Container Loading

Maik Riechert (10INM)

18. Februar 2011

1 Problembeschreibung

Dieser Artikel bezieht sich auf die Projektarbeit im Fach *Evolutionäre Algorithmen* an der HTWK Leipzig. Ein schwer zu lösendes kombinatorisches Problem soll mithilfe von evolutionären Algorithmen approximiert werden.

Der Autor untersuchte das sogenannte Behälterproblem, auch bekannt als dreidimensionales *Bin Packing* oder *Container Loading*. Es ist NP-schwer und bei großen Probleminstanzen nur approximiert lösbar.

Beim allgemeinen klassischen Bin Packing werden eine Anzahl von Objekten in mehrere Behälter verteilt, sodass dieser nicht „überläuft“ und die benötigte Behälteranzahl minimal wird. Dies kann sich auf den verfügbaren Raum, aber auch Objektgewichte oder Ähnliches beziehen. Allen Problemklassen gemeinsam ist, dass für jeden Behälter eine oder mehrere Bedingungen erfüllt werden müssen, z.B. das Maximalgewicht oder das verfügbare Volumen.

In dieser Arbeit wird der Fall betrachtet, dass nur ein einziger Container C mit der Größe (B_C, H_C, T_C) existiert, in den n quaderförmige Kisten K_i $(0 \leq i < n)$ teilweise verschiedener Größe (B_i, H_i, T_i) und erlaubter Drehung D_i möglichst platzsparend beladen werden sollen. Die erlaubte Drehung legt fest, ob die Breite und Tiefe in die Vertikale gestellt werden kann. Da die Höhe sich schon in der Vertikalen befindet, muss dies nicht festgelegt werden. Ebenso ist eine Drehung mit der Höhe als Achse immer erlaubt – jeweils um 90°. Für jede Kiste ist also D_i wie folgt festgelegt:

$$D_i = (B_i^{(D)}, T_i^{(D)}) \tag{1}$$

$$D_i.* = \begin{cases} 1, & \text{Kante als Vertikale erlaubt} \\ 0, & \text{sonst} \end{cases} \tag{2}$$

Die Schwierigkeit ergibt sich vor allem daraus, dass meist viel mehr Kisten vorhanden sind, als theoretisch in den Container passen würden. Es muss also eine Kistenmenge gefunden werden, bei deren Beladung L die Platznutzung N im Container maximal bzw. die Platzverschwendung V minimal wird. L ist eine Menge von Tupeln, wobei jedes Tupel

eine zu beladende Kiste mit dem Kistenindex idx_i, der Position p_i im Container, sowie der Rotation r_i beschreibt.

$$L = \{(idx_1, p_1, r_1), (idx_2, p_2, r_2), \ldots\} \text{ mit } p_i = (x_i, y_i, z_i), r_i = (b_i, t_i, v_i) \tag{3}$$

$$N = \sum_{i \in L.idx} B_i \cdot H_i \cdot T_i \tag{4}$$

$$V = B_C \cdot H_C \cdot T_C - N \tag{5}$$

$i \in L.idx$ beschreibt hier die Menge aller idx-Elemente aus jedem Tupel in L, d.h. $i \in L.idx = \{idx_1, idx_2, \ldots\}$.

2 Modellierung

Das vorliegende Problem soll durch einen genetischen Algorithmus möglichst optimal approximiert oder, falls alle gegebenen Kisten theoretisch in den Container passen, im Idealfall gelöst werden. Da ein solcher Algorithmus bei praktischer Anwendung sehr oft ausgeführt und zeitnah gute Ergebnisse liefern muss, steht vorallem die benötigte Zeit für die Erreichung solcher Ergebnisse im Vordergrund.

Im Folgenden wird das Format des Genotyps, eine Methode für dessen Dekodierung, sowie die Fitnessfunktion beschrieben.

2.1 Datenrepräsentation

Die triviale unveränderte Repräsentation des Phänotyps L als Genotyp, d.h. als Menge von Tupeln mit Positions- und Rotationsangaben der Kisten, ist als Ergebnis für den Benutzer zwar wünschenswert, aber völlig ungeeignet für die Verarbeitung im evolutionären Algorithmus. Jede Positions- oder Rotationsänderung einer Kiste würde in den meisten Fällen zu einem ungültigen Individuum führen, da sich Kisten überschneiden würden.

