit der Invariante die Korrektheit von PARTITION zu schließen, muss noch die Tauschoperation nach der Schleife betrachtet werden. Nach der letzten Iteration ist das Element $A[i]$ das Element im rechten Teilarray mit dem kleinsten

Index. Das Pivot war in $A[right]$ gespeichert und nimmt durch den Tausch die Position zwischen den beiden Teilarrays ein. Damit bestimmt i die Grenze der beiden Teilarrays.

□

3.2.2 Wahl des Pivots

In dem angegebenen Algorithmus wird immer das Element mit dem größten Index ($A[right]$) als Pivot gewählt. Falls eine andere Wahl als Pivotelement gewünscht wird, so lässt sich PARTITION dahingehend leicht erweitern, in dem vor dem Aufruf das entsprechende Element mit dem Element an der Position $right$ vertauscht wird.

In dem Algorithmus BREAKITEMB wird auf PARTITION zugegriffen. Die Laufzeit hängt daher direkt von der Größe der beiden Teilarrays ab, die PARTITION liefert. Der Algorithmus BREAKITEMB hat $\mathcal{O}(n)$ Kosten zusätzlich zu den Kosten des rekursiven Aufrufs. Der Algorithmus arbeitet nur auf einem der beiden Teilarrays weiter. Die Entscheidung, welches Teilarray gewählt wird, hängt von den Gewichten der Objekte ab. Da die verfügbare Residualkapazität t und die Gewichte der Objekte ebenso wie die Länge der Teilarrays diese Entscheidung beeinflussen, ist die optimale Zerlegung unklar. Eine Aufteilung von gleich großen Teillisten liefert in jedem Fall einen Linearzeitalgorithmus für das Problem.

Betrachten wir zu Beginn den Fall, dass das Pivot uniform zufällig gewählt wird. Im folgenden wird angenommen, dass der Algorithmus immer das größere Teilarray auswählt. Das verschlechtert die Laufzeit höchstens und ist somit eine Abschätzung nach oben. Die Zufallsvariable $\mathcal{X}$ wird definiert als die Größe des längeren Teilarrays.

$$\mathcal{X}_i = \begin{cases} n-i & \text{falls } i \leq \frac{n}{2} \\ i & \text{falls } i > \frac{n}{2} \end{cases}$$

Für den Erwartungswert der Zufallsvariablen gilt dann

$$\begin{aligned} E[\mathcal{X}] &= \sum_{i=0}^{\infty} \mathcal{X}_i \cdot Pr[\mathcal{X} = \mathcal{X}_i] \\ &= \sum_{i=1}^{n-1} \mathcal{X}_i \cdot Pr[\mathcal{X} = \mathcal{X}_i] \\ &\leq \sum_{i=n/2}^{n-1} i \frac{2}{n-1} \\ &= \frac{2}{n-1} \cdot \sum_{i=n/2}^{n-1} i \\ &= \frac{2}{n-1} \cdot \left(\sum_{i=1}^{n-1} i - \sum_{i=1}^{n/2} i \right) \end{aligned}$$

$$
\begin{aligned}
&= \frac{2}{n-1} \cdot \frac{(n-1)n}{2} - \frac{\frac{n}{2}(\frac{n}{2}+1)}{2} \\
&= \frac{(n^2-n) - (\frac{1}{4}n^2 + \frac{1}{2}n)}{n-1} \\
&= \frac{\frac{3}{4}n^2 - \frac{3}{2}n}{n-1} \\
&\leq \frac{\frac{3}{4}n^2 - \frac{3}{4}n}{n-1} \\
&= \frac{3}{4} \cdot \frac{n^2-n}{n-1} \\
&= \frac{3}{4}n
\end{aligned}
$$

Setzt man den Erwartungswert in eine Rekursionsgleichung ein, erhält man mit festen Konstanten c_1, c_2

$$
\begin{aligned}
T_{algo}(1) &= c_1 \\
T_{algo}(n) &= T_{algo}\left(\frac{3n}{4}\right) + c_2 n
\end{aligned}
$$

Im Master-Theorem vergleicht man die beiden Funktionen $n^{log_{\frac{4}{3}}(1)} = n^0 = 1$ sowie cn. Da hier die Funktion cn die Funktion 1 dominiert, liegt $T_{algo}(n)$ in $\mathcal{O}(n)$ nach [Rö14, Theorem 2.3].

Linearzeit kann auch mit einer deterministischen Strategie zur Wahl des Pivot erreicht werden. Der Algorithmus Median of Medians bestimmt einen approximaten Median, der ziemlich nah an dem eigentlichen Median liegt. Dadurch wird in jedem Schritt die Menge in etwa zwei gleich große Teilmengen aufgeteilt und die Rekursionsgleichung ändert sich leicht, was aber keinen Unterschied im Ergebnis macht. Statt $\frac{4}{3}$ wird überall 2 eingesetzt, die restlichen Gleichungen folgen analog.

Praktische Studien haben ergeben, dass der Aufwand für komplizierte Berechnungen wie Pseudozufallszahlen oder einen approximaten Median die dadurch ersparten Kosten nicht wieder erlangt. Eine lange bekannte und in den performantesten Implementierungen vorherrschende Strategie das Pivot zu wählen wird *pivotsampling* genannt.

Pivotsampling startet mit einer Auswahl von Kandidation für das Pivotelement. Die Anzahl der Kandidaten ist für gewöhnlich ungerade. Das Pivot wird der Median dieser Auswahl. Eine Methode, die nach dem Schema arbeitet, ist der Median of Three. Es werden drei Kandidaten gewählt, die Elemente mit den Indices $left$, $\frac{left+right}{2}$ und $right$. Median of Three wird unter anderem in der Quicksort Implementation der C Standardbibliothek verwendet und anderen konkurrierenden Implementationen.

Ein Nachteil den diese Methoden mit allen anderen festen Pivotwahlen gemeinsam hat, ist das asymptotische Verhalten der Laufzeit. Im Worst Case kann eine Eingabe konstruiert werden, sodass die drei betrachtete Kandidaten die jeweils wertkleinsten bzw. -größten sind. Die größere Liste hat dann n-viele Elemente und die Gesamtlaufzeit

ist quadratisch. Als besonders hervorzuhebende Eigenschaft von Median of Three ist das Verhalten auf bereits sortierten Eingaben. Einige in der Praxis auftretende Instanzen sind bereits zum Teil sortiert. Die feste Wahl von $A[right]$ hat auf vollständig sortierten Instanzen die schlechteste Laufzeit, während Median of Three immer gleich große Teilarrays bildet [Wil15, Seite 37 ff].

3.2.3 Pseudocode

Algorithmus 4 BreakitemB(A, $left$, $right$, t)

Der initiale Aufruf ist BreakitemB(A, 0, $n-1$, W).
Definiere $A[i] := w_i/p_i$ als Kehrwert der Effizienz des Objekts i, da Partition aufsteigend arbeitet und der Algorithmus eine absteigende Sortierung der Effizienz ewartet.

```
1: if left > right then
2:     q ← Partition(A, left, right)
3:     Wq ← Σ_{i=left}^{q} w_i
4:     if Wq > t then
5:         return BreakitemB(A, left, q − 1, t)
6:     else
7:         return (q − left + 1) + BreakitemB(A, q + 1, right, t − Wq)
8: return 0
```

Theorem 3.3. BreakitemB *ist korrekt, d.h. der Algorithmus berechnet eine optimale Lösung für das fraktionale Rucksackproblem.*

Beweis. Es wurde bewiesen, dass der vorher behandelte Algorithmus BreakitemA korrekt ist und eine optimale Lösung berechnet. Jetzt wird gezeigt, dass der Algorithmus BreakitemB die gleiche Lösung berechnet unter der Annahme, dass eine Sortierung der Objekte nach Effizienz eindeutig ist. Die Annahme ist äquivalent dazu, dass die Effizienz der Objekte paarweise verschieden und die optimale Lösung eindeutig ist.

Der Algorithmus betrachtet in allen rekursiven Aufrufen immer nur ein Teilarray der aktuellen Eingabe. Da in keinem Teilarray das Pivotelement aus Partition enthalten ist, muss das Teilarray um mindestens ein Element kleiner sein. Insgesamt muss der Algorithmus nach n vielen rekursiven Aufrufen terminieren.

Fall 1: $Wq \leq t$
Aus der Korrektheit von Partition lässt sich schließen, dass alle Elemente $\leq q$ effizienter sind als jedes Element $> q$. Die betrachteten ersten q Elemente mit der größten Effizienz im Algorithmus BreakitemA sind demnach alle in dem Teilarray $A[left, .., q]$ vorhanden. Zu diesem Zeitpunkt ist die Bestimmung der weiteren Objekte gleich einer Instanz, die genau die Objekte $\{q+1, q+2, ..., right\}$ enthält und eine Kapazität von $t - Wq$ besitzt. Der Aufruf des Teilarrays $A[q+1, ..., right]$ mit der Residualkapazität t liefert dasselbe Verhalten wie BreakitemA ab der Iteration $i = q+1$, da man wieder einen der beiden Fälle verwenden kann.

Fall 2: $Wq > t$
Die Summe der betrachteten ersten q Elemente ist größer als die Residualkapazität t. Demnach passen nicht alle diese Objekte zusammen in den Rucksack. Also gibt es mindestens ein Objekt, welches nur teilweise oder gar nicht eingepackt wird. Da die Effizienz aller Objekte $> q$ kleiner ist als dieses nach dem Argument aus Fall 1, brauchen die Objekte $> q$ nicht mehr betrachtet werden. Der Nutzen einer Lösung kann mit den Objekten $> q$ sich nur verschlechtern, weil das Gewicht weniger gewinnbringend genutzt würde.

□

Es soll nicht unerwähnt bleiben, dass es einen deutlichen Unterschied zwischen BreakitemB und BreakitemA gibt. BreakitemA bestimmt während der Bearbeitung einen Lösungsvektor x, welcher die Objekte beschreibt, die in den Rucksack gepackt werden. Das passiert in BreakitemB nicht explizit, aber eine solche Lösung kann leicht hergeleitet werden indem alle $x_{j<k} = 1$ gesetzt werden und danach $x_k = \frac{W - x^T w}{w_k}$ gesetzt wird. k ist hierbei der Rückgabewert des Algorithmus. Alternativ kann der Algorithmus auch modifiziert werden, sodass vor dem rekursiven Aufruf in Zeile 7 in einer Schleife $x_i = 1$ für alle $i \in \{left, ..., q\}$ gesetzt wird.

