Mark - Compact Garbage Collection

Seminar Garbage Collection WS 2001/2002
Universität des Saarlandes

Jens Regenberg *jens@ps.uni-sb.de*

22. April 2002

Zusammenfassung

Trotz des rasanten Wachstums von Speichermodulen ist der verfügbare Speicher selbst in den modernsten Computern nicht unbegrenzt. Wie mit allen endlichen Resourcen muss man diese sehr sparsam benutzen und wiederverwenden. Garbage Collection bezeichnet die automatische Freigabe von nicht mehr benötigtem Speicher. Verschiedene Algorithmen, wie zum Beispiel die Mark-Sweep Algorithmen, hinterlassen einen fragmentierten Speicher. So ist es sehr aufwendig, neue Knoten zu allozieren bzw. die beste Position für einen neuen Knoten zu finden. Kompaktiert man den Speicher nach einer Garbage Collection, so ist das Einfügen neuer Knoten extrem einfach.

1 Annahmen und Definitionen

Im Folgenden wird die Größe des Speichers mit M bezeichnet. Zeiger zwischen Knoten heißen interne Zeiger. Weiterhin wird angenommen, dass die lebendigen Knoten schon markiert sind. Dies kann zum Beispiel durch einen Mark-Sweep Algorithmus geschehen sein. Lebendige Knoten sind Knoten, die von dem Root-Set durch interne Zeiger erreichbar sind. Eine Lücke bezeichnet freien Speicher zwischen zwei lebendigen Knoten.

2 Einleitung

Die Kompaktierung des Speichers ist nicht ohne zusätzliche Kosten möglich. Es gibt verschiedene Algorithmen, die anhand folgender Kriterien charakterisiert werden. Der Speicherbedarf, der zusätzlich benötigt wird, um Informationen über die neuen Positionen verschobener Knoten zu speichern, spielt beispielsweise eine Rolle. Weiterhin unterscheiden sich die Algorithmen in der Anzahl der Speicherdurchläufe, die zum Kompaktieren des Speichers notwendig sind. Im Allgemeinen kommen die Algorithmen mit zwei bis drei Speicherdurchläufen aus. Ein weiteres Unterscheidungsmerkmal ist die Fähigkeit des Algorithmus mit verschiedenen Knotengrößen zurechtzukommen.

Bei der Kompaktierung von Speicher unterscheidet man grundsätzlich zwei Arten.

- Bei der *zufälligen Kompaktierung* werden die einzelnen Speicherblöcke ohne Rücksicht auf ihre ursprüngliche Anordnung verschoben. Die relative Anordnung der Knoten wird dabei nicht beibehalten.
- Die *gleitende Kompaktierung* erhält die relative Anordnung der Knoten. Hier werden die noch lebendigen Knoten zusammen an den Anfang des Speichers geschoben. Dabei werden die freien Blöcke an das Ende des Speichers geschrieben.

Ein Kompaktierungsalgorithmus hat normalerweise drei Phasen. In der ersten Phase werden die noch lebendigen Knoten markiert. Die zweite dient zur tatsächlichen Kompaktierung des Speichers. In der dritten Phase werden dann die Zeiger innerhalb der einzelnen Knoten aktualisiert.

In den nächsten Abschnitten werden verschiedene Kompaktierungsalgorithmen erläutert. Zunächst wird ein einfacher Algorithmus dargestellt, der dann in den darauf folgenden Beispielen in einzelnen Aspekten verändert wird.
Abschnitt 3 beschreibt den Two-Finger Algorithmus von Edward [Sau74] als Beispiel für einen einfachen und schnellen Algorithmus ($O(M)$) mit *zufälliger Kompaktierung*.
Die darauf folgenden Algorithmen sind Beispiele für *gleitende Kompaktierung*. Der Lisp2 Algorithmus [CN83, Mor78] hat ebenfalls eine asymptotische Laufzeit von $O(M)$, benötigt jedoch einen zusätzlichen Zeiger in jedem Knoten.
Der Haddon-Waite Algorithmus [Had67] im 5. Abschnitt benötigt keinen zusätzlichen Speicher, allerdings hat er eine asymptotische Laufzeit von $O(M \log M)$.
Jonkers Algortihmus [Jon79] im 6. Abschnitt ist ein Beispiel für einen Algorithmus, der lineare Laufzeit benötigt und keinen zusätzlichen Speicher verbraucht.

