Grundlegende Verfahren der Speicherbereinigung - Kompaktierer nach Haddon & Waite und Edwards

Christoph Müllner[1] and Harald Steinlechner[2]

[1] e0525067@student.tuwien.ac.at
[2] e0825851@student.tuwien.ac.at

Abstract. Automatische Speicherbereingung spielt eine zentrale Rolle für die effiziente Implementierung von modernen Programmiersprachen. Die Speicherkompaktierung ist ein wichtiges Thema der Speicherbereinigung. Sie wirkt der Fragmentierung entgegen und kann dadurch einen positiven Einfluss auf das Laufzeitverhalten haben. In diesem Artikel behandeln wir zwei grundlegenden Verfahren zur Kompaktierung des Heaps - Kompaktierer nach Haddon & Waite, sowie den Edwards Kompaktierer.

1 Einführung

Speicherverwaltung spielt eine essentielle Rolle in Programmiersprachen. Automatische Speicherbereinigung ist aus vielen vor allem moderneren Sprachen nicht wegzudenken und wirkt sich maßgeblich auf die Ausführungsgeschwindigkeit von Programmen aus. Dies hat die Entwicklung vieler verschiedener Algorithmen und Ansätze motiviert. Um moderne Garbage Collection Systeme verstehen zu können ist es notwendig, auch ältere, elementare Verfahren zu beherrschen.

Grundsätzlich ist Speicherbereinigung ein Vorgang, der den Speicherverbrauch eines Computerprogramms verringern soll. Konkret werden nicht mehr benötigte, jedoch allozierte Speicherbereiche freigegeben.

Die Speicherbereinigung kann in zwei Phasen aufgeteilt werden [1]:

1. Referenzierten Speicher erkennen
2. Nicht referenzierten Speicher freigeben

Verfahren bei denen lebende Objekte nach der zweiten Phase kontinuierlich im Speicher liegen, nennt man kompaktierende Verfahren oder Kompaktieralgorithmen. Durch diese soll erreicht werden, dass die Fragmentierung des belegten Speichers reduziert wird. Diese Optimierung der örtlichen Lokalität kann zu einem effizienteren Laufzeitverhalten führen, da die verschiedenen Ebenen der Speicherhierarchie besser ausgenutzt werden können.

Grundsätzlich wird bei Kompaktieralgorithmen so verfahren, dass in der zweiten Phase der Speicherbereinigung markierte Blöcke in einem bestimmten Bereich kompaktiert werden.

Formal kann man folgende Schritte unterscheiden:

- Erstelle einen Plan für das neue Speicherlayout. In Abbildung 1 wird ein derartiger Plan beispielhaft veranschaulicht.
- Modifiziere Zeiger auf Zellen entsprechend dieses Plans
- Verschiebe alle Speicherbereich, wie im Plan berechnet wurde

Ein wichtiges Qualitätsmerkmal von Kompaktieralgorithmen ist die Anordnung der Objekte bezüglich ihrer Ursprungsposition (vor der Speicherbereinigung). Folgende drei Anordnungen werden unterschieden:

- Beliebig (keinerlei Ordnung garantiert)
- Gleitend (Anordnung der Objekte im Speicher bleibt gleich)
- Linearisierend (Referenzierte Objekte werden näher plaziert)

In dieser Arbeit werden zwei wichtige grundlegende kompaktierende Verfahren behandelt:

- Kompaktierer nach Haddon & Waite [2]
- Edvarks Kompaktierer [3]