Der gängige und auch hier gewählte Ansatz ist es, die Beladungsreihenfolge als Genotyp zu benutzen. Die Dekodierungsfunktion stellt damit eine Heuristik dar, welche deterministisch eine Beladung mit gegebener Reihenfolge simuliert und damit die Positionsangaben erzeugt, was wiederum den Phänotyp ergibt.

Die Rotationseigenschaft kann nun entweder Teil des Genotyps sein, oder genau wie die Position in der Dekodierungsfunktion gefunden bzw. festgelegt werden. Hier ist die Rotation Teil des Genotyps $I.G$, welcher eine Permutation darstellt, in dessen Tupeln die Rotationskomponente geändert werden kann.

$$I.G = ((idx_1, r_1), (idx_2, r_2), \ldots, (idx_n, r_n)) \tag{6}$$

Es ist zu beachten, dass der Genotyp *jede* Kiste enthält. Die Dekodierungsfunktion muss also entscheiden, welche der Kisten nicht beladen werden, falls kein Platz mehr

im Container vorhanden ist. Dieses Vorgehen ist nötig, da die Information, welche Kisten weggelassen werden müssen, hier noch nicht existiert und weil sich eine ändernde Permutationsgröße schlecht im (evolutionären) Algorithmus nutzen lassen würde.

Anmerkung: In der Implementierung wird aus Effizienzgründen statt dem Index ein Tupel bestehend aus Index, Kistengröße und erlaubten Drehungen verwendet, um das ständige Abfragen dieser Daten aus einer zentralen Datenstruktur zu vermeiden.

2.2 Dekodierung

In der Literatur werden viele Beladungsheuristiken beschrieben, welche aber vor allem bei rein deterministischen Algorithmen eingesetzt werden [6][7][1][3]. Die Beladungsreihenfolge wird dabei meist erst in der Heuristik gewählt und ist nicht vorher festgelegt. Aus diesem Grund und weil der Autor ein eigenes Verfahren entwickeln wollte, werden die Heuristiken in der Literatur nicht weiter betrachtet. Da bei evolutionären Algorithmen eine hohe Generationszahl häufig bessere Ergebnisse liefert, stand hier eine eher simple Heuristik mit schneller Ausführungszeit im Vordergrund.

Die hier benutzte Heuristik stellt sicher, dass es keine Freiräume unter den Kisten gibt und Kisten immer von oben beladen werden. Der Ablauf ist wie folgt:

```
dec(G: Genotyp): Phänotyp
  L ← ∅
  for i ← 0 to |G|-1
    P ← sucheFreienPlatz(G(i))
    if P ∈ ∅
      return L
    L ← L ∪ (G(i) ∪ P)
  return L
```

Sobald die erste Kiste nicht mehr in den Container passt, wird abgebrochen. Das Ergebnis der Dekodierung ist die konkrete Beladung L.

$$L = dec(I.G) = \{(idx_1, \mathbf{p_1}, r_1), \ldots, (idx_k, \mathbf{p_k}, r_k)\} \text{ mit } k \leq n \tag{7}$$

Die Suche eines freien Platzes könnte naiv so umgesetzt werden, dass für jede Position (x, y, z) im Container geprüft wird, ob die Kiste komplett auf ein oder mehreren anderen Kisten stehen würde und ob die zu beladende Kiste eine andere geometrisch schneiden würde. Der Rechenaufwand steigt hier exponentiell mit der Anzahl der schon beladenen Kisten an. Eine andere Lösung musste also gefunden werden.

Die hier benutzte Platzsuche basiert auf dem Paradigma der dynamischen Programmierung und benutzt eine Datenstruktur, welche die Bedingung, dass jede Kiste vollständig auf anderen Kisten oder dem Boden stehen muss, direkt garantiert.

Der Beladungszustand des Containers wird auf eine Matrix M abgebildet, wobei die Indizes den (x, z)-Koordinaten und die Werte der aktuellen Beladungshöhe entsprechen. Die Suche nach einem geeigneten Platz beschränkt sich nun auf die Suche nach einem Rechteck mit dem Kistenboden als Größe in der Matrix, wobei alle Werte innerhalb des

Rechtecks den selben Wert und damit die selbe Beladungshöhe haben müssen. Rechtecke, dessen Werte größer als $H_C - H_i$ sind, können sofort ignoriert werden, da die Kiste über die Containerdecke ragen würde.