3.3 Algorithmus BreakitemC

R. Sedgewick arbeitet 1975 im Rahmen seiner Doktorarbeit einige Varianten von Quicksort aus und erarbeitet einen Ansatz, der zwei Pivotelemente in der Partition Methode verwendet. Seine Analyse stellt heraus, dass sein Algorithmus Quicksort nicht verbessert und der ursprüngliche Algorithmus eindeutig überlegen war. Seine Arbeit gilt dennoch als Meilenstein in der Analyse von Quicksort. P. Hennequin untersuchte später einen allgemeineren Quicksort mit $s - 1$ Pivotelementen und s vielen Teilarrays mit dem Ergebnis, dass der klassische Quicksort mit nur einem Pivotelement unerreichbar gut in der Laufzeit ist. Erst 2009 wurde eine Quicksort Variante mit zwei Pivotelementen von V. Yaroslavskiy vorgestellt, die deutlich performanter als die bis dahin besten Quicksort Implementierungen [Yar09] ist.

Praktische Studien haben vielfach gezeigt, dass der von V. Yaroslavskiy vorgeschlagene Quicksort eine signifikante Verbesserung gegenüber dem klassischen Quicksort darstellt. Eine theoretische Analyse der Laufzeit bietet S. Wild, die jedoch nicht endgültig die Verbesserung beweisen kann und weitere tiefergehende Analysen in Verbindung mit der ausführenden Architektur erfordert. Die Anzahl an nötigen Vergleichen und Vertauschungen beider Varianten bieten aus theoretischer Sicht keine Grundlage um eine Überlegenheit von Yaroslavskiys Methode zu beweisen [Wil15].

Für BreakitemB wurde bisher nur ein zusätzlicher Algorithmus zur Bestimmung einer Partition genutzt und das wird sich für diese Optimierung auch nicht ändern. Es wurde bereits angesprochen, dass der Algorithmus zum Unterteilen eines Arrays auch in Quickselect verwendet wird. Nachdem Yaroslavskiys Methode ertragreiche

Ergebnisse geliefert hat, sollte QUICKSELECT davon profitieren. Eine genauere Analyse hat jedoch ergeben, dass die positiven Auswirkungen auf QUICKSORT nicht übertragen wurden. Gründe hierfür sind die erwarteten Vergleiche und Vertauschungen, die nach [WNM13] beim Ausführen anfallen.

3.3.1 Verallgemeinerung an Partition

Der Algorithmus DUALPIVOTPARTITION ist eine Variante von PARTITION, in der statt einem nun zwei Pivotelemente benutzt werden. In der einfachen Variante starten zwei Zeiger an beiden Rändern des Eingabearrays und laufen solange in die Richtung des anderen Endes bis sich die Zeiger gekreuzt haben. Sobald mehr als zwei Teilarrays erzeugt werden sollen, muss die Anzahl der Zeiger erhöht werden. Da es aber nur zwei Enden eines Arrays gibt, lässt Yaroslavskiy an der einen Seite einen zusätzlichen Zeiger starten. Beide Zeiger auf der einen Seite werden nur dann gleichzeitig weiter gesetzt, wenn das betrachtete Element kleiner als beide Pivots sind. Wenn das betrachtete Element zwischen beiden Pivots liegt, wird nur der innere erhöht. Allgemeiner gesprochen werden die beiden Pivotelemente, zwischen denen das Element liegt, gesucht und dann werden alle Pivotelemente der entsprechenden Seite ab dem inneren Element in Richung der gegenüber liegenden Seite verschoben. Eine Veranschaulichung von DUALPIVOTPARTITION bietet Grafik 3.2.

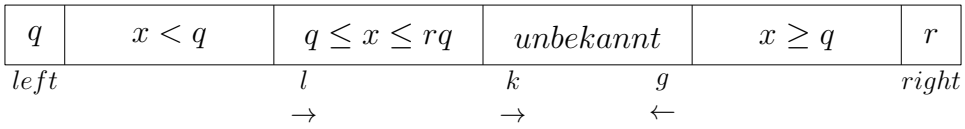

Abbildung 3.2: Partition mit zwei Pivots

Eine vollständige interaktive Visualisierung findet sich unter [Lyo].

Hinzu kommen außer den Änderungen am Vorgehen mit mehr als einem Pivotelement noch Änderungen an der Wahlstrategie für die Pivotelemente. Im Abschnitt 3.2.2 wurden einige Methoden dargestellt wie man das Pivotelement im klassischen QUICKSORT wählt. Besonders in der Praxis und daher auch in der Implementation von MEDIAN OF THREE verwendet. In der Methode werden drei Kandidaten für das Pivot von festen Positionen ausgewählt und davon der Median genommen.

Da jetzt statt einem direkt zwei Pivotelemente gewählt werden müssen, wird der Ansatz verallgemeinert. Man hat immer den Median gewählt, da dann auf den Seiten der Kandidatenliste, die größere und kleinere Seite, gleich viele Elemente sind. Bei zwei Pivots sollen dann in den Zwischenräumen, welche zwischen Pivotelementen und den äußeren Rändern entstehen, gleich viele nicht gewählte Kandidaten liegen. Diese Zwischenräume sind in Abbildung 3.2 als Felder visualisiert. Da ein großer Vorteil von MEDIAN OF THREE in dem geringen Aufwand besteht, werden in diesem zwei Pivotelemente gewählt mit nur einem Kandidaten für jeden der drei Teile. In der ersten vorgestellten Implementation wurden die 5 Elemente so gewählt, dass $Kandidati$ den Index $left + i \cdot \frac{length}{6}$ hat. Diese Kandidaten werden dann sortiert, sodass der wertkleinste Kandidat an die Stelle von $Kandidat1$ tritt, der nächstkleinere dann an

die Position des 2. Kandidaten und der Rest analog. Die Pivots p und q sind dann $Kandidat2$ und $Kandidat4$. Im Anschluss werden die Kandidaten dann mit den Positionen $left$ und $right$ vertauscht [Yar09].

3.3.2 Pseudocode

Algorithmus 5 DualPivotPartition(A, $left$, $right$)

Require: $left < right$
1: $p \leftarrow A[left]$
2: $q \leftarrow A[right]$
3: **if** $p > q$ **then** ▷ p ist das kleinere Pivot
4: Tausche p und q
5: $less \leftarrow left + 1$
6: $great \leftarrow right - 1$
7: $k \leftarrow less$
8:
9: **while** $k \leq great$ **do**
10: **if** $A[k] < p$ **then**
11: Tausche $A[k]$ und $A[less]$
12: $less \leftarrow less + 1$
13: **else if** $A[k] \geq q$ **then**
14: **while** $A[great] > q$ und $k < great$ **do**
15: $great \leftarrow great - 1$
16: Tausche $A[k]$ und $A[great]$
17: $great \leftarrow great - 1$
18: **if** $A[k] < p$ **then**
19: Tausche $A[k]$ und $A[less]$
20: $less \leftarrow less + 1$
21: $k \leftarrow k - 1$
22: $less \leftarrow less - 1$ ▷ Setzte beide Pivots auf Endposition
23: $great \leftarrow great + 1$
24: Tausche $A[left]$ und $A[less]$
25: Tausche $A[right]$ und $A[great]$
26: **return** ($less$, $great$)

In DualPivotPartition gibt es nicht nur 2 sondern 3 Teilarrays zu betrachten. Daher müssen wir den Algorithmus BreakitemC so gestalten, sodass dieser alle drei Teilarrays berücksichtigt. Einen vollständigen Beweis der Korrektheit gibt es hier nicht, da die Ideen exakt dieselben sind, die schon in dem Algorithmus mit nur einem Pivot benutzt wurden. Der Beweis lässt sich analog führen, insbesondere da die Argumente keine Eigenschaften wie die Anzahl der vom Algorithmus induzierten Teilarrays benutzt haben.

Algorithmus 6 BreakitemC(A, $left$, $right$, t)

Der initiale Aufruf ist BreakitemC(A, 0, $n-1$, W).
Definiere $A[i] := w_i/p_i$ als Kehrwert der Effizienz des Objekts i, da Partition aufsteigend arbeitet und der Algorithmus eine absteigende Sortierung der Effizienz ewartet.

```
1: q, r ← YaroslavskiyPartition(A, left, right)
2: Wq ← Σ_{i=left}^{q} w_i
3: if Wq > t then
4:     return BreakitemC(A, left, q − 1, t)
5: Wr ← Σ_{i=q+1}^{r} w_i
6: if Wq + Wr > t then
7:     return (left − q + 1) + BreakitemC(A, q + 1, r − 1, t − Wq)
8: return (left − r + 1) + BreakitemC(A, r + 1, right, t − Wq − Wq)
```

3.3.3 Erwartungen

Aus der oben zitierten Analyse von Quickselect mit zwei Pivotelementen geht hervor, dass sich die Kosten beim Partitionieren bei zwei Pivotelementen erhöht haben. Da sich der Algorithmus von BreakitemC außer den zusätzlichen Berechungen nicht unterscheidet, werden diese auch diesen betreffen. Ungeachtet der möglichen Verschlechterung der Laufzeit werde ich BreakitemC untersuchen, um herauszufinden, ob es vorteilhaft ist mit zwei Pivotelementen zu arbeiten. Die Gründe hierfür liegen in der Hoffnung, dass die unterschiedliche Struktur der Algorithmen mit kleineren Teilproblemen Nachteile bei den Partitionskosten ausgleichen. Sowohl bei Quicksort als auch bei Quickselect fällt der Großteil des Arbeitsaufwands beim Partitionieren des Arrays an. Nach dem Partitionieren reichen zwei rekursive Aufrufe bzw. ein rekursiver Aufruf und eine Bedingung. In beiden Fällen sind die Kosten sehr gering. Bei BreakitemB hingegen steht nach dem Aufruf von Partition noch das Berechnen einer Summe aus, die aus n vielen Summanden besteht. Der neue Algorithmus BreakitemC verwendet den Algorithmus DualPivotPartition von Yaroslavskiy.

Lemma 3.4. *[NW14, Abschnitt B.2] Eine zufällige Verteilung der Eingabe wird durch* Partition *oder* DualPivotPartition *nicht beeinflusst, sodass bei den sich ergebenden Teilarrays wieder von zufälligen Eingaben gesprochen werden kann.*

In der anschließenden Analyse sollen die Kosten von BreakitemB und BreakitemC bestimmt werden. Dieser Punkt ist wichtig, da sonst bei den rekursiven Aufrufen nicht mehr von zufälligen Eingaben gesprochen werden kann.