3 Two-Finger Algorithmus

In diesem Abschnitt wird vorrausgesetzt, dass die Knoten im Speicher eine feste Größe haben. Außerdem muss der Speicher ein zusammenhängender Block sein.

Der Two-Finger Algorithmus benötigt zwei Durchläufe durch den Speicher, um ihn zu kompaktieren.

- Im ersten Durchlauf werden die lebendigen Knoten an den Anfang des Speichers verschoben. Der zweite Durchlauf untersucht die Knoten in dem schon kompaktierten Speicher und aktualisiert die Zeiger. Der Algorithmus benötigt dazu zwei Zeiger

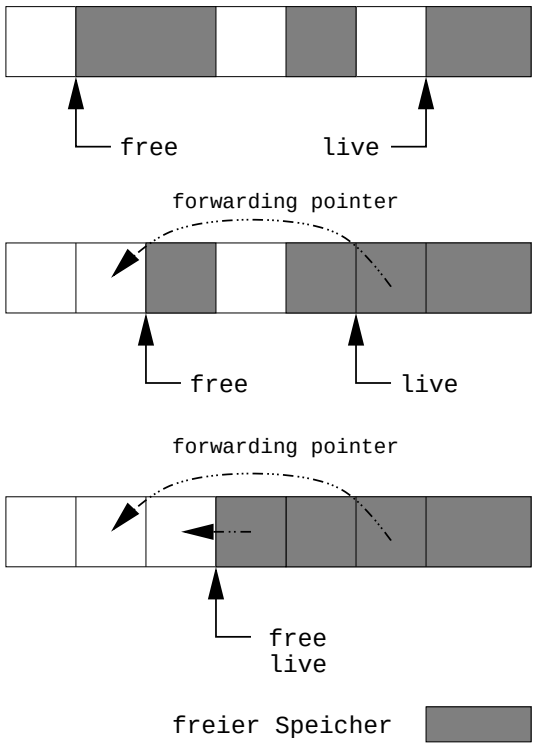

Abbildung 1: Der Two-Finger Algorithmus

(siehe Abbildung 1). Der Zeiger `free` zeigt immer auf die erste freie Stelle und durchläuft den Speicher von Anfang nach Ende. Der zweite Zeiger `live` untersucht den Speicher von oben nach unten. Sobald `live` auf einen als lebendig markierten Knoten trifft, wird dieser an die Stelle verschoben die von `free` markiert wird. Jetzt wird `free` an die nächste freie Position im Speicher gesetzt. Bevor `live` nun weiterläuft, wird an die Stelle des gerade verschobenen Knotens die neue Adresse des Knotens geschrieben. Danach beginnt die Prozedur von vorne. Der Algorithmus terminiert sobald sich `live` und `free` treffen.

Der Speicher ist nun partitioniert. Die lebendigen Knoten befinden sich im unteren Teil des Speichers, die beiden Zeiger verweisen auf die erste freie Stelle, und der Teil des Speichers oberhalb der beiden Zeiger ist frei.

- Der zweite Durchlauf dient dazu, die internen Zeiger zu aktualisieren. Der Algorithmus durchläuft den Speicher nochmals von Anfang an, bis er die Adresse der Zeiger `free` und `live` erreicht. Trifft er dabei auf einen Knoten, der einen internen Zeiger auf eine größere Adresse hat als die von `free` bzw. `live`, ist bekannt, dass das Ziel des Zeigers verschoben wurde. Der interne Zeiger wird nun mit der neuen Adresse des verschobenen Knotens aktualisiert. Diese wurde im ersten Durchlauf an seiner alten Adresse hinterlassen. Enthält ein Knoten keinen internen Zeiger oder solch einen internen Zeiger, dessen Adresse kleiner ist als die von `free` und `live`, ist keine Aktualisierung notwendig.

3.1 Analyse

Der Two-Finger Algorithmus besticht durch seine Einfachheit. Er hat lineare Laufzeit in der Größe des Speichers und benötigt keinen zusätzlichen Speicher. Zudem benötigt er zum Kompaktieren des Speichers lediglich zwei Durchläufe.