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 4 & 2 & 2 & 2 & 0 & 0 \\ 4 & 4 & 2 & 2 & 2 & 0 & 0 \\ 0 & 0 & 2 & 2 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \tag{8}$$

$$R = \left\{ \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 2 & 2 \\ 2 & 2 \\ 2 & 2 \end{pmatrix}, \begin{pmatrix} 2 & 2 \\ 2 & 2 \\ 2 & 2 \end{pmatrix} \right\} \tag{9}$$

In diesem Beispiel ist ein Rechteck der Größe 2×3 gesucht. Alle Elemente in R sind gültig, wobei der Algorithmus beim ersten Treffer abbricht und keine weiteren Eigenschaften, wie z.B. maximale Berührungsfläche, untersucht. Der Suchalgorithmus beginnt stets bei (0,0), prüft eine Zeile, und führt die Suche in der folgenden Zeile fort. Dadurch ist auch garantiert, dass Kisten nicht wahllos, sondern immer an andere Kisten bzw. den Containerwänden platziert werden. Sobald eine Kiste platziert wird, werden die Werte des Rechtecks in der Matrix um die Kistenhöhe erhöht. Die Laufzeit für das Finden eines Rechtecks beträgt hier $O(B_C \cdot B_T)$.

Anmerkung: Es ist zu beachten, dass dieser Algorithmus einen hohen Speicherverbrauch hat. Bei jedem Dekodiervorgang muss ein zweidimensionales Array der Größe $B_C \times B_T$ angelegt werden, was bei Testinstanzen mit einer Containergröße von 10000×10000 zu einem Speicherverbrauch von ca. 500MB führt (JVM). Allerdings haben die in der Literatur am meisten benutzten Instanzen eine Größe von 233×587 und $[2000,2800] \times [3000,6000]$, weswegen dies hier kein besonderes Problem darstellte.

2.3 Fitnessfunktion

Die Fitness eines Phänotyps ist das Verhältnis der Summe der Volumina aller aufgeladenen Kisten zum Containervolumen. Es stellt damit die prozentuale Raumnutzung dar und eignet sich zum Vergleich mit Ergebnissen aus der Literatur.

$$F(L) = \frac{\sum_{i \in L.idx} B_i \cdot H_i \cdot T_i}{B_C \cdot H_C \cdot T_C} \tag{10}$$

3 Selektion

Das Beladungsproblem hat die Eigenschaft sehr viele lokale Optima zu besitzen. Um diesem Problem zu begegnen, war schnell klar, dass die Selektion einen hohen Fokus auf Erforschung haben und erst zum Ende eine Feinabstimmung bewirken sollte.

Basierend auf einer Evaluation mehrerer Methoden der Fitnessskalierung von Hopgood und Mierzejewska [5], wurde deutlich, dass eine reine fitnessproportionale oder

rangbasierte Selektion, oder auch das stochastische universelle Sampling, in den wenigsten Fällen bei Gütelandschaften mit vielen lokalen Optima zu guten Ergebnissen führen. Die Skalierung von Fitnesswerten wird vorallem angewandt, um den Selektionsdruck zu Beginn niedrig zu halten (Erforschung), und ihn mit fortschreitender Generationszahl stark ansteigen zu lassen (Feinabstimmung).

In diesem Projekt wurde die Sigmaskalierung in Verbindung mit stochastischem universellen Sampling angewandt, da bei dieser Methode der zusätzliche Rechenaufwand im Rahmen bleibt und gleichzeitig sehr gute Ergebnisse bei der 2D-Schwefel sowie der 2D-Griewank Funktion von Hopgood und Mierzejewska erzielt wurden. Ein weiterer Vorteil ist, dass die Methode nicht von einem Parameter abhängt, der erst hätte gefunden werden müssen.

Bei der Sigmaskalierung wird die Durchschnittsfitness $\overline{F}$ und dessen Standardabweichung σ in der Population berücksichtigt. Solange die Standardabweichung relativ groß ist, stellt die Skalierung sicher, dass diese solange wie möglich erhalten bleibt, um einen möglichst großen Suchbereich zu erforschen. Durch den geringen Selektionsdruck passierte es häufig, dass sehr gute Individuen verloren gingen und nicht wieder gefunden wurden. Deshalb wird hier zusätzlich Elitismus mit einem Individuum benutzt, sodass das beste Individuum auf jeden Fall in die nächste Generation gelangt. Wenn die Population dann langsam konvergiert, wird die Standardabweichung geringer, was wiederum zur Folge hat, dass Individuen mit einer Güte über dem Durchschnitt stark bevorzugt werden und damit der Selektionsdruck ansteigt.

$$F_S = \begin{cases} 1 & , \sigma = 0 \\ 1 + \frac{F - \overline{F}}{2 \cdot \sigma} & , \sigma > 0 \end{cases} \tag{11}$$

Anmerkung: Bei sehr schlechten Individuen kann es passieren, dass F_S einen negativen Wert annimmt. In diesem Fall wird die skalierte Fitness durch eine sehr kleine positive Fitness ersetzt, hier 0,1.