Theorem 3.5. *Die Kosten der Laufzeit von* BreakitemB *und* BreakitemC *sind im Erwartungswert gleich $k + n$, wobei k die Anzahl der in die Lösung einbezogenen Objekte und n die Anzahl der Objekte ist.*

Beweis. Zuerst wird mit ein paar gemeinsamen Punkten in der Kostenberechnung begonnen. Es wird die Anzahl der aufsummierten Gewichte betrachtet. Diese Summanden werden in zwei Gruppen aufgeteilt. Die erste Gruppe besteht aus den Summanden,

die zu einem Objekt der optimalen fraktionalen Lösung gehören. Die Anzahl der Vorkommen in einer Addition eines einzelnen Summanden ist 1 plus die Anzahl von einzelnen Aufrufen der Teilarrays, in denen dieser Summand enthalten ist. Die Konstante 1 stammt aus dem Fakt, dass der Algorithmus beim Akzeptieren der Summe von $left$ bis q bzw. $q+1$ bis r alle diese Objekte einpackt und das Gewicht mindestens in einem Aufruf aller dieser Summanden einmal vorkommen muss. Die berechnete Summe hat nach dem Bestimmen des richtigen Teilarrays keine Relevanz mehr, ein mehrmaliges Neuberechnen ist erforderlich. Daher kann ein bestimmter Summand häufig konsultiert werden. Die zweite Gruppe besteht aus allen anderen Summanden. Die Kosten unterscheiden sich nur darin, dass sie nicht zwingend einmal vorkommen und so die konstanten Kosten wegfallen.

Es wird nun die Summe aller Einzelkosten betrachtet. Um jene abzuschätzen hilft es die Gesamtkosten nach allen Schritten zu berechnen. Hier wird klar, dass sich alle konstanten Kosten auf k aufsummieren, wobei k der Rückgabeindex des Algorithmus ist. Dann wird angenommen, dass das Array der Eingabe uniform zufällig ist und in jedem Schritt in zwei bzw. drei gleiche große Teilarrays aufgeteilt wird, da die Anordnung der Zahlen zufällig ist und daher auch die Wahl des oder der Pivotelemente. Darüber hinaus sind nach Lemma 3.4 alle diese Teilarrays wieder Eingaben zufälliger Zahlen. Die Kosten sind dann die Vorkommen aller rekursiven aufgerufenen Summanden, die nicht direkt zur Bestimmung der Lösung gebraucht werden.

Den Anfang macht BreakitemB, da hier nur zwei statt drei Teilarrays und in jedem rekursiven Aufruf genau eine Summe entsteht. Das vereinfacht eine genauere Analyse. Die konstanten Kosten sind mit k oben beziffert. In jedem Schritt wird immer eine Summe von $q - left$ vielen Objekten bestimmt. Die Kosten sind pro rekursiven Aufruf $\frac{n}{2}$, da Partition im Erwartungswert beide Teilarrays gleich groß werden lässt. Das Pivot q liegt in der Mitte beider Teilarrays. Es gibt nun zwei Fälle zu unterscheiden, $Wq > t$ und $Wq \leq t$. Im Fall $Wq \leq t$ sind die Kosten für das Summieren in den konstanten Kosten abgedeckt. Im $Wq > t$ kommen diese Kosten dazu. Da die Eingabe uniform zufällig ist, treten beide Fälle mit gleicher Wahrscheinlichkeit auf.

Insgesamt sind die erwarteten Kosten für BreakitemB

$$
\begin{aligned}
& k + \sum_{i=0}^{\log_2(n)} \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot \frac{n}{2^i} \\
&= k + \frac{n}{2} \cdot \sum_{i=0}^{\log_2(n)} \left(\frac{1}{2}\right)^i \\
&= k + \frac{n}{2} \cdot \frac{1 - \frac{1}{2}^{\log_2(n)+1}}{1 - \frac{1}{2}} \\
&= k + n \left(1 - \frac{1}{2}^{\log_2(n)+1}\right) \\
&= k + n - n \cdot \frac{1}{2}^{\log_2(n)+1} \\
&= k + n - n \cdot 2^{\log_2(\frac{1}{2})(\log_2(n)+1)} \\
&= k + n - n \cdot n^{\log_2(\frac{1}{2})} \cdot 2^{\log_2(\frac{1}{2})}
\end{aligned}
$$

$$= k + n - \frac{1}{2}$$

Die Kosten für BreakitemC sind ähnlich zu bestimmen. Die konstanten Kosten sind auch hier k, aber es gibt einen zusätzlichen Fall. Der Fall $Wq > t$ resultiert hier in Kosten $\frac{n}{3}$, da das erste Teilarray nicht mehr halb sondern nur noch ein Drittel so groß wie das Eingabearray des Aufrufs ist. Fall $Wq \leq t \wedge Wr > t$ hat Kosten von $\frac{n}{3}$ für Wq und $\frac{n}{3}$ für Wr, die ersten Kosten stecken bereits in den gesamten Kosten und brauchen nicht mehr berücksichtigt zu werden. Die Kosten für den letzten Fall $Wr \leq t$ sind bereits durch die Konstante k abgedeckt.

$$\begin{aligned}
& k + \sum_{i=0}^{\log_3(n)} \frac{1}{3} \cdot 0 + \frac{1}{3} \cdot \frac{n}{3^i} + \frac{1}{3} \cdot \frac{n}{3^i} \\
&= k + \frac{2n}{3} \cdot \sum_{i=0}^{\log_3(n)} \left(\frac{1}{3}\right)^i \\
&= k + \frac{2n}{3} \cdot \frac{1 - \frac{1}{3}^{\log_3(n)+1}}{1 - \frac{1}{3}} \\
&= k + n \left(1 - \frac{1}{3}^{\log_3(n)+1}\right) \\
&= k + n - n \cdot \frac{1}{3}^{\log_3(n)+1} \\
&= k + n - n \cdot 3^{\log_3(\frac{1}{3})(\log_3(n)+1)} \\
&= k + n - n \cdot n^{\log_3(\frac{1}{3})} \cdot 3^{\log_3(\frac{1}{3})} \\
&= k + n - \frac{1}{3}
\end{aligned}$$

Die Kosten beider Algorithmen entsprechen $k + n$ abzüglich dem Bruchteil eines Summanden. Da eine Summe eine natürliche Anzahl an Summanden hat und der Unterschied $\frac{1}{6}$ Summand beträgt, kann der Bruchteil vernachlässigt werden und die Kosten aufgerundet werden. Damit sind die Kosten exakt gleich. □

Betrachtet man nun die Kosten, die bei DualPivotQuickselect zuzüglich zu den Partitionierungskosten auftreten, erhält man folgende Rekursionsgleichung mit Konstante c

$$\begin{aligned}
T_{select}(1) &= \Theta(1) \\
T_{select}(n) &= T\left(\frac{n}{3}\right) + c
\end{aligned}$$

Nach dem Anwenden des Master-Theorems auf die Funktionen $n^{\log_3(1)} = n^0 = 1$ und c liegen die Gesamtkosten der Laufzeit ohne die Aufrufe von DualPivotPartition $T_{select}(n)$ in $\mathcal{O}(\log(n))$.

Analog gilt für DualPivotQuicksort die Rekursionsgleichung

$$T_{sort}(1) = \Theta(1)$$
$$T_{sort}(n) = 3T\left(\frac{n}{3}\right) + c$$

Somit sind die Kosten nach dem Master-Theorem $\mathcal{O}(n)$. Die Rekursionsgleichungen und somit auch die Kosten beider Algorithmen ändern sich nicht, wenn stattdessen nur mit einem Pivotelement partitioniert wird.

Vergleicht man die drei Algorithmen, die auf DualPivotPartition basieren, erkennt man jeweils Ähnlichkeiten zwischen BreakitemC und den beiden bekannten Algorithmen. Es findet wie bei DualPivotQuickselect nur ein rekursiver Aufruf statt, sodass diese beiden Algorithmen beide eine lineare Laufzeit besitzen. Betrachtet man jedoch die Kosten, die anfallen, wenn die externen Aufrufe zum Partitionieren weggelassen werden, braucht hier BreakitemC deutlich länger als DualPivotQuickselect. BreakitemB und BreakitemC haben beide Kosten von $k + n = \mathcal{O}(n)$, sodass es sich auszahlen könnte, dass kleinere Teilprobleme gegen etwas höhere Kosten beim Partitionieren eingetauscht werden.

Kapitel 4

Nemhauser-Ullmann Algorithmus

In diesem Kapitel wird der Nemhauser-Ullmann Algorithmus behandelt, welcher Instanzen des Rucksackproblems löst. Im Folgenden bezeichnen wir mit einer Lösung x einen Vektor $x \in \{0,1\}^n$. Eine Lösung ist gültig, wenn $w^T x \leq W$ ist und sie enthält ein Objekt i, genau dann wenn $x_i = 1$ ist.

Um das Rucksackproblem zu lösen, bietet sich die naheliegende Möglichkeit alle gültigen Lösungen zu bestimmen und den größten Gesamtnutzen auszugeben. Da jede Lösung x insgesamt n Objekte zur Auswahl hat und x_i die Werte 0 oder 1 annehmen kann, gibt es insgesamt $|\{0,1\}|^n = |2^n|$ viele Lösungen. Bei diesem Vorgehen wächst die Laufzeit exponentiell. Daher ist dieser naive Algorithmus in der Praxis nicht einsetzbar. An dieser Stelle kommen Filter zum Einsatz um die Anzahl der für die Lösung relevanten Vektoren zu verringern.

Definition 4.1. *Eine Lösung y dominiert x, wenn $p^T y \geq p^T x$ und $w^T y \leq w^T x$ und eine der Ungleichungen streng ist. Eine Lösung x heißt pareto-optimal, wenn sie von keiner Lösung dominiert wird. Die Pareto-Menge ist die Menge aller pareto-optimalen Lösungen.*

Diese Definition fordert nicht, dass die Lösung gültig ist in Bezug auf die Gewichtskapazität. Diese Definition legt die Grundlage für den Nemhauser-Ullmann Algorithmus. Mit diesem Dominanzkriterium können Lösungen einfach und effektiv auf Relevanz geprüft werden.