Im Gegensatz zu den in den folgenden Abschnitten vorgestellten Algorithmen ist der Two-Finger Algorithmus nicht fähig Knoten verschiedener Größe zu bearbeiten. Er funktioniert nur dann, wenn sichergestellt ist, dass an der Stelle, die von `free` bezeichnet wird, ausreichend Platz für jeden Knoten ist. Der Two-Finger Algorithmus zerstört außerdem die relative Ordnung der Knoten.

3.2 Variationen

Man kann den Two-Finger Algorithmus jedoch so modifizieren, dass er mit verschiedenen Knotengrößen zurecht kommt. Der Speicher wird dazu in Regionen für verschiedene Knotengrößen unterteilt. Der Algorithmus wird dann auf den einzelnen Regionen des Speichers seperat ausgeführt.

4 Lisp 2 Algorithmus

Der Lisp 2 Algorithmus benötigt zum Kompaktieren des Speichers einen zusätzlichen Zeiger in jedem Knoten. Dieser Zeiger kann schon zum Markieren der lebendigen Knoten benutzt werden, zum Beispiel durch einen von *nil* verschiedenen Wert. Im Gegensatz zum Two-Finger Algorithmus benötigt der Lisp 2 Algorithmus drei Läufe durch den Speicher.

- Der erste Durchlauf wird dazu benutzt, die neuen Adressen der Knoten zu berechnen. Dazu durchläuft ein Zeiger den Speicher von Anfang bis Ende und schreibt in das zusätztliche Feld des Knotens die neue Adresse. Diese Adresse berechnet sich durch die Länge der lebendigen Knoten, die bisher überschritten wurden (siehe Abb. 2).

- Im zweiten Durchlauf werden die internen Zeiger mit den neu berechneten Adressen aktualisiert. Dazu läuft der Algorithmus wieder von Anfang bis Ende durch den

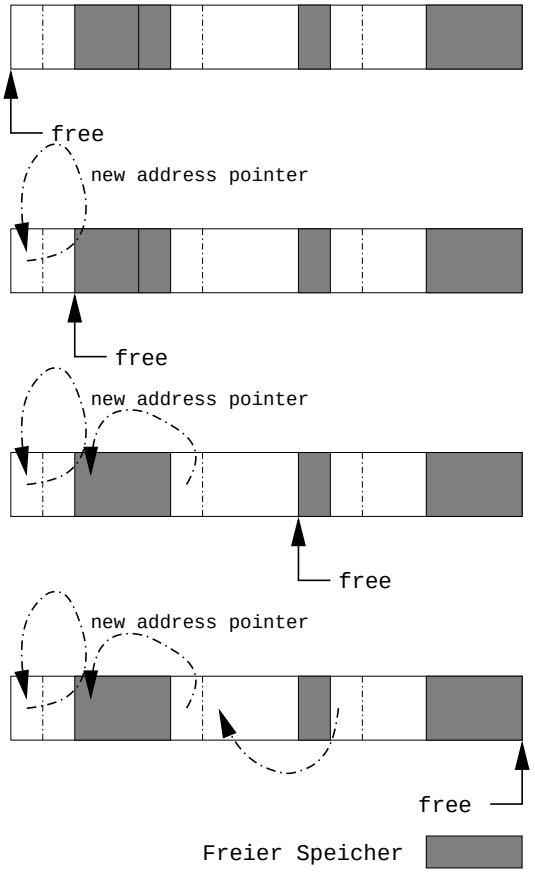

Abbildung 2: Die Berechnung der neuen Adressen im Lisp 2 Algorithmus

Speicher. Jeder interne Zeiger wird mit der neuen Adresse seines Zieles aktualisiert, die in dem zusätzlichen Feld gespeichert ist.

- Der dritte und letzte Durchlauf dient dazu, die Knoten tatsächlich zu verschieben. Diese werden jetzt an die im ersten Durchlauf berechneten Adressen verschoben. Abschliessend wird die Markierung aufgehoben.

4.1 Analyse

Der Lisp 2 Algorithmus erhält im Gegensatz zu dem Two-Finger Algorithmus die relative Anordnung der einzelnen Knoten. Asymptotisch hat er ebenfalls lineare Laufzeit. Allerdings kostet die Erhaltung der relativen Anordnung einen zusätzlichen Speicherdurchlauf und den Speicher für das zeigergroße Feld, in dem die neue Adresse des Knotens gespeichert wird.