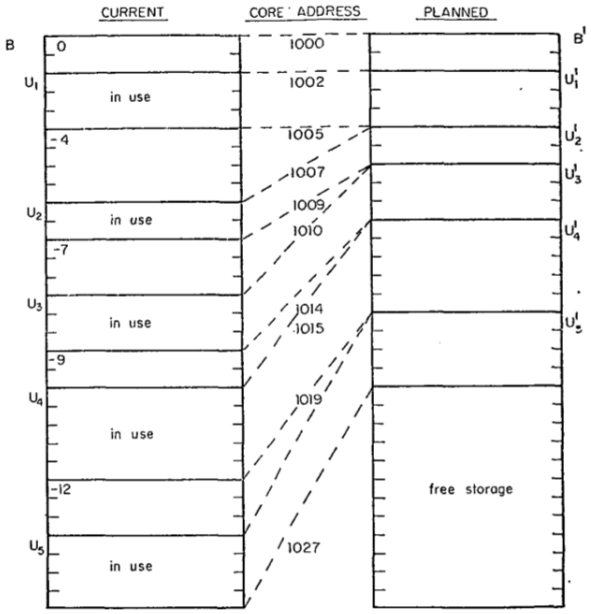
2 Table Compactors, Haddon & Waite, Wegbreit

Wie im vorigen Kapitel erwähnt, ist Speicherfragmentierung ein ernst zu nehmendes Problem bei nicht kompaktierender Speicherbereinigung. In Extremfällen führt Fragmentierung dazu, dass das Programm gar nicht fortgesetzt werden kann obwohl in Summe noch ausreichend Speicherplatz verfügbar wäre. In [4] wird ein Speicherbereinigungsalgorithmus vorgestellt, welcher alle belegten Zellen als kontinuierlichen Block an den Beginn des Speichers verschiebt, wodurch ein darauf folgender ebenfalls kontinuierlicher freier Speicherbereich entsteht, in welchem neue Zellen alloziert werden können. Durch diese kompaktierende Vorgehensweise kann das Programm in weiterer Folge fortgesetzt werden.

In Haddon et al. [4, 6] wird die Markierungsphase und Erstellung des Speicherplans kombiniert. Der Algorithmus verschiebt alle Zellen an die niedrigstmögliche Position im Speicher. Exemplarisch wird dieser konzeptuelle Plan in Abbildung 1 veranschaulicht. Das heißt die kompaktierende Phase wird nur aufgerufen falls benötigt.

Der Algorithmus beginnt also sofort mit dem verschieben der Zellen und hinterlässt in frei-werdenden Zellen Information über den aktualisierten Aufenthaltsort der Zelle - dazu wird das Paar $(urspr\!nglicheAdresse, Offset)$ dort abgelegt. Dies zeigt eine Limitierung des Algorithmus: der kleinste freiwerdende Bock muss mindestens diese Relokationsinformation abspeichern können. Kommt man in die Situation, dass eine Zelle auf einen bereits verschobenem Bock (hier steht also die Relokationsinformation) verschoben werden sollte, wird dieser Eintrag im freien Speicher hinten angereiht. Die Tabelle wird wie die Autoren Haddon & Waite [4] beschreiben also durch den freiwerdenden Speicher *gerollt*. Nach

Fig. 1. Plan für die Kompaktierung. Links wird die aktuelle Speicherbelegung dargestellt, rechts das Speicher-Layout nach Kompaktierung (entnommen aus [5])

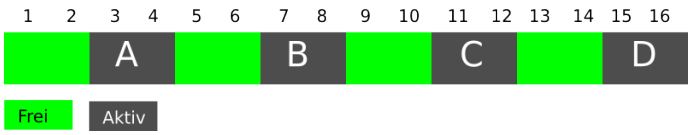


Abschluss dieser Phase müssen die Adressen entsprechend der Tabelle angepasst werden. Obwohl der Speicher sequentiell durchlaufen wurde, ist die Tabelle bedingt durch das *Rollen* unsortiert und muss für die effiziente Anpassung der Adressen erneut sortiert werden. Dies ergibt den dominierenden Aufwand von $O(n * logn)$ für den Algorithmus.

In Abbildung 2 wird diese Vorgehensweise veranschaulicht.