Lemma 4.2. *[Rö15, Lemma 3.2] Es gibt eine optimale Lösung, welche pareto-optimal ist.*

Beweis. Sei x eine beliebige optimale Lösung. Wenn x nicht pareto-optimal ist, muss es eine Lösung y geben, welche x dominiert, d.h. $p^T y \geq p^T x$ und $w^T y \leq w^T x$ und eine Ungleichung ist streng.

Die Lösung y ist gültig, da x eine gültige Lösung ist und $w^T y \leq w^T x \leq W$ gilt. Der Nutzen der Lösung y kann nicht größer sein als der von x, da x optimal ist. Dann muss $p^T y = p^T x$ gelten. Da die Lösung y die Lösung x dominiert, gilt $w^T y < w^T x$.

Aus der Argumentation ist y eine optimale Lösung. Falls y ebenfalls nicht pareto-optimal ist, kann das Vorgehen genau so auch auf y angewendet werden. Mit der entstehenden Lösung wird so verfahren bis eine Lösung nicht mehr dominiert wird und somit dann pareto-optimal ist. Da es maximal 2^n viele Lösungen gibt, erhält man nach endlich vielen Anwendungen eine optimale Lösung mit dem kleinsten Gewicht, welche dann pareto-optimal ist. □

Der Nemhauser-Ullmann Algorithmus bestimmt die Pareto-Menge und findet in dieser dann in Linearzeit die optimale Lösung. Der Algorithmus basiert auf dynamischer Programmierung und berechnet in jedem Schritt die Pareto-Menge $\mathcal{P}_i = \{x \mid x \text{ pareto-optimal}, x_k = 1 \iff k \leq i\}$, welche nur aus Lösungen der ersten i Objekte besteht. Die Menge $\mathcal{P}_{i+1}$ kann aus der bekannten Menge $\mathcal{P}_i$ bestimmt werden. Die zuletzt berechnete Menge $\mathcal{P}_n = \mathcal{P}$ enthält dann eine optimale Lösung.

Algorithmus 7 Nemhauser-Ullmann Algorithmus

1: $\mathcal{P}_0 \leftarrow \{0^n\}$
2: **for** i = 1, 2, ..., n **do**
3: $\quad \mathcal{Q}_i \leftarrow \mathcal{P}_{i-1} \cup \mathcal{P}_{i-1}^{+i}$
4: $\quad \mathcal{P}_i \leftarrow \{x \in \mathcal{Q}_i \mid \nexists y \in \mathcal{Q}_i : y \text{ dominates } x\}$
5: **return** $x^\star \leftarrow \arg\max_{x \in \mathcal{P}_n}\{p^T x \mid w^T x \leq W\}$

Begonnen wird mit der trivialen Paretomenge $\mathcal{P}_0$, welche nur die triviale Lösung enthält. Alle weiteren Paretomengen $\mathcal{P}_i$ werden aus dem direkten Vorgänger $\mathcal{P}_{i-1}$ und der Hinzunahme des i. Objekts konstruiert. Der Algorithmus erstellt zuerst die Menge $\mathcal{Q}_i$ in Zeile 3. Diese Menge besteht aus allen Lösungen aus den beiden Mengen $\mathcal{P}_{i-1}$ und $\mathcal{P}_{i-1}^{+i}$. Die Menge $\mathcal{P}_{i-1}$ ist die alte Paretomenge der vorherigen Iteration. Die Menge $\mathcal{P}_{i-1}^{+i}$ ist eine modifizierte Paretomenge, welche jede Lösung um das Objekt i erweitert. Nach Definition haben die Lösungsvektoren zu den Lösungen in $\mathcal{P}_{i-1}$ nur Einträge gleich 1, die zu den ersten $i-1$ Objekten gehören. Der i. Eintrag aller Vektoren aus $\mathcal{P}_{i-1}$ ist demnach 0 und der i. Eintrag aller Vektoren aus $\mathcal{P}_{i-1}^{+i}$ ist 1. Daher sind die Paretomengen disjunkt und die Kardinalität von $\mathcal{Q}_i$ ist doppelt so groß wie von $\mathcal{P}_{i-1}$.

In Zeile 4 wird dann aus $\mathcal{Q}_i$ die Paretomenge $\mathcal{P}_i$ bestimmt. Im naiven Algorithmus zur Bestimmung aller möglichen Lösungen sind diese beiden Mengen gleich. Im Nemhauser-Ullmann Algorithmus wird dann auf die Lösungen aus $\mathcal{Q}_i$ das Dominanzkriterium angewendet um diese zu filtern.

4.1 Verwaltung der Paretomengen

Der Algorithmus verwaltet die Paretomengen in jeweils einzelnen Arrays, d.h. jede Paretomenge entspricht einem Array. Da Arrays immer eine feste Größe haben und in CONSTRUCT nicht zu Beginn klar ist, wie groß die zu erstellende Paretomenge ist, muss der Speicher konservativ doppelt so groß sein die wie die vorherige Paretomenge. Da die tatsächliche Paretomenge aber nicht doppelt so groß ist, muss zusätzlich noch die Länge gespeichert werden.

Um eine Lösung zu speichern ist für gewöhnlich ein Lösungsvektor der Länge n nötig, welcher an sich als binäres Array dargestellt werden kann. Die gesamte Größe der vollständigen Paretomenge ist dann ein Array von Arrays und ein einfaches Kopieren hat bereits eine quadratische Laufzeit. Eine Alternative besteht darin statt des Lösungsvektors nur die Summe aller Nutzen $p^T x$ und Gewichte $w^T x$ zu speichern. Dann sinkt die Größe pro Lösung von $\mathcal{O}(n)$ auf $\mathcal{O}(1)$, wenn uniformes Kostenmaß angenommen wird. Die Summe der Nutzen und Gewichte steigt mit der Anzahl von Objekten in der entsprechenden Lösung an, sodass diese Zahl nicht durch eine Konstante beschränkt ist und bei großem n langsam steigt.

Diese Objekte sind Paretopunkte. Ein weiterer Vorteil den die Paretopunkte bieten, liegt in der Behandlung von mehreren identischen Lösungen. Wenn es zwei Lösungen mit dem gleichen Nutzen und Gewicht aber anderer Objektwahl gibt, reicht ein Paretopunkt aus um beide Lösungen zu repräsentieren. Dabei geht die Information verloren wie sich eine Lösung zusammen setzt.

Um aus den Paretopunkten, die berechnet werden, wieder eine vollständige Lösung zu gewinnen, muss man die Paretomengen rückwärts betrachten. Ein einfaches Zurückverfolgen reicht dabei aus. Dafür werden die n Paretomengen untersucht, die im Nemhauser-Ullmann Algorithmus erstellt werden. Man fängt mit der Paretomenge $\mathcal{P}_n$ an, in der die optimale Lösung y als Paretopunkt vorliegt. Dann sucht man die vorherige Paretomenge $\mathcal{P}_{n-1}$ und das Objekt n, welches zur Konstruktion von $\mathcal{P}_n$ geführt hat. Aus dem Punkt y wird y' konstruiert, indem der Nutzen um den Nutzen des Objekts n reduziert wird und mit dem Gewicht ebenso verfahren wird. Dann kann man mittels binärer Suche $\mathcal{P}_{n-1}$ durchmustern, ob der Punkt y' vorhanden ist. Falls dem so ist, gehört das Objekt n zur Lösung. Sonst kann die Lösung nicht mithilfe dieses Objekts entstanden sein. Im Erfolg wird mit y' und sonst mit y weiter gearbeitet. Dann führt man das so lange weiter aus bis man nur noch die triviale Lösung 0^n übrig hat. In $\mathcal{P}_0$ ist nur noch diese Lösung vorhanden, sodass das Vorgehen immer zu einem Ergebnis führt. Die Verwendung der binären Suche ist möglich, da in der effizienten Implementierung des Algorithmus alle Paretomengen sortiert gespeichert werden.

Die Laufzeit für die Rekonstruktion der Lösung ist $\mathcal{O}(n \log(|\mathcal{P}_n|))$. Es sind n viele Schritte nötig, die jeweils ein Objekt und eine Paretomenge verarbeiten. Ein einzelner Schritt braucht mit der binären Suche nur logarithmische Laufzeit der Paretomenge. Da im Schnitt jede Paretomenge mehr Elemente als ihre Vorgängermenge hat, ist $|\mathcal{P}_n|$ eine obere Schranke für die Kardinalität einer jeden Paretomenge.

4.2 Effiziente Implementierung

Damit dieser Schritt effizient gelöst werden kann, sollen die Lösungen der Mengen nach dem Gewicht sortiert werden. Wenn $\mathcal{Q}_i$ sortiert ist, reicht es aus, das Dominanzkriterium einer Lösung auf alle Lösungen kleineren Gewichts anzuwenden. Alle Lösungen, die ein größeres Gewicht haben, können die betrachtete Lösung nicht dominieren. Eine nützliche Eigenschaft der nach Gewicht sortierten Paretomenge ist, dass der Nutzen monoton steigend ist, was direkt aus der Definition folgt. Dadurch genügt es sogar,

sich nur den Nutzen der Lösung anzuschauen, die bereits zur Paretomenge gehört. Der Schritt zum Filtern der Menge $\mathcal{Q}_i$ lässt sich demnach in $\mathcal{O}(|\mathcal{P}_i|)$ ausführen.

In Zeile 3 kann man die Menge $\mathcal{Q}_i$ in $\mathcal{O}(|\mathcal{P}_i|)$ trivial bestimmen, in dem die beiden Mengen $\mathcal{P}_{i-1}$ und $\mathcal{P}_{i-1}^{+i}$ einfach hintereinander gehängt werden. Um die resultierende Menge jedoch sortiert zu lassen, reicht das nicht aus. Es ist jedoch bekannt, dass diese Mengen bereits sortiert sind. Um diese zu einer sortierten Menge zu vereinigen, kann der Schritt aus dem Sortieralgorithmus MERGESORT übernommen werden, welcher genau dies leistet [Rö14, Seite 13 ff].

Ein signifikater Unterschied ist jedoch, dass man im Nemhauser-Ullmann Algorithmus zwei sehr ähnliche Mengen hat. Daher wird der Algorithmus CONSTRUCT die Zeilen 3 und 4 im Pseudocode nicht getrennt sondern in einem Schritt ausführen und dabei das an manchen Stellen unnötige Kopieren und Modifizieren der Lösungen einsparen. Die Erstellung eines konkreten Objekts einer Lösung erfolgt daher erst, wenn klar ist, dass es auch Element der Menge $\mathcal{P}_i$ ist. Um das zu realisieren werden Zeiger verwaltet, die sich auf ein Element der Paretomenge $\mathcal{P}_{i-1}$ bzw. $\mathcal{P}_i$ beziehen.