5 Haddon-Waite Algorithmus

Der Haddon-Waite Algorithmus [Had67] gehört zu den Tabellen basierten Algorithmen. Er erzeugt eine sogenannte “Break Table”, in der die Informationen über die neuen Positionen der einzelnen Knoten gespeichert werden. Die einzige Vorraussetzung für den Speicher ist, dass der kleinste Knoten mindestens zwei Adressen enthalten kann. Im Gegensatz zum Lisp 2 Algorithmus benötigt der Haddon-Waite Algorithmus keinen zusätzlichen Speicher, sondern kommt mit den vorhanden Lücken im Speicher aus.

Der Algorithmus benötigt zwei Speicherdurchläufe. Im ersten Durchlauf werden die Knoten an den Anfang des Speichers verschoben und es wird die Tabelle erzeugt, die es im zweiten Durchlauf ermöglicht, die internen Zeiger zu aktualisieren.

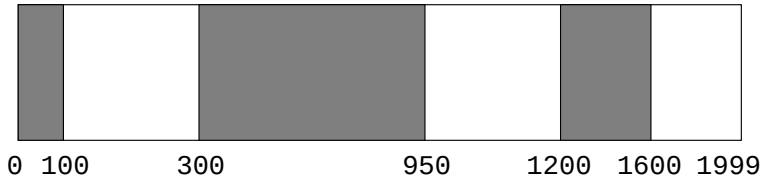

Abbildung 3: Der Speicher vor der Kompaktierung

- Wenn der Algorithmus im ersten Durchlauf einen Block aus lebendig markierten Knoten erreicht, verschiebt er diesen an die erste freie Stelle im Speicher. Die erste freie Stelle berechnet sich wie beim Lisp 2 Algorithmus aus den Größen der bisher überschrittenen Knoten. Als nächstes erzeugt der Algorithmus einen neuen Eintrag in der Tabelle.

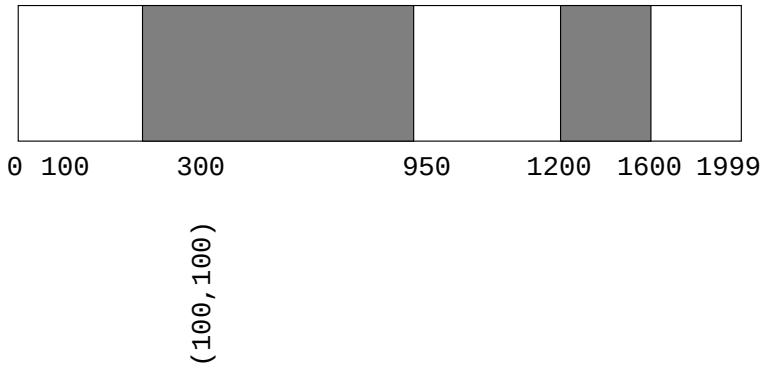

Abbildung 4: Der erste Block wurde an den Anfang des Speichers verschoben und ein Tabelleneintrag erzeugt.

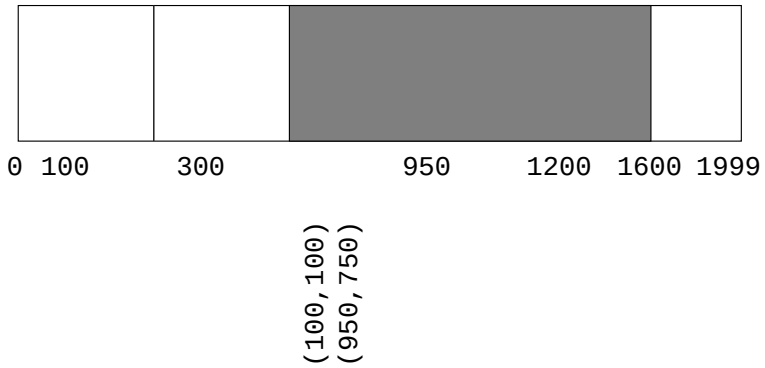

Abbildung 5: Der nächste Block ist an den Anfang des Speichers verschoben worden und ein neuer Eintrag wurde der Tabelle hinzugefügt.

