Kurs-ID:	BA-INF 051
Kurs:	Wissensentdeckung und Maschinelles Lernen
Semester:	Sommersemester 2021
Betreuer/-in:	Vanessa Toborek

UNIVERSITÄT BONN ML AI lab

Nature Inspired Algorithms für das Bin Packing Problem

Lars Gehlen Tobias Mette Tim Schätz

15. September 2021

Zusammenfassung

Die vorliegende Arbeit beschreibt drei naturinspirierte Optimierungsalgorithmen. Die Algorithmen werden verwendet um das zweidimensionale Bin Packing Problem zu lösen. Es werden theoretische Grundlagen und Implementierungen der Particle Swarm Optimization, der Ant Colony Optimization und des genetischen Algorithmus vorgestellt. Die Algorithmen werden hinsichtlich ihrer Laufzeiten und Qualität der Lösungen untersucht und mit den Heuristiken First Fit und First Fit Decreasing verglichen. Die Auswertung der Algorithmen zeigt, dass die gewählten Implementierungen der Ant Colony Optimization und des genetische Algorithmus für die Lösung des zweidimensionalen Bin Packing Problems geeignet sind, dabei aber nur geringfügig bessere Ergebnisse erzielen als die First Fit Decreasing Heuristik.

1 Einleitung

Technologieunternehmen gehören zu den finanziell erfolgreichsten Unternehmen auf der Welt und im Zeitalter der Digitalisierung gilt es immer größere Datenmengen und aufwendigere Probleme möglichst effizient zu lösen. NP-Schwere Probleme bezeichnen Entscheidungsprobleme welche nur in nichtdeterministischer polynomieller Zeit gelöst werden. Es wurden noch keine Algorithmen gefunden welche all diese Probleme optimal und schnell lösen können und bei steigender Problemgröße sorgen oft lange Laufzeiten oder hoher Rechenaufwand für wenig effiziente Optimierungsansätze.

Eines dieser NP-Schweren Probleme ist das Bin Packing Problem, bei welchem Objekte verschiedener Größen auf Behälter aufgeteilt werden. Die Container besitzen eine Maximalkapazität und es gilt alle Objekte zu verteilen und dabei die Anzahl der benutzten Container zu minimieren.

Zum Lösen des Bin Packing Problem untersuchen und implementieren wir drei naturinspirierte Metaheuristiken. Der von Evolution inspirierte genetische Algorithmus, die Ameisenstraßen simulierende Ant Colony Optimization und die Schwarmverhalten nachbildende Particle Swarm Optimization. Im Laufe dieser Arbeit werden die drei Algorithmen vorgestellt und Implementationen für das Bin Packing Problem entwickelt. Bereits entwickelte Realisierungen der Heuristiken werden angesprochen, Probleme beim Anwenden der Metaheuritsiken werden dargestellt und die verschiedenen Implementierungen werden auf ihre Optimalität und Laufzeit untersucht. Zum Schluss erfolgt ein Ver-

gleich zu den Lösungsmethoden First Fit und First Decreasing auf das Bin Packing Problem. Problematiken der Particle Swarm Optimization Implementierung werden aufgezeigt und die möglichen Nutzen der Genetic Algorithm und Ant Colony Optimization als zukünftige Lösungsansätze für das Bin Packing Problem erklärt.

2 Problemstellung

Das Bin Packing Problem ist ein kombinatorisches Optimierungsproblem. Die hier beschriebene Variante ist ein zweidimensionales Bin Packing Problem. Gegeben ist eine Menge von Objekten O und eine Containergröße B. Jedes Objekt ist definiert durch sein Gewicht g und Volumen v. Die Container besitzen ein maximales Gewicht und Volumen.

$$O = \left\{ o \mid o = (g, v)^T \in \mathbb{N}^2_{>0} \right\}, o_k = \begin{pmatrix} g_k \\ v_k \end{pmatrix}$$

$$B = \left\{ b \mid b = (g_{max}, v_{max}) \in \mathbb{N}^2_{>0} \right\}, |B| = n, n \in \mathbb{N}^+$$

Ein Lösungskandidat wird bei einem Problem mit n Objekten als $j \times n$ Matrix beschrieben.

$$\vec{x} \in \{0, 1\}^{i \times j}$$

$$X = \left[\vec{x}_1^T, ..., \vec{x}_n^T \right], X_l = \text{Spalte}$$

Die Summe der Gewichte und Volumen aller Objekte in einem Container dürfen nicht das maximale Gewicht oder Volumen des Containers übersteigen.

$$\sum_{l=1}^{n} (OX_l) \leq \begin{pmatrix} b_g^n \\ b_v^n \end{pmatrix}$$

Das Ziel ist es alle Objekte auf möglichst wenige Container zu verteilen.

$$\min \sum_l f(X_l), f(X_l) = \begin{cases} 1 & \sum_i X_{l_i} > 0 \\ 0 & \end{cases}$$