4 Operatoren

In diesem Abschnitt werden die benutzten bzw. entwickelten Operatoren in der Reihenfolge ihrer Integration in das Projekt beschrieben.

4.1 Rekombination

Die Wahl des Rekombinationsoperators beschränkt sich auf solche, die garantieren, dass keine Elemente nach der Rekombination doppelt vorkommen. Hierbei ist zu beachten, dass ein Element von einem anderen nur anhand seiner eindeutigen ID (Kistenindex idx_i) unterschieden wird – die Rotationskomponente wird ignoriert.

Der Zweck der Rekombination ist vor allem die Verbindung guter Eigenschaften zweier Individuen. Der Autor konnte kein Verfahren finden, welches dieses Kriterium für das Beladungsproblem erfüllt. Der Hauptgrund ist, dass die Dekodierung sehr empfindlich

auf Änderungen der Permutation reagiert. Umso weiter am Anfang der Permutation ein Element verändert wird, umso größer ist die Wahrscheinlichkeit, dass eine ab diesem Element komplett verschiedene Beladung resultiert. Da bei der Rekombination größere Teile geändert werden, führt dies häufig zu schlechteren Individuuen, die erst durch Mutation optimiert werden müssen. Folglich ähnelt die Wirkung einer Rekombination auf dieses Problem einer mischenden Mutation.

Dennoch wurde eine angepasste Version des PMX-Operators [4] (partially mapped crossover) benutzt, da positive Effekte nicht endgültig ausgeschlossen werden konnten. Selbst, wenn die Rekombination nur wie eine mischende Mutation wirkt, hätte dies keine Nachteile, da dadurch zumindest eine über die Zeit konstante Erforschung des Suchraums gewährleistet ist. Der PM-Crossover wurde dahingehend angepasst, dass das linke Ende des jeweils zu übernehmenden Teilstücks stets beim ersten Element beginnt. Dadurch wird eine übermäßige Zerstörung des Genotyps verhindert. Die gewählte Rekombinationswahrscheinlichkeit beträgt $p_x = 0{,}5$. Werte über 0,5 hatten einen zu großen Einfluss auf die Durchschnittsfitness und sorgten dafür, dass diese sich nicht wesentlich über viele Generationen verbesserte – zu viele Individuen wurden zerstört.

Anmerkung: Es wurde keine Abbildungsrekombination benutzt, da diese in dem verwendeten Framework nicht mitgeliefert wird und kein weiterer Vorteil für eine spezielle Art der Rekombination erkannt wurde.

4.2 Mutation

Da hier die Rekombination nur bedingt zu besseren Individuen führt, lag der Fokus auf der Entwicklung geeigneter Mutationsoperatoren. Im Folgenden wird jeder benutzte bzw. entworfene Operator beschrieben und dessen Sinnhaftigkeit untersucht.

4.2.1 Vertauschung

Der wohl einfachste und naheliegendste Mutationsoperator für Permutationen ist die Vertauschung. Es werden zufällig zwei Elemente im Genotyp gewählt und miteinander vertauscht.

Die Umsetzung dieses Operators lässt gewisse Freiräume offen. Zum einen ist festzulegen, wie viele Vertauschungen in einem Aufruf durchgeführt werden. Zum anderen muss entschieden werden, ob die Positionen zufällig gewählt werden oder abhängig voneinander sein sollen. Dies ergibt die zwei Parameter Vertauschungsanzahl N und Schrittweite S. Soll genau ein Element mit irgendeinem anderen getauscht werden, wäre $N = 1$ und $S \sim U(1, |I.G|)$, wobei U für eine Uniformverteilung steht. Das erste Element wird dabei immer uniform gewählt und ist nicht von den Parametern abhängig.