Im Folgenden wird durch Induktion über die Anzahl *n* der lebendigen Blöcke gezeigt, dass immernoch genügend Speicher für einen neuen Tabelleneintrag vorhanden ist.

– $n = 1$

  Gibt es nur einen Block aus zusammenhängenden lebendigen Knoten im Speicher, so sind zwei Fälle zu unterscheiden. Der erste Fall ist, dass der Block bereits am Anfang des Speichers liegt. In diesem Fall muss kein Tabelleneintrag erzeugt werden, da der Speicher bereits kompaktiert ist. Der zweite Fall tritt dann ein, wenn am Anfang des Speichers wenigstens eine freie Speicherzelle ist. In diesem Fall wird der Block an den Anfang des Speichers verschoben. Hinter dem gerade verschobenen Block ist jetzt mindestens eine freie Speicherzelle vorhanden. Aufgrund der Vorraussetzung, dass die kleinste Zelle im Speicher wenigstens zwei Worte gross ist, ist hinter dem Block genügend Platz für einen Tabelleneintrag.

– $n \rightsquigarrow n+1$

  Nach Induktionsvorraussetzung können die ersten n Blöcke verschoben und die dazugehörigen Tabelleneinträge erzeugt werden. Nun muss zwischen dem $n-$ten und dem $(n+1)-$ten Block wenigstens eine freie Zelle liegen, in der noch kein Tabelleneintrag ist. Analog zum Induktionsanfang wird nun der $(n+1)-$te Block an das Ende des $n-$ten Blocks verschoben. Dabei muss die Tabelle schrittweise hinter den $(n+1)-$ten Block kopiert werden. Nachdem der $(n+1)-$te Block verschoben ist, ist mindestens eine freie Zelle Speicher am Ende der Tabelle noch frei, da keine neuen Einträge erzeugt, sondern nur bereits vorhandene Einträge verschoben wurden. Durch dieses "Durchrollen" der Tabelle geht evtl. die Ordnung verloren.

Der Eintrag in der Tabelle besteht aus zwei Zahlen. Die erste Zahl ist die alte Startadresse des verschobenen Blocks, die zweite Zahl ist die Differenz der Adressen, um die der Block nach vorne verschoben wurde (siehe Abbildung 4). Das Verschieben der lebendigen Blöcke und das Erzeugen der Tabelleneinträge werden jetzt iterativ so lange wiederholt, bis alle als lebendig markierten Knoten am Anfang des Speichers stehen (siehe Abbildung 6). Jetzt muss die Tabelle aufsteigend nach den urspünglichen Addressen sortiert werden.

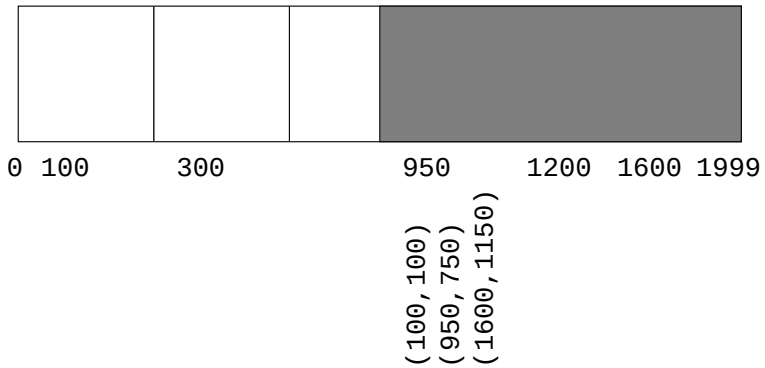

Abbildung 6: Der Speicher ist partitioniert und die Tabelle wieder sortiert.

- Der zweite Durchlauf aktualisiert nun die internen Zeiger. Wenn der Algorithmus einen internen Zeiger erreicht, sucht er per binärer Suche in der Tabelle zwei Einträge (a,s) und (a',s'), so dass die Adresse des Ziels zwischen a und a' liegt. Die neue Zieladresse p des Zeigers berechnet sich dann einfach als $p = p - s$. Bildlich gesprochen wird der Zeiger einfach um die gleiche Differenz verschoben wie der Block im ersten Durchlauf.