3 Particle Swarm Optimization

Lars

Die Particle Swarm Optimization (PSO) ist ein von biologischem Schwarmverhalten inspiriertes Verfahren zur Lösung von Optimierungsproblemen. Erstmals wurde der Algorithmus 1995 von James Kennedy und Russel Eberhart vorgestellt. (J. Kennedy und Eberhart, 1995)

3.1 Inspiration

Das ursprüngliche Ziel von James Kennedy und Russel Eberhart war die Simulierung von Sozialverhalten, welches man zu Beispiel in Vogel- und Fischschwärmen beobachten kann. In diesen Schwärmen synchronisieren sich die Individuen in ihren Bewegungen und ihrem Verhalten. Im Laufe von Tests, wie das Finden von einem Ort zu welchem sich der Schwarm bewegen soll, wurde die erste PSO Heuristik für Optimierungsprobleme entworfen. Seit der ersten Vorstellung wurde der Algorithmus bis zu der heutigen genutzten Metaheuristik weiterentwickelt. (Poli, James Kennedy und Blackwell, 2007)

3.2 Algorithmus

Der Algorithmus benutzt einen Schwarm an Partikeln, welche Lösungskandidaten darstellen und die in einem Suchraum mithilfe einer Fitnessfunktion ausgewertet werden. Ein Partikel i wird durch seinen Positionsvektor $\vec{x}_i$ in dem Suchraum beschrieben. Neben dem Positionsvektor besitzt ein Partikel noch seinen Geschwindigkeitsvektor $\vec{v}_i$ und einen Vektor $\vec{pbest}_i$ für den persönlichen Bestwert, also die Position mit der besten Fitness, welche das Partikel bis zu dem Zeitpunkt erreicht hat. Der gesamte Schwarm speichert weiterhin über den Vektor $\vec{gbest}$ die während der Laufzeit von einem Partikel gefundene beste Position. Für die Bewegung besitzt der Schwarm Koeffizienten. Die Trägheit w bezieht sich auf die Geschwindigkeitsvektoren. Der Koeffizient c_p beschreibt die Stärke der Bewegung Richtung des persönlichen Bestwert $\vec{pbest}_i$. Analog existiert für den globalen Bestwert $\vec{gbest}$ der Koeffizient c_g.

Für die Bewegung in einem Iterationsschritt generiert der Algorithmus für jedes Partikel zwei Zufallswerte r_1 und r_2 bevor die folgenden beiden Formeln genutzt werden:

- $\vec{v}_i = w \cdot \vec{v}_i + c_p \cdot r_1 \cdot (\vec{pbest}_i - \vec{x}_i) + c_g \cdot r_2 \cdot (\vec{gbest} - \vec{x}_i)$
- $\vec{x}_i = \vec{x}_i + \vec{v}_i$

Mit der ersten Formeln wird für das Partikel der neue Geschwindigkeitsvektor abhängig von den Koeffizienten, seinem alten Geschwindigkeitsvektor und der Distanz zum persönlichen und globalen Bestwert erschaffen. Die neue Position des Partikels berechnet sich mit diesem neu berechneten Geschwindigkeitsvektor. Nachdem ein Partikel seine Bewegung vollzogen hat kontrolliert der Algorithmus die neue Position mithilfe der problembezogenden Fitnessfunktion f und falls diese besser als der alte persönliche Bestwert ist wird der Bestwert aktualisiert. Analog wird der globale Bestwert nach Abschluss der Bewegungen aller Partikel geupdated.

1. Initialise the Swarm with random positions for all Particles and set $\vec{pbest}_i$ and the gbest
2. Loop (until criterion met):
3. For every Particle i:
4. $\vec{v}_i = w \cdot \vec{v}_i + c_p \cdot r_1 \cdot (\vec{pbest}_i - \vec{x}_i) + c_g \cdot r_2 \cdot (\vec{gbest} - \vec{x}_i)$
5. $\vec{x}_i = \vec{x}_i + \vec{v}_i$
6. If $f(\vec{x}_i)$ better than $f(\vec{pbest}_i)$, update $\vec{pbest}_i = \vec{x}_i$
7. For every Particle i:
8. If $f(\vec{x}_i)$ better than $f(\vec{gbest})$, update $\vec{gbest} = \vec{x}_i$

3.2.1 Bedeutung der Koeffizienten

Damit der Algorithmus seine besten Ergebnisse findet, gilt es die Koeffizienten auf das Problem anzupassen. Die Trägheit w beeinflusst die Geschwindigkeit mit welcher sich das Partikel im Suchraum bewegt, bei niedrig gewählter Trägheit bewegt sich das Partikel langsamer und kann genauer seine Umgebung untersuchen, aber seine allgemeine Bewegung im Suchraum, besonders zu dem im Moment untersuchten globalen Optimum wird eingeschränkt. Wenn die Trägheit hoch gewählt wird können die Partikel besser Optima im Suchraum finden, doch der Schwarm wird möglicherweise wegen der höheren Bewegungsrate instabil und Partikel können sich nicht in den Optima einpendeln. Manche Implementierungen starten mit einer hohen Trägheit und senken diese über die Laufzeit.