Da schon kleine Veränderungen die Fitness stark beeinflussen können, wurde hier $N = S \sim P_1$ benutzt, wobei P_λ der Poisson-Verteilung wie in Abbildung 1 entspricht. Damit ist gewährleistet, dass die Anzahl der Vertauschungen und die Schrittweite stets gering bleiben, aber dennoch mit niedriger Wahrscheinlichkeit auch höher sein können. Durch Benutzung dieser Verteilung ist ebenfalls eine Mutationswahrscheinlichkeit als

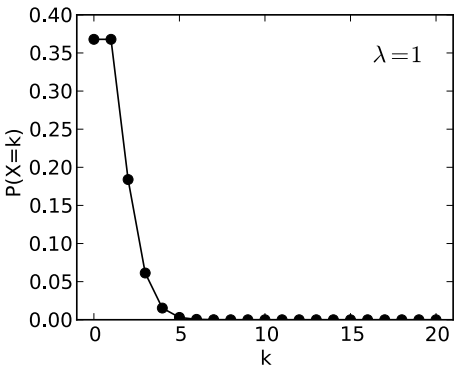

Abbildung 1: Poisson-Verteilung mit $\lambda = 1$

dritter Parameter nicht zwingend nötig, da $P_1(X = 0) \approx 0{,}4$ ist und somit häufig keine Vertauschung auftritt – im Fall $N = 0$ oder $S = 0$.

4.2.2 Einzel-Rotation

Da die Rotationsinformation Teil des Genotyps ist, muss diese ebenfalls mutiert werden. Dafür wurde die Einzel-Rotation entwickelt, welche abhängig von einer Mutationswahrscheinlichkeit zufällig ein Element der Permutation wählt und dann dessen Rotationskomponente zufällig ändert. Die Wahl der neuen Rotation erfolgt mit einer Uniformverteilung, wobei es erlaubt ist, dass die alte Rotation wieder gewählt wird. Für die Mutationswahrscheinlichkeit wurde mit $p_r = 0{,}3$ ein Wert gefunden, bei dem eine frühe Konvergenz verhindert und die Erforschung des Suchraums gefördert wird.

Anmerkung: Dieser Operator wurde anfangs noch ohne Mutationswahrscheinlichkeit umgesetzt. Um dennoch eine solche zu simulieren, wurde der *Split-Evolution* Operator des benutzten Frameworks verwendet. Dieser erlaubt es, zwei Tupel von Operatoren jeweils auf einen Teil der Population getrennt anzuwenden. Auf 90% der Population wurde der PM-Crossover und die Vertauschungsmutation angewandt. Die restlichen 10% wurden der Einzel-Rotation unterzogen.

4.2.3 Ersetzung

Zum Zeitpunkt der Implementierung aller vorheriger Operatoren benötigte ein Durchlauf des genetischen Algorithmus mit 60 Generationen und einer Populationsgröße von 40 knapp zwei Stunden (Intel Core 2 Duo mit 2,8 GHz). Unter diesen Voraussetzungen entstand häufig das in Abbildung 2 zu sehende Szenario. Es ist zu erkennen, dass die Population schnell konvergiert und offensichtlich keine hohe Diversität erhalten werden konnte.

Aufgrund späterer Erfahrungen hat sich gezeigt, dass die Erhöhung der Populationsgröße der einzig gute Kompromiss ist. Da die Geschwindigkeit allerdings zu diesem Zeitpunkt noch relativ niedrig war, hat der Autor nach anderen Lösungen gesucht. Eine davon ist die Implementierung des Ersetzungsoperators. Abhängig von einer Mutations-

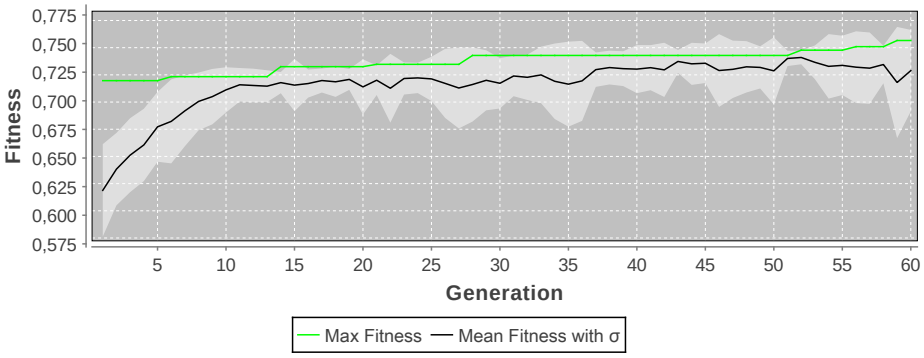

Abbildung 2: Fitnessverlauf mit 40 Individuen und den Operatoren PMX, Vertauschung und Einzel-Rotation; Testinstanz THPACK8-2 aus [2]