5.1 Analyse

Der Haddon-Waite Algorithmus kommt ohne zusätzlichen Speicher aus und erhält die relative Anordnung der Knoten. Allerdings tauscht man im Vergleich zu dem Lisp 2 Algorithmus Speicher gegen zusätzliche Laufzeit. Zwar braucht der Algorithmus nur zwei Durchläufe durch den Speicher, jedoch muss unter Umständen die Tabelle nach dem ersten Durchlauf sortiert werden, um die internen Zeiger aktualisieren zu können. Man erhält so eine asymptotische Laufzeit von $O(M \log M)$. Wegbreit schlägt vor[Weg72], statt der Tabelle eine Hash Tabelle zu benutzen, um diesen Nachteil zu kompensieren.

6 Jonkers Algorithmus

Jonkers Algorithmus stellt drei Bedingungen. Interne Zeiger dürfen nur auf den Header eines Knoten zeigen. Diese Header müssen groß genug sein, um einen Zeiger enthalten zu können. Ausserdem dürfen Header nur Werte enthalten, die von Zeigern verschieden sind.

6.1 Threading

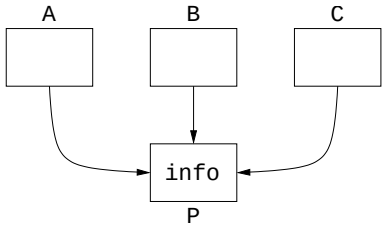

Abbildung 7: Zustand vor dem Threading. Die Information steht noch in dem Knoten *P*

Um bei dem Aktualisieren der internen Zeiger nicht jeden lebendigen Knoten nach Zeigern durchsuchen zu müssen, werden die Knoten so umgeordnet, dass alle Zeiger, die auf einen Knoten *P* zeigen, auch von *P* gefunden werden können. Diese Technik wurde zuerst von Fischer [Fis74] vorgestellt und wird “Threading” oder Auffädeln genannt.

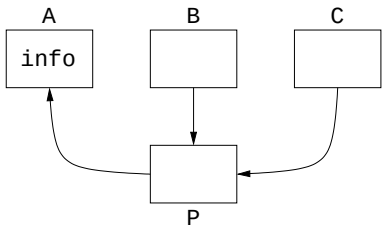

Abbildung 8: *A* und *P* sind aufgefädelt

Die Knoten werden jetzt der Reihe nach untersucht. Zuerst wird der Knoten *A* betrachtet. *A* und *P* werden aufgefädelt. Dazu werden die Inhalte von *P* und *A* vertauscht. *A* enthält also nach dem ersten Schritt die Information von *P*, und *P* enthält jetzt einen Zeiger auf die Zelle von *A*, die vorher den Zeiger auf *P* enthielt (siehe Abbildung 8). Da Zeiger nach Voraussetzung nur auf den Header eines Knoten zeigen dürfen, kann man somit aufgefädelte Knoten von nicht aufgefädelten Knoten unterscheiden.

Das gleiche passiert nun auch mit *B*. Nach dem Threaden enthält dann *B* einen Zeiger auf *A* und *P* einen Zeiger auf *B*. Das gleiche gilt dann analog auch für *C*. Das Ergebnis ist in Abbildung 9 zu sehen.

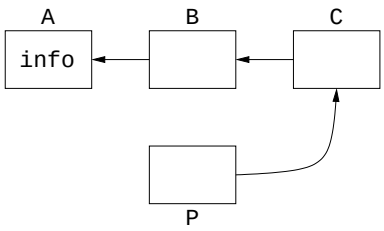

Abbildung 9: Alle Knoten die einen Zeiger auf *P* enthielten sind aufgefädelt

Im folgenden Abschnitt werden Zeiger, die durch Auffädeln entstehen, immer in dem Header des aufgefädelten Knotens gespeichert. Sie zeigen auf die Zelle, in der vorher der urspüngliche Zeiger stand.

6.2 Kompaktierung

Im Folgenden wird der Algorithmus von Jonkers [Jon79] betrachtet. Dieser Algorithmus benötigt zwei Durchläufe durch den Speicher. Der erste Durchlauf bearbeitet die internen Zeiger, die vorwärts auf andere Knoten zeigen, während der zweite Durchlauf sich um die rückwärts zeigenden Knoten kümmert und die Knoten dann verschiebt.