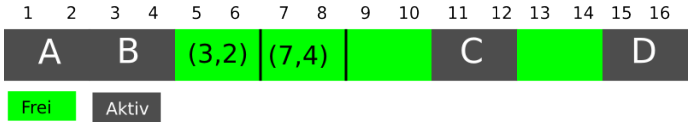
2.1 Analyse

Der Algorithmus von Haddon et al. [4] benötigt für die Speicherbereinigung keinen zusätzlichen Speicherplatz. Dies ist wie in Schorr et al. [6] beschrieben dadurch möglich, dass bei der Markierungsphase in Listen direkt Rückwärtszeiger gespeichert werden wodurch die Zeigerstruktur ohne zusätzlichen Speicher (wie etwa Stack) durchmustert werden kann. Auf diese Weise wird etwas mehr Zeit beim markieren benötigt - die Laufzeit beträgt allerdings trotzdem $n*logn$, was sich durch den Sortierungsaufwand der Verschiebungstabelle ergibt. Lang et al. [5] markiert etwas schneller, was allerdings auf Kosten des Speicherplatzes geht - hier wird zusätzlicher Speicherplatz benötigt. Wegbreit [2] generalisiert den Algorithmus von Haddon et [4] auf allgemeine Datenstrukturen wie etwa zusammengesetzten Datentypen. Fitch et al. [7] verbessert Haddon & Waite [4] ebenfalls mit zusätzlichem Heap - Speicherplatz, ist allerdings

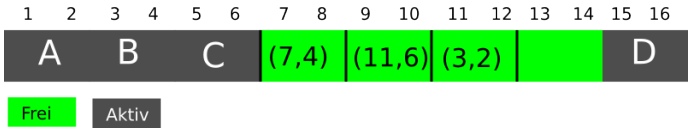


Zu Beginn ist in diesem Beispiel der Speicher vollkommen fragmentiert.

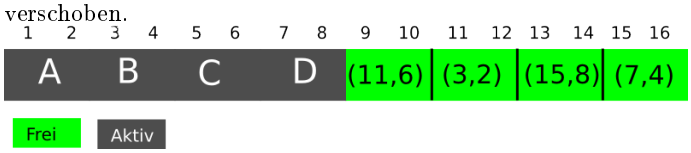
Die erste Belegte Zelle wird an $(1, 2)$ verschoben und Relokationsinformation in der nächsten freien Zelle hinterlassen.



Als nächstes wird B auf $(3, 4)$ verschoben und Relokationsinformation in der nächsten freien Zelle hinterlassen.



Die nächste belegte Zelle ist C, in der ersten eigentlich freien Zelle befindet sich allerdings bereit ein Relokationseintrag. Dieser wird an das Ende des freien Speicherbereichs verschoben.



Nach erneuter Verschiebung von Zelle und Relokationsinformation erhält man einen kontinuierlichen Speicherbereich.

Fig. 2. Exemplarische Veranschaulichung des Kompaktierungsvorgang

im Regelfall schneller, da die Sortierung ausgespart wird. Im schlechtesten Fall ist der dort vorgestellte Algorithmus allerdings gleich schnell wie Haddon & Waite [4].

3 Edwards Kompaktierer

Der Edwards Kompaktierer ist ein weiterer Kompaktierer, welcher in einer Übungsaufgabe in [3] beschrieben ist.

Der Edwards Kompaktierer setzt die erste Phase der Speicherbereinigung voraus, d.h. alle validen Speicherzellen wurden markiert (und der Markierungszustand einer Zelle ist in dieser gespeichert).

3.1 Grundlagen

Der Algorithmus arbeitet mit Knoten (nodes). Der Aufbau eines Knotens ist in Abbildung 3 dargestellt. Ein Knoten enthält die folgenden Felder:

- **MARK**
  MARK ist ein Bit, welches den Markierungszustand des Knotens angibt. Markierte Knoten (MARK = 1) werden nicht freigegeben.
- **ATOM**
  ATOM ist ebenfalls ein Bit, welches angibt, ob es sich bei dem Knoten um ein Atom handelt (ATOM = 1) oder nicht.
- **ALINK**
  ALINK ist ein Zeiger auf einen weiteren Knoten. Der Wert α repräsentiert dabei einen ungültigen Zeiger. Dieses Feld wird vom Algorithmus ignoriert, wenn es sich bei dem Knoten um ein Atom handelt.
- **BLINK**
  BLINK ist wie ALINK ein weiterer Zeiger auf einen Knoten. BLINK besitzt die gleichen Eigenschaften wie ALINK.