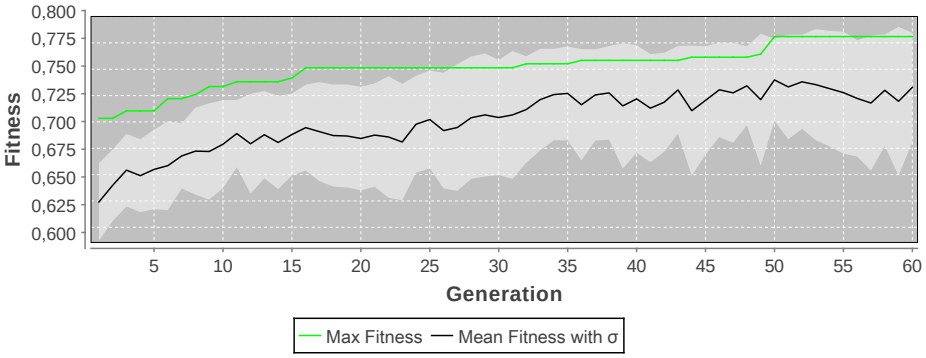

Abbildung 3: Fitnessverlauf mit 40 Individuen und den Operatoren PMX, Vertauschung, Einzel-Rotation und Ersetzung; Testinstanz THPACK8-2

wahrscheinlichkeit ersetzt dieser Operator ein Individuum komplett durch ein zufällig neu generiertes, wobei es keine Abhängigkeiten zwischen beiden Genotypen gibt. Das Ziel dabei ist, dass immer wieder neues Genmaterial einfließen soll, welches letztlich die Diversität begünstigen soll. Die Mutationswahrscheinlichkeit wurde hier mit $p_e = 0{,}1$ festgelegt, da bei höheren Werten der Suchfortschritt immer weiter sinken würde. In Abbildung 3 ist zu sehen, dass der Ersetzungsoperator die Standardabweichung relativ konstant hält und somit für eine gute Diversität sorgt.

4.2.4 Typ-Rotation

Durch diverse Optimierungen am Algorithmus, der Ausführungsumgebung sowie der Kompilierung konnte die Geschwindigkeit mehr als verdoppelt werden, sodass die Populationsgröße von 40 auf 100 bei gleichbleibender Gesamtzeit erhöht werden konnte. Auch wenn dies zur weiteren Steigerung der Diversität und teilweise besseren Ergebnissen führte, waren die erreichten Werte noch recht bescheiden im Vergleich zu denen in der Literatur (z.B. $F = 0{,}91$ für Probleminstanz THPACK8-2 durch Eley [1]). Das Ziel war es nun, möglichst geschickt weitere Mutationsoperatoren zu entwerfen, welche spe-

Tabelle 1: Fitnesswerte, rotierte sowie ausgelassene Kisten für THPACK1-10 nach 5 Minuten mit und ohne Typ-Rotation

	Fitness	rotierte Kisten	ausgelassene Kisten
ohne Typ-Rotation	0,8386	3	23
	0,8358	4	21
	0,8440	3	27
	0,8437	3	24
	0,8165	5	28
	0,8250	2	21
	0,8386	3	23
mit Typ-Rotation	0,8494	28	15
	0,8611	82	20
	0,8386	38	23
	0,8511	23	25
	0,8511	46	21
	0,8541	4	22
	0,8226	55	26

ziell auf das Beladungsproblem bezogen sein und besondere Eigenschaften von typischen Probleminstanzen berücksichtigen sollten.

Beim Betrachten des für Instanz THPACK1-10 visuell dargestellten Lösungskandidaten in [1] wurde offensichtlich, dass eine gute Lösung vor allem dann entsteht, wenn Kisten des selben Typs aneinander oder aufeinander gestapelt werden. Dieser Umstand war direkt von der Heterogenität des gegebenen Kistenbestands abhängig. Eine schwache Heterogenität, d.h. wenige verschiedene Kistengrößen, konnte vorteilhaft benutzt werden, um in einem gewissen Maße eine Stapelung zu forcieren bzw. zu fördern und damit eine effiziente Beladung zu erreichen.