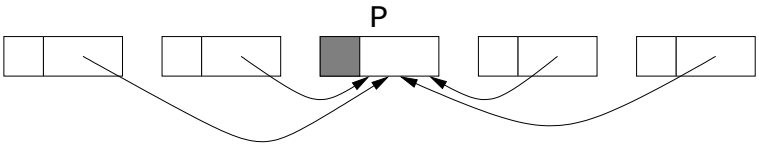

Abbildung 10: Die initiale Konfiguration enthält alle Knoten die auf *P* verweisen

Zur Verdeutlichung wird der Algorithmus am Beispiel eines Knotens analysiert. Er funktioniert analog dazu für mehrere Knoten. Dazu sei eine Konfiguration wie in Abbildung 10 gegeben. Nun werden solange alle Knoten gethreaded, bis der Knoten *P* erreicht wird. Das ist der erste Knoten, der einen nach hinten gerichteten internen Zeiger hat (siehe Abbildung 11).

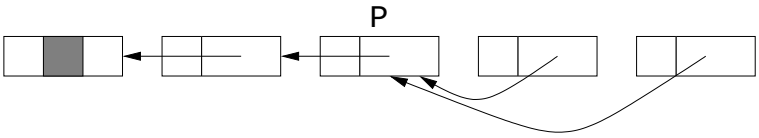

Abbildung 11: Alle vorwärtsweisenden Zeiger auf *P* sind gethreaded

Da *P* erreicht ist, können die vorwärtsweisenden Zeiger mit der neuen Adresse von *P* aktualisiert werden. Diese berechnet sich wieder als die Summe der Größen der bisher überschrittenen lebendigen Knoten (siehe Abbildung 12).

Der erste Durchgang endet damit, dass alle rückwärtsweisenden Zeiger auf *P* gethreaded werden. Alle Selbst-Referenzen werden als rückwärtsweisende Zeiger interpretiert. Am Ende des ersten Durchlaufs sind alle vorwärtsweisenden Zeiger aktualisiert und alle rückwärtsweisenden Zeiger auf *P* gethreaded (siehe Abbildung 13).

Im zweiten Speicherdurchlauf werden die im ersten Durchlauf schon aktualisierten Knoten an den Anfang des Speichers verschoben. Dies ist bis einschliesslich *P* möglich. Außerdem berechnet der Algorithmus die neue Position von *P* nocheinmal. Nach dem Verschieben von *P* aktualisiert der Algorithmus die rückwärtsweisenden Zeiger auf *P* (siehe Abbildung 14). Abschliessend werden jetzt auch noch die restlichen Knoten verschoben.

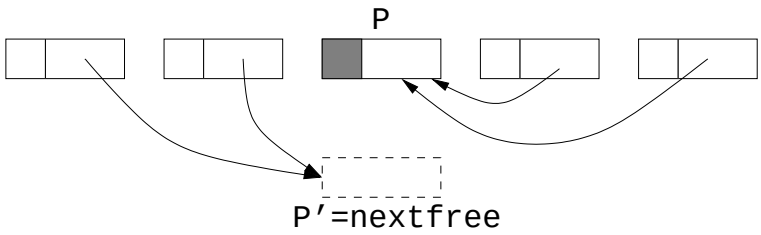

Abbildung 12: Alle vorwärtsweisenden Zeiger sind mit der neuen Adresse von *P* aktualisiert worden

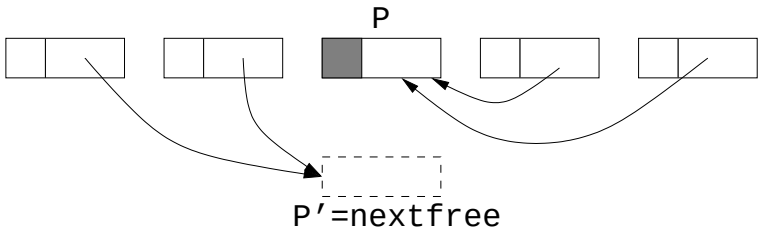

Abbildung 13: Alle rückwärts auf *P* weisenden Zeiger werden gethreaded

6.3 Analyse

Die Vorteile des Algorithmus von Jonkers sind offensichtlich. Er benötigt lediglich zwei Durchläufe durch den Speicher um diesen zu Kompaktieren. Dabei hat er auch nur lineare Laufzeit und benötigt keinen zusätzlichen Speicher. Sein Nachteil liegt in der Komplexität. So wird im Vergleich zu den in den Abschnitten 3 - 5 vorgestellten Algorithmen pro Schritt mehr Aufwand betrieben. Jeder lebendige Knoten wird dreimal vom Algorithmus untersucht. Zuerst im ersten Durchlauf, dann im zweiten Durchlauf und schliesslich beim Threading.