Die Zeiger, welche sich ein entsprechendes Objekt merken, heißen id_1, id_2 und id_3. id_1 und id_2 zeigen auf ein Objekt aus der Menge $\mathcal{P}_{i-1}$ und id_3 immer auf das zuletzt eingefügte Objekt der Menge $\mathcal{P}_i$, welche gerade konstruiert wird. id_1 merkt sich eine reguläre Lösung der $\mathcal{P}_{i-1}$, welche das geringste Gewicht unter den noch nicht betrachteten hat. id_2 merkt sich eine Lösung aus $\mathcal{P}_{i-1}$ zu der noch der Nutzen und das Gewicht des i. Objekts hinzugerechnet wird. Diese Lösung gehört dann zu $\mathcal{P}_{i-1}^{+i}$ und ist als Paretopunkt noch nicht gespeichert. Sowohl id_1 und id_2 starten beim kleinsten Index 0 und werden ausschließlich inkrementiert, bis sie das Ende der Paretomenge $\mathcal{P}_{i-1}$ erreicht haben.

Im weiteren Verlauf wird dann geprüft, ob die Lösungen, die aus id_1 und id_2 hervor gehen, pareto-dominiert werden. Sollte das der Fall sein, werden die Lösungen übersprungen und nicht in die neue Paretomenge eingefügt. Sollten beide Lösungen nicht von den bisher bekannten pareto-optimalen Lösungen dominiert werden, wird die Lösung mit dem geringeren Gewicht in die Stelle von id_3 eingefügt. Die verwendeten Indices werden dann verschoben.

CONSTRUCT besteht aus mehreren Schleifen. In jeder Iteration einer jeden Schleife wird neben einigen anderen Operationen entweder id_1 oder id_2 erhöht. Insgesamt erhöht der Algorithmus id_1 und id_2 genau $|\mathcal{P}_i|$ mal. Da jede Operation einer Inkrementierung hinzugerechet werden kann, ergibt sich eine Laufzeit von $\Theta(|\mathcal{P}_i|)$ für CONSTRUCT.

Die Gesamtlaufzeit des Nemhauser-Ullmann Algorithmus lässt sich darüber definieren. In jeder Iteration der Schleife wird für eine Paretomenge und ein Objekt der Algorithmus CONSTRUCT aufgerufen. Da bereits bei der Verwaltung der Datenstrukturen die monoton steigende Kardinalität der Paretomengen beschrieben wurde, folgt das Korollar.

Korollar 4.3. *Die Laufzeit des Algorithmus liegt in* $\mathcal{O}(\sum_{i=1}^{n} |\mathcal{P}_i|) = \mathcal{O}(n \cdot |\mathcal{P}_n|)$.

Es ist unklar, inwiefern die Kardinalität der Paretomengen mit der Anzahl der Objekten korreliert. Es ist leicht eine Instanz zu konstruieren, in welcher die finale Pa-

Algorithmus 8 CONSTRUCT

Der Algorithmus nimmt der Einfachheit halber paarweise verschiedene Nutzen und Gewichte der Lösungen an.
Alle Indexvariablen id_1, id_2, id_3 sind mit 0 initialisiert.

$maxProfit \leftarrow -\infty$
while id_1 und id_2 gültige Indices **do**
  while $\mathcal{P}_{i-1}^{id_1}[p] \leq maxProfit$ **do**
    erhöhe Index id_1

  while $\mathcal{P}_{i-1}^{id_2}[p] + Item_i[p] \leq maxProfit$ **do**
    erhöhe Index id_2

  if $\mathcal{P}_{i-1}^{id_1}[w] \leq \mathcal{P}_{i-1}^{id_2}[w] + Item_i[w]$ **then**
    $\mathcal{P}_i^{id_3} \leftarrow \mathcal{P}_{i-1}^{id_1}$
    passe $maxProfit$ an
    erhöhe die Indices i_1, i_3
  else
    $\mathcal{P}_i^{id_3} \leftarrow \mathcal{P}_{i-1}^{id_2} + Item_i$
    passe $maxProfit$ an
    erhöhe die Indices i_2, i_3

while id_1 gültig **do**
  if $\mathcal{P}_{i-1}^{id_1}[p] > maxProfit$ **then**
    $\mathcal{P}_i^{id_3} \leftarrow \mathcal{P}_{i-1}^{id_1}$
    passe $maxProfit$ an
    erhöhe Index i_3
  erhöhe id_1

while id_2 gültig **do**
  if $\mathcal{P}_{i-1}^{id_2}[p] + Item_i[p] > maxProfit$ **then**
    $\mathcal{P}_i^{id_3} \leftarrow \mathcal{P}_{i-1}^{id_2} + Item_i$
    passe $maxProfit$ an
    erhöhe Index i_3
  erhöhe id_2

retomenge 2^n viele Lösungen enthält. Hierzu reicht es aus die Objektmenge $\{(2^i, 2^i) \mid i \in [n]\}$ als Eingabe zu nehmen. Jede Lösung aus einer Kombination dieser Objekte hat genau so viel Gewicht wie Nutzen. Eine Lösung kann aber nur pareto-dominiert werden, wenn es eine andere Lösung mit weniger Gewicht und mehr Nutzen gibt. Das lässt diese Konstruktion nicht zu. Jede Instanz in der alle Lösungen pareto-optimal sind, verhält sich identisch.

Eine probabilistische Analyse der Laufzeit kommt jedoch zu einer wesentlich besseren Laufzeit. Eine solche Analyse arbeitet etwas anders als eine typische Worst Case Analyse. Hierbei wird mit einer belieben Instanz gestartet, die absichtlich schlecht gewählt werden darf, ähnlich des worst case. Dann werden die Zahlen der Eingabe leicht verwackelt, d.h. zufällig um einen geringen Wert geändert. Diese Instanz wird dann untersucht. Das ist sehr viel näher an der Praxis, da in der Realität z.B. Rundungsfehler bei großen Zahlen im Gleichkommaformat vorkommen. Eine wie beschrieben untersuchte Instanz mit n vielen Objekten hat eine erwartete Laufzeit von $\mathcal{O}(n^3\phi)$. Der Faktor ϕ gibt an, wie viel Einfluss der Zufall auf die Eingabe nimmt. Falls ϕ sehr groß ist, nähert sich die Analyse der worst case Analyse an. Falls ϕ sehr klein ist, nähert man sich vollständig zufälligen Instanzen an [Rö15, Korollar 3.11].

4.3 Einbettung in ExpandingCore

In der Pareto-Definition 4.1 wird nicht gefordert, dass die Lösung gültig ist in Bezug auf die Gewichtskapazität oder der Verlust geringer als der Nutzenabstand einer optimalen Lösung. Die Lösungsmenge kann stärker eingeschränkt werden als die Definition es fordert.

Die Gewichtskapazität ist eine Beschränkung des maximalen Gewichts, das eine Lösung haben darf. Diese Ungleichung aus der Problemspezifikation muss eingehalten werden. Für die Objekte war gefordert, dass sowohl der Nutzen als auch das Gewicht größer als 0 sind. Daher kann ein Gewicht, welches einmal über der Gewichtskapazität liegt, nicht durch Hinzufügen weiterer Objekte gesenkt werden. Ein wichtiger Punkt ist, dass nicht die Kapazität des Core sondern der Instanz zugrunde gelegt wird, da die Kapazität des Core im Verlauf von ExpandingCore ansteigen kann. Eine solche Filterung sollte möglichst konservativ angesetzt werden.

Der Nutzenabstand Γ beschreibt die Schranke, die der Verlust einer Lösung nicht überschreiten darf. Im Algorithmus ExpandingCore aus dem Kapitel 2 werden in den Core Objekte eingefügt und aus allen vorhandenen Objekten eine Lösung erstellt. Diese Lösung kombiniert mit weiteren Objekten soll eine optimale Lösung für die vollständige Instanz ergeben. Der Algorithmus setzt darauf, dass nur die Objekte mit einem Verlust kleiner gleich der Schranke für den Nutzenabstand γ in den Core eingefügt werden. Die Begründung hierfür war, dass wenn der Verlust eines Objekts schon so groß ist, dieses nicht mehr relevant für die optimale Lösung ist.

Die Lösungen des Core sollen mit dem Nemhauser-Ullmann Algorithmus bestimmt werden. Die Verwendung ist gut umsetzbar, da immer sequenziell Objekte in den Core genommen werden und für diese Lösungen immer eine neue Paretomenge bestimmt werden kann. Der abschließende Schritt aus der Paretomenge die beste Lösung zu finden, kommt dagegen häufiger vor. Dieser kostet jedoch nur $\mathcal{O}(\log(|\mathcal{P}_i|))$ durch Verwendung einer Abwandlung von binärer Suche. Anders als bei der binären Suche wird nicht ein bestimmtes Objekt gesucht, sondern die Lösung mit dem größten Gewicht kleiner gleich der Gewichtsschranke. Diese Lösung hat auch den größten Nutzen aller Lösungen einer Paretomenge.

Wichtig im weiteren Filtern ist, dass das Lemma 2.1 weniger grob abgeschätzt wird. Statt nur nach den Verlusten eines Objekts zu schauen, werden in der Lösung die Summe der Verluste aller Objekte einer Lösung berücksichtigt. Alle Lösungen, die in dem Schritt aus der Paretomenge gefiltert werden, sind nach demselben Argument nicht optimal, nach welchem auch die Korrektheit von ExpandingCore bewiesen wurde.

Kapitel 5

Experimentelle Analyse

Eine experimentelle Analyse wurde bereits im Jahr 2006 von R. Beier und B. Vöcking in [BV06] durchgeführt.

Meine Experimente wurden auf einem Lenovo Thinkpad T430 Modell 2349-D17 ausgeführt und unter Ubuntu Linux 14.04 kompiliert. Das Thinkpad hat einen Intel Core i5-3320M mit 2,6 GHz, 8GB DDR3 Arbeitsspeicher. Es wurde die *GNU Compiler Collection* zur Erstellung der Binärdateien verwendet, welches in der Programmiersprache C nach dem ISO-Standard ISO/IEC 9899:1999 geschrieben wurde.