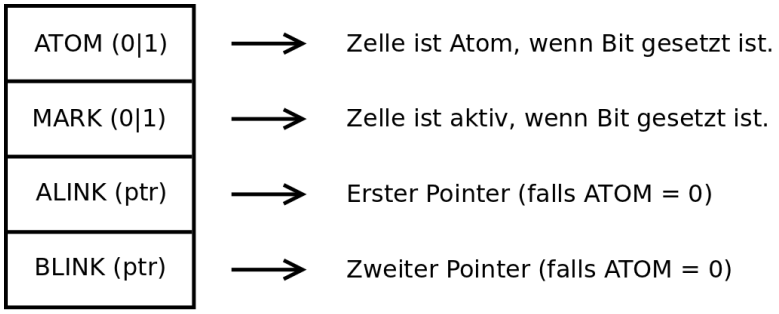


Fig. 3. Speicherzelle für Edwards Kompaktierer

Die Zellen besitzen die Größe eines Maschinenwortes.

Der zu bearbeitende Speicher besteht aus M Knoten, welche mit $NODE(i)$ (mit $i < -1...M$ erreicht werden können. Analog zu $NODE(i)$ gibt es noch die Möglichkeit auf die einzelnen Elemente der Knoten zuzugreifen ($MARK(i)$, $ATOM(i)$, $ALINK(i)$ und $BLINK(i)$).

3.2 Phasen des Edwards Kompaktierers

Der Edwards Kompaktierer besitzt zwei Phasen:

1. Phase 1: Kompaktieren der Knoten
   In dieser Phase werden die (markierten) Knoten kompaktiert, indem jene Knoten mit hohen Adressen in leere Bereiche (unmarkierte Knoten) mit niedrigen Adressen kopiert werden. Im alten, bereits kopierten Knoten, wird im ALINK Feld die neue Adresse des Knotens abgelegt.
2. Phase 2: Korrigieren der Knoten
   Nun wird über alle Knoten iteriert und die ALINK und BLINK Felder aller Knoten, welche keine Atome sind, bei Bedarf angepasst (d.h. die Zeiger werden auf die neue Adresse gesetzt, falls der referenzierte Knoten eine neue Position besitzt).

Der Edwards Kompaktierer ist in Listing 1.1 in Python dargestellt. Der Algorithmus setzt zwei zusätzliche Knoten voraus, welche mit $NODE(0)$ bzw. $NODE(M+1)$ erreichbar sind. Diese zusätzlichen Knoten dienen als Terminierungshilfe für den Algorithmus. In Phase 1 werden zunächst zwei Zeiger, L und K, initialisiert, welche auf den ersten (L) bzw. den letzten Knoten (K) zeigen. Dann wird nach dem niedrigsten unmarkierten Bereich (Knoten) und dem höchsten markierten Knoten gesucht und der markierte Knoten in den unmarkierten Bereich kopiert. Dabei wird im ALINK Feld der alten Kopie die neue Adresse des Knotens hinterlassen. Nach dem Kopieren, werden die nächsten Knoten gesucht. Treffen die beiden Zeiger von Phase 1 zusammen, ist Phase 1 abgeschlossen.

In Phase 2 des Edwards Kompaktierers wird über die kompaktierten Knoten 1...K iteriert. Die Markierung der Knoten wird gelöscht und die Adressen der ALINK und BLINK Felder wird bei Knoten, welche keine Atome sind, korrigiert, sofern sie auf verschobene Knoten zeigen.