Da Kisten gleicher Größe in den meisten Fällen nur aufeinander gestapelt werden können, wenn sie die selbe Rotation haben, wurde der Operator Typ-Rotation entworfen. Dieser wählt abhängig von einer Mutationswahrscheinlichkeit zufällig eine Kiste im Genotyp aus und ändert genau wie die Einzel-Rotation die Rotationskomponente. Zusätzlich wird allen weiteren Kisten im Genotyp mit derselben Größe die selbe Rotationskomponente zugewiesen. Die Hoffnung ist, dass zufällig in der Permutation nebeneinander liegende Kisten selber Größe (aber unterschiedlicher Rotation) nun gestapelt werden können, da die Rotationskomponenten gleichgesetzt wurden. Einige Tests mit dem Operator lieferten keine offensichtlichen Verbesserungen, weshalb ein Hypothesentest durchgeführt wurde.

Getestet wurde mit THPACK1-10 und den bisher genannten Parameterwerten. Nach jeweils 5 Minuten wurde die Evolution angehalten. Aus Tabelle 1 ergibt sich für den t-Test $t = 1{,}9992$, was als nicht wirklich statistisch signifikant interpretiert werden kann. Allerdings fällt in den Testdaten der letzte Fitnesswert auf, welcher als Ausrutscher gesehen werden könnte. Wird dieser Wert ignoriert, ist $t = 3{,}253$, was statistisch als sehr

signifikant betrachtet werden kann.

Es folgt, dass der Operator eine leichte Tendenz zu besseren Lösungskandidaten aufweist. Dennoch sind die Unterschiede zu gering. Eine Ursache dafür könnte sein, dass die Wahrscheinlichkeit, dass gleichgroße Kisten in der Permutation direkt nebeneinander liegen, zu niedrig ist.

4.2.5 Clusterbildung

Um das Potential der Typ-Rotation auszunutzen, war es also wichtig, gleichgroße Kisten möglichst nah oder direkt nebeneinander in der Permutation zu platzieren. Zu diesem Zweck wurde der Clusterbildungsoperator entworfen. Der Ablauf des Operators ist im Folgenden zu sehen:

```
cluster(G: Genotyp): Genotyp
  R ← clone(G)
  refIdx ← U(0,|G|-1)
  refBox ← G(refIdx)
  freeIdx ← -1
  for i ← idx + 1 to |G|-1
    box ← R(i)
    if box.size = refBox.size && box.rotation = refBox.rotation
      if freeIdx != -1
        swap(R, freeIdx, i)
        freeIdx ← freeIdx + 1
    else if freeIdx = -1
      freeIdx ← i
  return R
```

Der Operator wählt also zunächst zufällig eine Referenzkiste. Danach werden alle darauf folgenden Kisten auf Typgleichheit untersucht. Typgleichheit ist hier gegeben, wenn die Größe und Rotation übereinstimmen. Sobald eine Kiste mit verschiedenem Typ gefunden wurde, gilt dies als ein freier Platz. Wird nun wieder eine typgleiche Kiste gefunden, wird diese mit der auf dem freien Platz getauscht. Das Ergebnis ist, dass alle Kisten selber Größe und Rotation, die hinter der Referenzkiste liegen, direkt hinter diese platziert werden und damit eine Kette entsteht. Die Hoffnung ist, dass typgleiche Ketten zu einem besseren Lösungskandidaten führen.

Als Mutationswahrscheinlichkeit wurde $p_c = 0{,}3$ gewählt. Die Durchschnittsfitness hat sich bei THPACK1-10 tatsächlich erheblich verbessert und betrug bei 5 Durchläufen 0,8751 – im Vergleich zu 0,8469, als nur Typ-Rotation benutzt wurde. Ein Hypothesentest ist hier nicht nötig. Allerdings wäre es interessant, zu prüfen, ob der Clusterbildungsoperator auch ohne die Typ-Rotation gute Ergebnisse liefert, oder ob beide Operatoren nur im Zusammenspiel gut funktionieren. Tabelle 2 zeigt die experimentellen Ergebnisse, woraus sich für den t-Test $t = 3{,}7225$ ergibt, was als statistisch sehr signifikant gesehen werden kann. Die Werte im zweiten Teil lassen auch erkennen, dass eine Evolution mit Clusterbildung ohne Typ-Rotation sehr häufig in bestimmten lokalen Optima landet.