In den Experimenten wurden zufällige Instanzen gelöst. Die Nutzen und Gewichte sind uniform zufällige Zahlen aus dem Intervall $[0, 1]$, welcher auf 31 Bit hochskaliert und dann gerundet wurden, sodass jedes Objekt von zwei Variablen des Typs *Integer* repräsentiert werden. Die Kapazität wird durch $\beta \cdot \sum_{i=1}^{n} w_i$ bestimmt, wobei β einem Wert aus dem Intervall $[0, 1]$ entspricht.

5.1 Vergleich der Algorithmen zur Bestimmung des Teilungsobjekt

Die Abbildung 5.1 vergleicht die Laufzeiten der Algorithmen BreakitemA, BreakitemB und BreakitemC. In Kapitel 3 wurde bereits diskutiert, dass die asympotische Laufzeit von BreakitemA nur $\mathcal{O}(n \log(n))$ ist und die beiden anderen Algorithmen eine lineare Laufzeit haben. Es ist daher nicht verwunderlich zu sehen, dass dieser Algorithmus deutlich langsamer ist. Die offene Frage, welcher der beiden Linearzeitalgorithmen besser in der Praxis ist, sollte diese Grafik eindeutig beantwortet haben.

Die Graphen entstehen aus vielen Testpunkten, welche in gleichem Abstand von einander entfernt sind. An jedem Testpunkt werden 100 zufällige Instanzen erzeugt, die jeder der drei Algorithmen zu lösen hat. Dann wird die durchschnittliche Laufzeitzeit der 100 Versuche genommen und diese als Laufzeit eingetragen. Die Funktionsgraphen stellen die durschnittliche Laufzeit dar. Die einzelnen Punkte, die über den Graphen in der entsprechenden Farbe gekennzeichnet sind, stellen die oberen Außreißer im Bezug

Tabelle 5.1: Regressionsanalyse der BREAKITEM Algorithmen

Algorithmus	Laufzeit
BREAKITEMA	$2,01371 \cdot 10^{-8}\ n\ \log(n)$
BREAKITEMB	$2,33124 \cdot 10^{-8}\ n$
BREAKITEMC	$3,93765 \cdot 10^{-8}\ n$

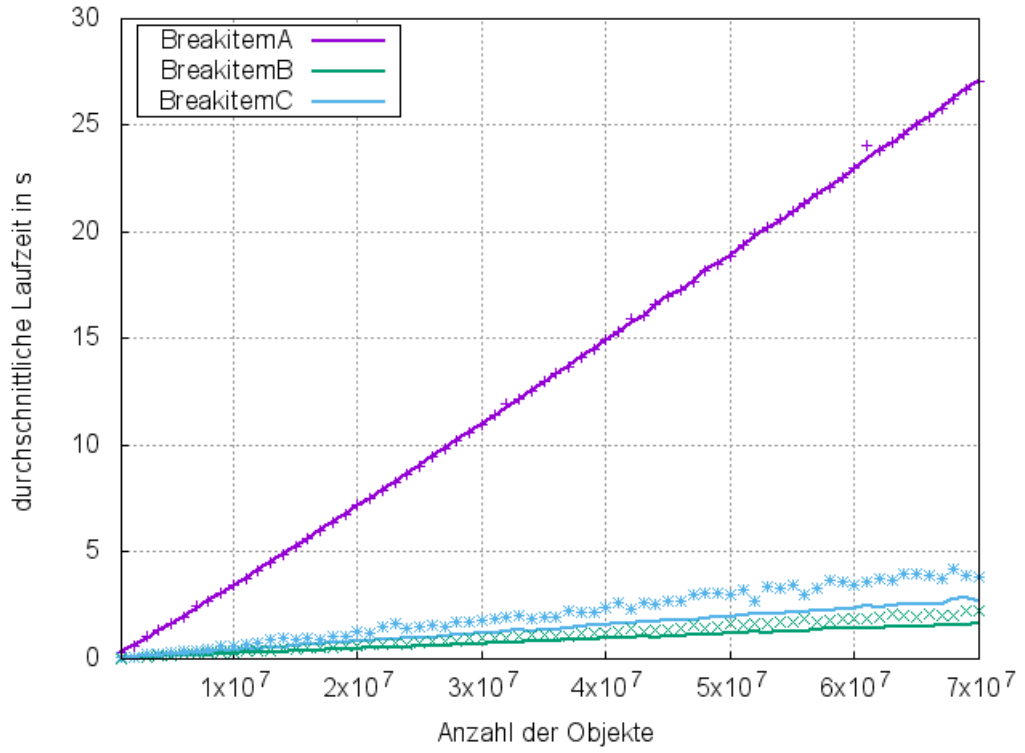

Abbildung 5.1: durchschnittliche Laufzeiten der BREAKITEM Algorithmen

auf die Laufzeit dar. Jeder Punkt stellt die maximale Laufzeit dar, welche in einer der Wiederholungen gemessen wurde.

Während der auf einem Sortieralgorithmus basierende BREAKITEMA nahezu gleich schnell auf allen zufälligen Distanzen läuft, ist das bei den anderen beiden Algorithmen nicht unbedingt der Fall. Der Grund hierfür ist, dass der Sortieralgorithmus zwingend alle Elemente einmal betrachten muss. Die Linearzeitalgorithmen dagegen brauchen nur einen statt zwei rekursive Aufrufe auszuführen. Je nachdem welches Teilungsobjekt gesucht wird, können so einige Objekte aus dem Betrachtungsraum verschwinden, sodass ein Algorithmus sehr viel schneller das richtige Objekt gefunden hat.

Der Partitionierungsalgorithmus DUALPIVOTPARTITION ist deutlich langsamer als PARTITION. Dargestellt wurde bereits, dass der Algorithmus QUICKSELECT mit 2 Pivotelementen spürbar langsamer läuft. Das bereits angesprochene Paper [WNM13] analysiert dies theoretisch und kommt zu dem Schluss, dass hierbei der Vorteil von kleineren Teilproblemen nicht den zusätzlichen Aufwand für das umständlichere Partitionieren ausgleicht. Der Gedanke bei BREAKITEMC basiert darauf, dass auch bei

diesem Problem ähnlich zu QUICKSORT Laufzeitverbesserungen zu erhoffen waren. Etwas anderes als das letztendlich experimentell zu untersuchen blieb nicht übrig.

5.1.1 Mögliche weitere Algorithmen

Die Überlegung muss aber nicht mit der Schlussfolgerung enden, dass alle Änderungen von BREAKITEMB zu schlechten Ergebnissen führen. Es gibt noch weitere Algorithmen, deren Ideen man auf das Problem zur Bestimmung des Teilungsobjekts übertragen kann. Die Änderungen, die im Algorithmus SELECT von Floyd und Rivest vorgenommen wurden, könnten weitere Verbessungerung an den bisherigen Algorithmen erzielen [Kiw05].

Bei SELECT geht es vorwiegend darum, die Struktur der Partition zu ändern, sodass häufig der Fall eintritt, dass alle Objekte zur fraktionalen Lösung gehören. Es ist klar, dass immer eine Wahl für den rekursiven Aufruf getroffen wird. Sollte die Entscheidung nicht auf den Fall treffen, dass alle bereits aufsummierten Gewichte kleiner gleich der Restkapazität sind, müssen Teile davon nochmals betrachtet und berechnet werden. An diesem Punkt können die Kosten nach dem Aufruf eines Partitionierungsalgorithmus, die in Abschnitt 3.3.3 bestimmt wurden, stark reduziert werden. Sollte dadurch vermieden werden, dass größere Gewichtssummen aufsummiert und verworfen werden, kann dadurch Rechenzeit eingespart werden. In dem Algorithmus von Floyd und Rivest werden die Parameter der Partition so angepasst, dass eine nicht symmetrische Partition wahrscheinlich ist. Diese Partition hat zum Ziel, dass es fünf Teilarrays gibt. Zwei Teilarrays enthalten nur jeweils gleiche Elemente und bieten diese als Lösung an. Die anderen drei Teilarrays sind ähnlich zu BREAKITEMC und werden rekursiv weiter verarbeitet. Dieser Algorithmus ist der schnellste bekannte für das Problem zum Finden des k.-kleinsten Elements eines Arrays. In der vorliegenden Arbeit wurden Ideen aus den aktuell besten Algorithmen für das Sortierproblems übernommen und auf das Finden des Teilungsobjekts übertragen. Da dies keine Verbesserungen bot, kann man bei anderen ähnlichen Problemen nach möglichen Ansätzen suchen und diese versuchen zu übernehmen.

Konkret würde das bedeuten, die Pivotelemente anders als in DUALPIVOTPARTITION zu wählen und durch eine geschicktere Aufteilung eine Verbesserung zu erreichen. Eine größere Liste möglicher Kandidaten für die Pivotelemente könnte dazu benutzt werden, die Teilarrays kleiner werden zu lassen. Je kleiner das Pivotelement gewählt ist, desto geringer ist das Teilarray der Elemente kleiner diesem Pivot.

5.2 Performance von ExpandingCore

Zu Anfang des Kapitels wurde bereits beschrieben wie die zufälligen Instanzen aussehen. Für ein festes n werden dann einige zufällige Instanzen erzeugt und diese jeweils gelöst. Um eine begründete Aussage treffen zu können, werden viele Wiederholungen mit denselben Parametern durchgeführt. Da bei dem Algorithmus EXPANDINGCORE

die Laufzeit sehr stark abhängig von den konkreten Nutzen- und Gewichtswerten abhängt, sind die abweichenden Resultate bei gleich bleibenden n unvermeidbar und treten sehr viel stärker auf als bei den Algorithmen zur Bestimmung des Teilungsobjekts. Daher habe ich die Anzahl der Wiederholungen von 100 auf 1000 erhöht, sodass der Grad der Abweichung besser eingeschränkt werden kann.