7 Zusammenfassung

Die Kompaktierung des Speichers ist ohne Frage kostenintensiv. Jedoch werden die Allozierungskosten für neue Knoten dramatisch gesenkt. Statt wie bei Mark-Sweep eine Lücke suchen zu müssen, in die der Knoten hineinpasst, kann man, ähnlich einem Copying-Collector, einfach ab dem `free`-Zeiger neuen Speicher allozieren. Lange lebendige Knoten werden nicht wie bei einem Copying-Collector immer wieder kopiert, sondern einmal an den Anfang des Speichers verschoben und dann nicht weiter berücksichtigt. Ein weiterer Vorteil gegenüber den Copying-Collectoren ist der um die Hälfte kleinere Adressraum, den die Mark-Compact Algorithmen benötigen. Auch kann die relative Anordnung der Knoten im Speicher erhalten werden. Um den Aufwand möglichst gering zu halten, ist es üblich, den Speicher nicht bei jeder Garbage-Collection zu Kompatkieren. Es gibt Heuristiken, die vorschlagen, wann zusätzlich zu einer Garbage Collection der Speicher Kompaktiert werden sollte.

Algorithmus	Art	Knotengröße	Durchläufe	zus. Speicher	Laufzeit
Two-Finger	zufällig	fest	2	keiner	$O(M)$
Lisp 2	gleitend	beliebig	3	1 Zeiger je Knoten	$O(M)$
Haddon-Waite	gleitend	beliebig	2	keiner	$O(M \log M)$
Jonker	gleitend	beliebig	2	zeigergroße Header	$O(M)$

Tabelle 7 - Eigenschaften der in den Abschnitten 3 - 6 vorgestellten Algorithmen

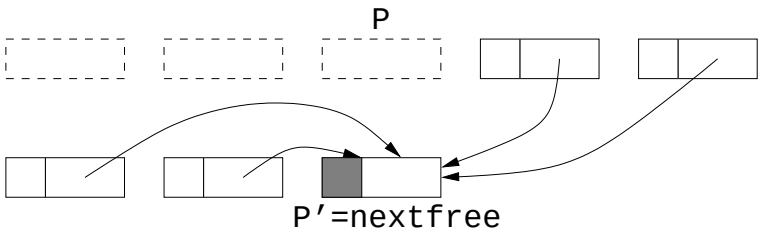

Abbildung 14: Die rückwärtsweisenden Zeiger sind aktualisiert und *P* wurde verschoben

Die Entscheidung für die einzelnen Algorithmen hängt sehr von den Datenstrukturen ab, die damit kompaktiert werden sollen. Two-Finger Algorithmen können nur Knoten einer festen Größe bzw. einer Menge fester Größen bearbeiten. Jonkers Algorithmus verlangt, dass man die Headerwerte von Zeigern zweifelsfrei unterscheiden kann. Der Platz- und Zeitverbrauch der Algorithmen spielt bei dieser Entscheidung sicherlich auch eine Rolle. Tabelle 7 gibt einen Überblick über die besprochenen Algorithmen sowie deren Kompaktierungseigenschaften und Vorraussetzungen.

Literatur

[CN83] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *Programming Languages and Systems*, 5(4):532–553, 1983.

[Fis74] David A. Fisher. Bounded workspace garbage collection in an address order preserving list processing environment. *Information Processing Letters*, 3(1):25–32, July 1974.

[Had67] B. Haddon. Waite: A compaction procedure for variable length storage elements, 1967.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection - Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[Jon79] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):25–30, July 1979.

[Mor78] F. Morris. A time- and space-efficient garbage compaction algorithm, 1978.

[Sau74] R. Saunders. The lisp system for the q–32 computer, 1974.

[Weg72] B. Wegbreit. A generalised compactifying garbage collector. *Computer Journal*, 15(3):204–208, August 1972.