Tabelle 2: Fitnesswerte, rotierte sowie ausgelassene Kisten für THPACK1-10 nach 5 Minuten mit und ohne Typ-Rotation und mit Clusterbildung

	Fitness	rotierte Kisten	ausgelassene Kisten
Typ-Rotation und Clusterbildung	0,8756	58	26
	0,8656	49	36
	0,8738	32	29
	0,8716	11	34
	0,8890	40	25
nur Clusterbildung	0,8593	2	35
	0,8615	3	35
	0,8593	1	35
	0,8615	3	35
	0,8615	3	35

5 Ergebnisse

Das Projekt wird abgeschlossen, indem für eine Reihe von leichten sowie schwierigen Probleminstanzen ein Vergleich mit Ergebnissen aus der Literatur hergestellt wird. Dem Autor ist bewusst, dass es sich nicht nur um evolutionäre Algorithmen handelt, sondern auch um komplexe deterministische Heuristiken. Da ein umfassender Algorithmenvergleich den Rahmen dieses Projekts übersteigen würde, werden hier lediglich die Fitnesswerte zum Vergleich aufgelistet. Die folgenden Probleminstanzen wurden dabei benutzt.

Instanz [2]	Container	Typen	Kisten	Schwierigkeit	eigenes Limit
THPACK1-10	$233 \times 220 \times 587$	3	130	schwer	5 Minuten
THPACK8-1	$2000 \times 1000 \times 3000$	7	100	einfach	5 Minuten
THPACK8-2	$2000 \times 1000 \times 3000$	7	175	schwer	30 Minuten
THPACK8-4	$2000 \times 1100 \times 3000$	7	100	einfach	5 Minuten
THPACK8-5	$2000 \times 900 \times 3000$	7	120	einfach/mittel	10 Minuten
THPACK8-7	$2400 \times 1300 \times 3500$	8	200	schwer	30 Minuten
THPACK8-12	$2400 \times 1000 \times 3200$	7	120	mittel	30 Minuten

Das beste Ergebnis (hier in %) nach fünfmaliger Ausführung ist rechts aufgelistet. Zahlen in Klammern bedeuten ausgelassene Kisten.

Instanz	Ngoi et al. (1994)	Bischoff et al. (1995)	Bischoff und Rat. (1995)	Eley (2002)	selbst
1-10	-	-	-	88.7 (o.A.)	88.9 (25)
8-1	62.5	62.5	62.5	62.5	62.5
8-2	80.7 (54)	89.7 (23)	90.0 (35)	90.8 (53)	83.93 (24)
8-4	55.0	55.0	55.0	55.0	55.0
8-5	77.2	77.2	77.2	77.2	77.2
8-7	81.8 (10)	83.9 (1)	78.7 (18)	84.7	76.7 (20)
8-12	78.5	76.5 (3)	78.5	78.5	77.4 (2)

Literatur

[1] Eley, M.: *Solving container loading problems by block arrangement.* European Journal of Operational Research, 141(2):393–409, 2002.

[2] Fekete, S.P. und J.C. van der Veen: *PackLib2: An integrated library of multi-dimensional packing problems.* `http://www.ibr.cs.tu-bs.de/alg/packlib/instances.shtml` [17.02.2011].

[3] Gehring, H. und A. Bortfeldt: *A genetic algorithm for solving the container loading problem.* International Transactions in Operational Research, 4(5-6):401–418, 1997.

[4] Goldberg, D.E. und R. Lingle Jr: *Alleles, loci, and the traveling salesman problem.* In: *Proceedings of the 1st International Conference on Genetic Algorithms*, Seiten 154–159. L. Erlbaum Associates Inc., 1985.

[5] Hopgood, A.A. und A. Mierzejewska: *Transform ranking: a new method of fitness scaling in genetic algorithms.* In: *Research and Development in Intelligent Systems Xxv: Proceedings of Ai-2008, the Twenty-eighth Sgai International Conference on Innovative Techniques and Applications of Artificial Intelligence*, Seiten 349–354. Springer-Verlag New York Inc, 2008.

[6] Mack, D. und A. Bortfeldt: *Eine Heuristik für das mehrdimensionale Bin Packing Problem.* 2008.

[7] WenQi, H. und HE Kun: *A pure quasi-human algorithm for solving the cuboid packing problem.* Science in China Series F: Information Sciences, 52(1):52–58, 2009.