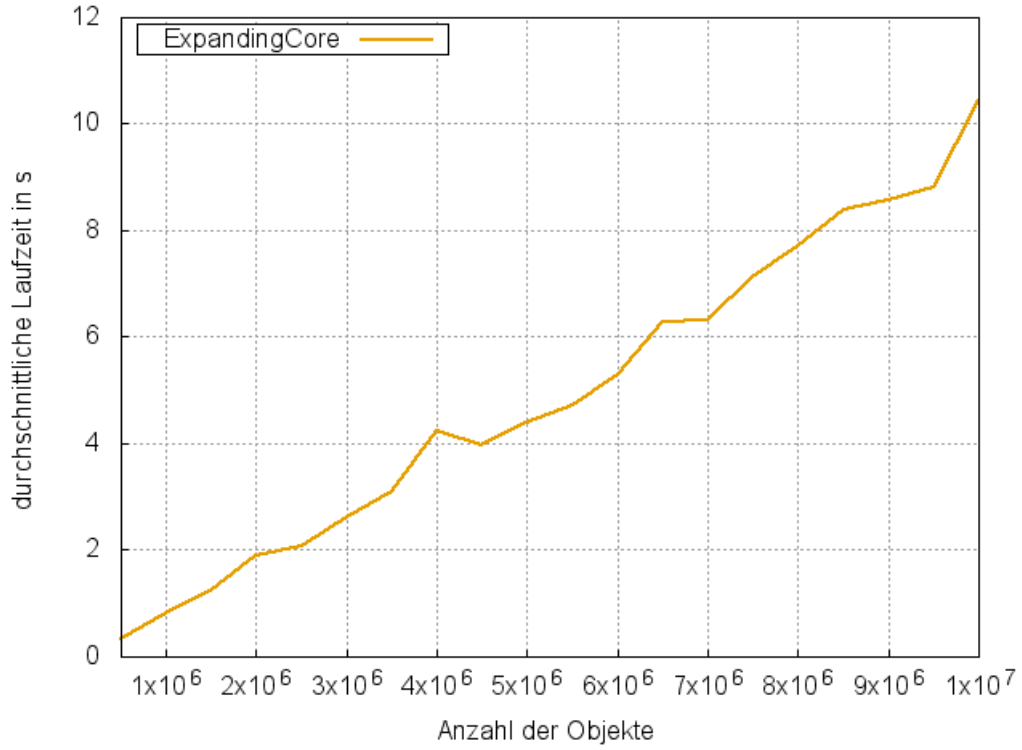

Abbildung 5.2: durchschnittliche Laufzeit von ExpandingCore

Um die Ergebnisse darzustellen, werden im Abschnitt 5.2 zwei unterschiedliche Typen von Diagrammen verwendet. Der erste Typ ist ein ganz normaler Graph und stellt wie in Abbildung 5.1 die durchschnittliche Laufzeit dar. Auch hier sind die darüber liegenden Punkte Maximalwerte einer Wiederholung. Der zweite Typ sind sogenannte Boxplots. Boxplots zeichnen sich dadurch aus, dass sie nicht nur einen einzigen Wert darstellen, sondern die Menge aller Messwerte mit ihrer Abweichung anzeigen. Die Linie, die durch alle Boxplots geht, verbindet den Median jeder Menge miteinander. Die angezeigten Boxen stellen die mittleren 50% der Messwerte dar. Die sogenannten Quartile charakterisieren die mittleren beiden Viertel der Messdaten. Die Striche an jeder Box heißen Wisker und sind maximal so wie lang wie das 1,5-Fache der Boxlänge, jedoch enden diese immer an einem Messwert.

Gemessen wird bei dem Algorithmus ExpandingCore nicht ausschließlich die Laufzeit. Es gibt einige andere Kenngrößen, die durchaus relevant sind und einige Aufschlüsse geben. Das ist die Größe des Core, also die Anzahl der Objekte in dem reduzierten Rucksackproblem, und die Anzahl der berechneten Paretopunkte.

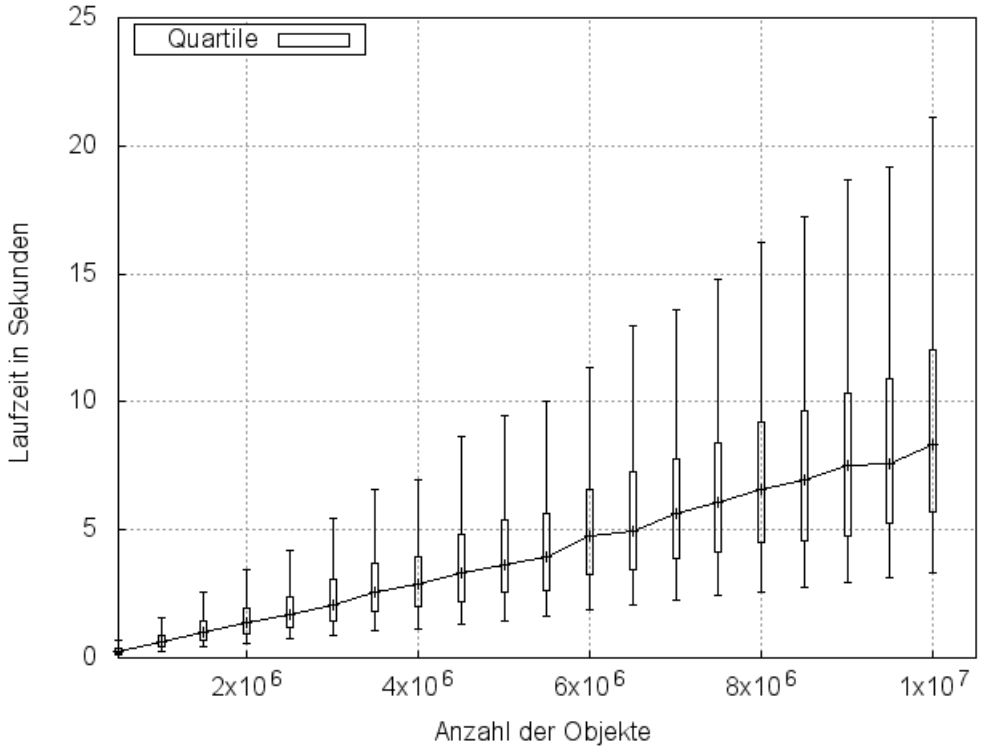

Abbildung 5.3: Laufzeit von ExpandingCore als Boxplot

In Abbildung 5.3 sieht man die Laufzeit als Boxplot und in Abbildung 5.2 die durchschnittliche Laufzeit. Eine Regressionsanalyse liefert für die Laufzeit

$$T(n) = 2,83013 \cdot 10^{-8}\, n\, log(n)^{1,27386}.$$

Der Graph, welcher die durchschnittliche Laufzeit darstellt, ist etwas anders gestaltet als der, welcher die Algorithmen zur Bestimmung des Teilungsobjekts darstellt. Der Grund hierfür liegt in dem Zoom auf den Graphen. In Abbildung 5.1 werden die maximalen Ausreißer als Punkte hinzugfügt. Das funktioniert nicht in diesem Fall, da die Ausreißer weit außerhalb des abgebildeten Raumes sind. Der Laufzeit Boxplot gibt jedoch an, wie die Abweichungen vom Mittel verlaufen. Die Punkte weit außerhalb des Abbildungsraums des Graphen liegen, sind daher auch nicht im Boxplot enthalten. Der Abstand der Quartile, also die Länge des Boxplots, gibt jedoch an mit wie viel Abweichung vom Standard auszugehen ist.

In Abbildung 5.4 sieht man das vermutlich interessanteste Ergebnis aus den Experimenten. Während die späteren Abbildungen einen ähnlichen Verlauf wie die Laufzeit haben, ist der Anstieg die Coregröße vergleichsweise niedrig. Ein besonders großer Anstieg ist nicht zu erkennen, der die Laufzeit begründen könnte. Die im Allgemeinen gute Laufzeit von ExpandingCore stammt daher, dass nicht mehr alle Objekte der Eingabe relevant sind und stattdessen eine reduzierte Probleminstanz gelöst wird. Die Laufzeit sollte also schnell sein, da die Instanzgröße gering wird. In diesem Fall scheint die Größe nicht einen so starken Einfluss zu haben wie angenommen. Die Abweichung hier scheint ähnlich stark zu sein wie das auch bei der Laufzeit beobachtet werden konnte.