Listing 1.1. Edwards Kompaktierer

```
def edwards():
        # Initialisierung
        L = 0
        K = M + 1
        MARK(0) = 1 # Terminierungshilfe
        MARK(M+1) = 0 # Terminierungshilfe

        # Phase 1: Kompaktieren der Knoten
        while True:

                # Suche nach niedrigstem
                # unmarkierten Knoten
                L = L + 1
                while MARK(L) == 1:
                        L = L + 1

                # Suche nach hoechstem
                # markierten Knoten
                K = K - 1
                while MARK(K) == 0:
                        K = K - 1
```

```
            # Solange L und K nicht
            # zusammentreffen wird
            # verschoben
            if L > K:
                    break

            # Kopiere Knoten K nach L
            NODE(L) = NODE(K)
            # Setze Zeiger ALINK auf neue
            # Position im alten Knoten
            ALINK(K) = L
            # Loesche die Markierung
            # des alten Knoten
            MARK(K) = 0

    # Phase 2: Korrigieren der Knoten
    # Iteriere ueber die, nun kompaktierten,
    # Knoten (L = 1 ... K)
    for L in range(1, K+1):
            # Loesche die Markierung des
            # alten Knoten
            MARK(L) = 0
            # Korrigiere ALINK bzw. BLINK,
            # wenn sie auf einen verschobenen
            # Knoten zeigen
            if ATOM(L) == 0 and ALINK(L) > K:
                    ALINK(L) = ALINK(ALINK(L))
            if ATOM(L) == 0 and BLINK(L) > K:
                    BLINK(L) = ALINK(BLINK(L))
```

3.3 Analyse

Die Laufzeit des Algorithmus ist linear zur Speichergröße und benötigt keinen zusätzlichen Speicherplatz. Der Algorithmus berücksichtigt nicht die Anordnung der Objekte im Speicher und ist deshalb den Kompaktierern mit beliebiger Anordnung der Objekte im Speicher zuzuordnen. Der Algorithmus setzt Knoten mit der Größe eines Maschinenworts ("one-word nodes" [3]) voraus. Eine Verallgemeinerung auf Knoten mit beliebiger, aber gleicher Größe ist jedoch möglich.

3.4 Optimierung

Steel [8] beschreibt ein parallelisiertes Speicherbereinigungsverfahren, bei dem in der Relokationsphase eine Kompaktierung nach Edwards durchgeführt wird.

4 Zusammenfassung

In diesem Artikel haben wir wichtige kompaktierende Verfahren kennen gelernt, die ohne zusätzlichen Speicherplatz während der Kompaktierungsphase auskom-

men. Haddon & Waite hat logarithmische Laufzeit bezüglich der Speichergröße und erhält die relative Ordnung der Objekte. Edwards kompaktierer arbeitet linear zur Speichergröße, erhält aber die relative Ordnung der Zellen nicht.

References

1. Wilson, P.R.: Uniprocessor garbage collection techniques. In: Proceedings of the International Workshop on Memory Management. IWMM '92, London, UK, UK, Springer-Verlag (1992) 1–42
2. Wegbreit, B.: A generalised compactifying garbage collector. The Computer Journal **15**(3) (March 1972) 204–208
3. Knuth, D.E.: The art of computer programming, volume 1 (3rd ed.): Fundamental Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
4. Haddon, B.K., Waite, W.M.: A compaction procedure for variable-length storage elements. The Computer Journal **10**(2) (August 1967) 162–165
5. B.Lang, B.: Fast compactification (1972)
6. Schorr, H., Waite, W.M.: An efficient machine-independent procedure for garbage collection in various list structures. Commun. ACM **10**(8) (August 1967) 501–506
7. Fitch, J.P. & Norman, A.: A note on compactifying garbage collection. The Computer Journal **21**(1) (January 1978) 31–34
8. Steele, Jr., G.L.: Multiprocessing compactifying garbage collection. Commun. ACM **18**(9) (September 1975) 495–508