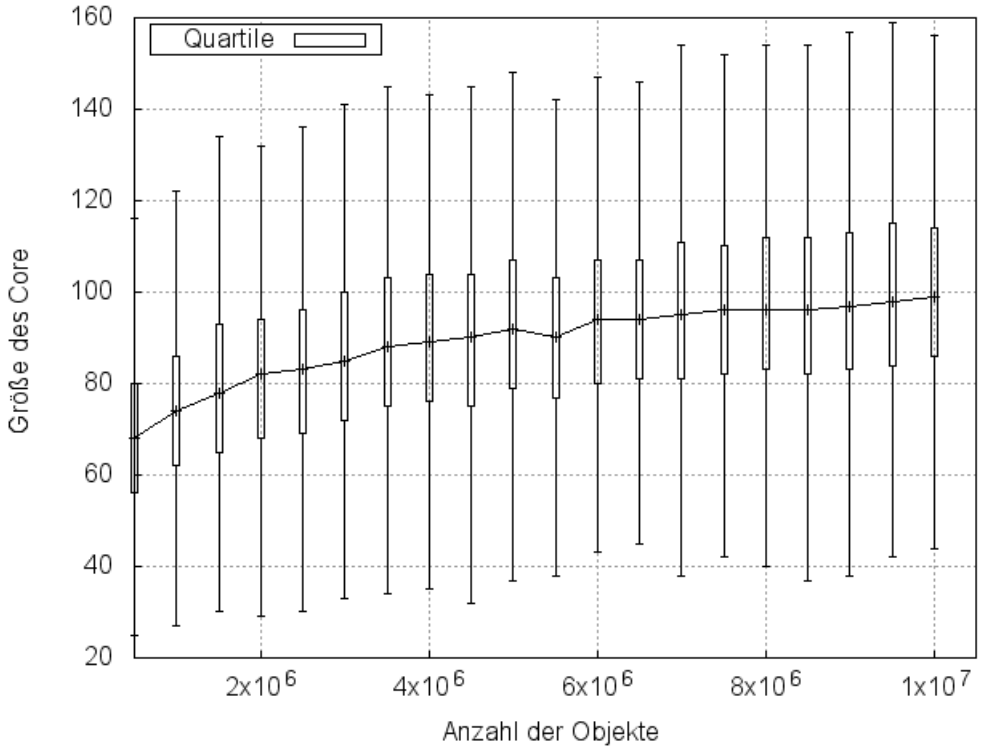

Abbildung 5.4: Größe des Core

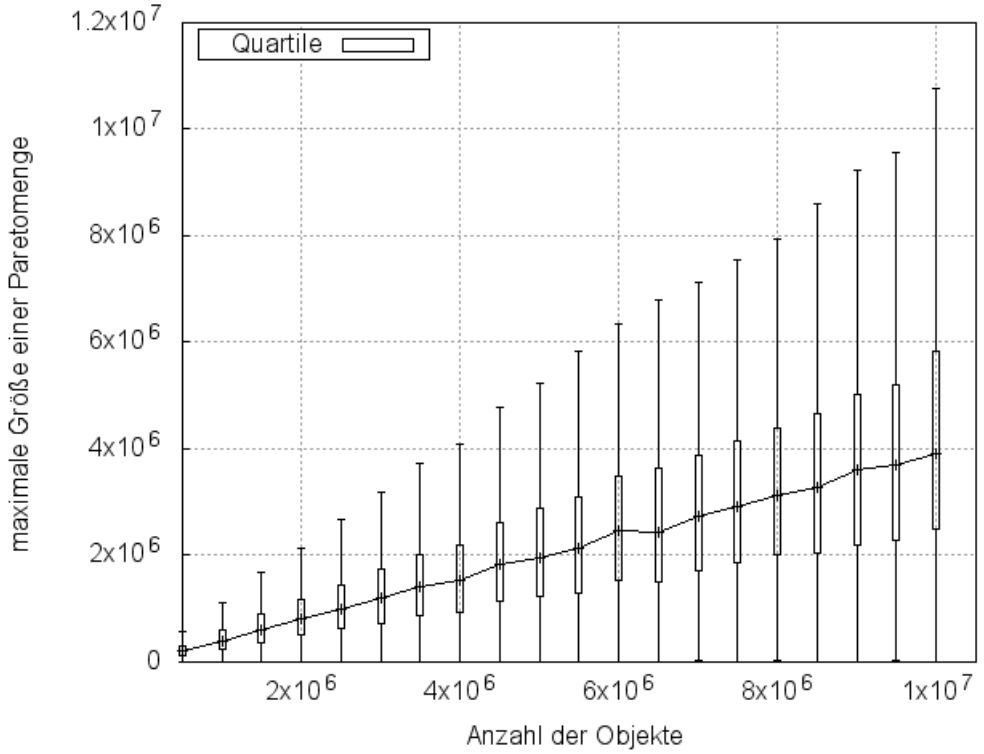

Abbildung 5.5: Maximum aller gespeicherten Paretopunkte

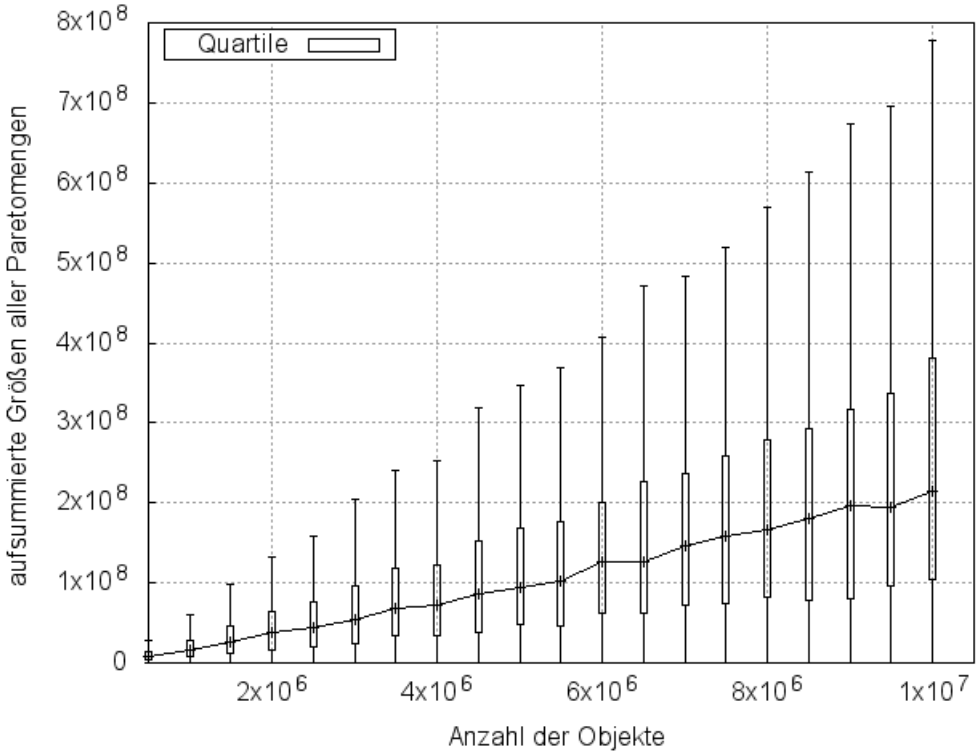

Abbildung 5.6: Summe aller gespeicherten Paretopunkte

Die Abbildungen 5.5 und 5.6 zeigen die Entwicklung der Paretopunkte. Die Paretopunkte stellen die Anzahl der berechneten Lösungen dar, welche nur Objekte des Core betrachten. Daher liegt die maximale Anzahl der Paretopunkte in $2^{|\text{Core}|}$ und ist direkt von der Coregröße abhängig. Die Graphen der Paretopunkte gleichen jedoch deutlich mehr denen der Laufzeit als der Coregröße.

Während der Aufführung des Algorithmus reicht es aus, zwei Paretomengen zu verwalten. Die Laufzeit des verwendeten Nemhauser-Ullmann Algorithmus liegt linear in der Anzahl der maximalen Paretopunkte, wodurch sich die Ähnlichkeit mit dem Laufzeit Boxplot erklären lässt. Wie in dem Kapitel bereits beschrieben, lässt sich der Lösungsvektor aus den Paretomengen und deren Paretopunkte wiederherstellen. Hierzu ist es dann notwendig, nicht nur zwei, sondern alle Mengen zu speichern. An diesem Punkt ist dann aus der Abbildung 5.6 abzulesen, wie viel Speicher insgesamt benötigt wird.

Der Unterschied zwischen den Paretopunkten und der Coregröße ist die Art, wie diese bestimmt werden. Die Coregröße hängt direkt mit den Verlusten der einzelnen Objekte und dem ganzzahligen Nutzenabstand Γ zusammen. Der Nutzen der optimalen Lösung und der fraktionalen Lösung sinkt stark mit der Anzahl der Objekte in der Lösung, da ein fehlendes Objekt bei 100 Objekten sich insgesamt sich stärker bemerkbar macht als bei einer Millionen Objekte. Ein kleiner werdendes Γ verkleinert den Core, da mehr Objekte ausgeschlossen werden. Im Gegenzug dazu sammeln sich deutlich mehr Objekte in der Nähe des Danzig-Strahls, sodass mehr Objekte zum Core gehören. Die Paretopunkte leiten sich jedoch aus dem Dominanzkriterium ab, welches keinen direkten Zusammenhang zu der Definition des Objektverlusts hat. Ein größer werdendes

n lässt die Paretopunkte auch dann ansteigen, wenn die Coregröße nur geringfügig geändert wird. Die Anzahl der Paretopunkte scheinen ein deutlich genaueres Bild der Laufzeit zu bieten als die Coregröße.

5.3 Fazit

Insgesamt spricht einiges für die gute Verwendbarkeit der Core-Algorithmen in der Praxis, die ein NP-vollständiges Problem in durchaus vertretbarer Zeit lösen können. Die Speicherkomplexität verhält sich ähnlich wie die Laufzeitkomplexität. Für noch viel größere Instanzen, in denen nicht alle Paretomengen gleichzeitig in den Arbeitsspeicher passen würden, lässt sich eine Auslagerung auf die Festplatte und spätere Rekonstruktion ohne größere Einschränkungen realisieren. Daher sollten sich alle für die Praxis relevanten Probleminstanzen unter Betrachtung der Laufzeit- und Speicherkomplexität effizient lösen.

Die vorgestellten Algorithmen zur Bestimmung des Teilungsobjekts sind deutlich effizienter als die naive Methode, die auf einen Sortieralgorithmus zurückgreift. Im Abschnitt 5.1.1 wurde noch eine Möglichkeit angesprochen, mit der man einen neuen Algorithmus für dieses Problem implementieren könnte. Dieser sollte inbesondere bei genauerer Kenntnis über die Struktur der Eingabeinstanz besser optimiert werden können, sodass weitere Verbesserungen der Laufzeit erzielt werden können. Klar beobachtbar ist jedoch, dass das vorherige Berechnen des Teilungsobjekts keinen besonders großen Einfluss auf die Laufzeit hat und vermutlich keine signifikante Steigerung der Gesamtlaufzeit durch Optimierungen erwartbar ist.

Literaturverzeichnis

[BV04] René Beier and Berthold Vöcking. Probabilistic analysis of knapsack core algorithms. In J. Ian Munro, editor, *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 468–477. SIAM, 2004.

[BV06] René Beier and Berthold Vöcking. An experimental study of random knapsack problems. *Algorithmica*, 45(1):121–136, 2006.

[Kiw05] Krzysztof C. Kiwiel. On floyd and rivest's SELECT algorithm. *Theor. Comput. Sci.*, 347(1-2):214–238, 2005.

[Lyo] Bradford F. Lyon. Visualization of dual pivot partition by Yaroslavskiy. `https://googledrive.com/host/0B2GQktu-wcTiNEtsejVjRWlmaWs/`.

[NW14] Markus E. Nebel and Sebastian Wild. Pivot sampling in java 7's dual-pivot quicksort. *CoRR*, abs/1403.6602, 2014.

[Rö14] Heiko Röglin. Algorithmen und Berechnungskomplexität 1. `http://www.roeglin.org/teaching/WS2013/AlgoI/AlgoI.pdf`, Vorlesungsskript, Universität Bonn, Wintersemester 2013/14.

[Rö15] Heiko Röglin. Probabilistc analysis of algorithms. `http://www.i1.informatik.uni-bonn.de/lib/exe/fetch.php?media=lehre:ss15:probanalysis:probanalysis.pdf`, Vorlesungsskript, Universität Bonn, Sommersemester 2015.

[Wil15] Sebastian Wild. Java 7's dual pivot quicksort. Master's thesis, 2015. Buchvordruckt `http://wwwagak.cs.uni-kl.de/images/sebastian/wild-master-thesis-book-preprint.pdf`.

[WNM13] Sebastian Wild, Markus E. Nebel, and Hosam Mahmoud. Analysis of quickselect under yaroslavskiy's dual-pivoting algorithm. *CoRR*, abs/1306.3819, 2013.

[Yar09] Vladimir Yaroslavskiy. Dual-pivot quicksort. `http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf`, 2009.