Der lokale Koeffizient c_p legt fest wie stark die Partikel ihre lokalen Optima untersuchen. c_g lenkt die Konvergenz zum gefundenen globalen Optimum. Die beiden Koeffizienten sollten aneinander angepasst werden. Sollte c_g zu stark sein bewegt sich der Schwarm möglicherweise zu schnell zu diesem Optimum und kann andere Optima nicht auf eine bessere Lösung untersuchen. Ist wiederum c_p zu mächtig wird möglicherweise das Optimum nicht genug untersucht und die tatsächlich optimale Lösung nicht gefunden. Die beiden Koeffizienten können über die Algorithmuslaufzeit angepasst werden, zum Beispiel zu Beginn ein hoher c_p Wert welcher kleiner wird und ein niedriger c_g Wert der steigt. Dies gibt eine Suchphase am Anfang der Laufzeit bevor der Schwarm zum globalen Optimum konvergiert.

3.2.2 Alternativen

Seit der Erstvorstellung in 1995 wurden viele Ergänzungen, Variationen und Spezialisierungen zu der PSO entworfen. Die Trägheit w war nicht Teil der Erstvorstellung in 1995 und wurde von Yuhui Shi und B. Gireesha Obaiahnahatti in „A Modified Particle Swarm Optimizer“ (1998) vorgeschlagen. Ziel war ein besserer Ausgleich zwischen lokaler und globaler Suche.(Shi und Obaiahnahatti, 1998)

Untersucht wurden Nachbarschaften von Partikeln im Schwarm mit einem Nachbarschafts-Bestwert der den globalen Bestwert ergänzt oder ersetzt. Verschiedene Auswertungen wie binäre Systeme um mehr Probleme mit PSO behandeln zu können. Alternative Bewegungsformen wie eine Gleichverteilung abhängig von allen Werten. Die vielen Spezialisierungen und Entwicklungen zeigten mindestens gleichwertige Ergebnisse gegenüber der klassischen PSO.(Poli, James Kennedy und Blackwell, 2007)

3.3 Implementierung zum Bin Packing Problem

Die Lösung für das Bin Packing Problem sieht eine Matrix vor, welche jedem Objekt einen Container zuordnet und dabei die Anzahl der Container minimiert, ohne das Container überfüllt sind. Als solches wählen wir auch für die Partikel solche Verteilungen und eine Fitnessfunktion die eine Matrix auf ihre Legalität als ein Lösungskandidat und die Anzahl der Container auswertet.

Bei einem klassischen Ansatz findet die Bewegung für jedes Objekt in der Matrix über seine Zeile statt. Bei einer Implementierung könnten für die Entscheidung, in welchem Container das Objekt liegt, leicht Nachkommastellen ignoriert oder gerundet werden. Problematiken treten durch die Verteilungen der Objekte auf, die illegale Container aufgrund von Überfüllung bilden. Diese Verteilungen können einen Großteil des theoretischen Suchraums darstellen. Eine Matrix in einer Nachbarschaft, mit zum Beispiel nur einem Objekt in einem benachbarten Container, kann aus der optimalen Lösung einen für das Problem illegalen Lösungskandidaten erstellen. Einem Partikel mit illegaler Verteilung ist nicht garantiert über die klassische Bewegung im Folgeschritt einen legalen Lösungskandidaten zu finden. Auch seine Auswertung über eine Fitnessfunktion wird erschwert, da diese Lösungen niemals besser als legale Lösungen berechnet werden sollten und eine Bewertung von überfüllten Containern kompliziert ist. Die Fitnessfunktion wird nur bei der Wahl für $\vec{pbest}_i$ und $\vec{gbest}$ verwendet, falls mit der Initialverteilung nur legale Lösungskandidaten garantiert werden, müssen zukünftige illegale Verteilungen bereits nach Kontrolle auf Legalität von der Fitnessfunktion nicht weiter analysiert werden. Es kann dadurch aber weiterhin nicht verhindert werden, dass ein Teil des Schwarms in den meisten Iterationen nicht zur Verbesserung führen kann.

Ein Lösungsansatz zum Unterbinden überfüllter Container wäre die Bewegung zum nächstliegenden passenden Container mit genug Platz für das betroffene Objekt. Dadurch kann es sich aber entgegen der zugewiesenen Richtung bewegen oder das Ziel weit überschießen, der Geschwindigkeitsvektor ist der tatsächlichen Bewegung nicht mehr repräsentativ. Die direkte Bewegung mithilfe eines Geschwindigkeitsvektor zeigt sich ungeeignet für das Bin Packing Problem.

Laurent und Klement, 2019, beschreiben in „Bin Packing with priorities and incompatibilities using PSO: application in a health care community“ eine Alternative Bewegungsform über Sprünge mit Wahrscheinlichkeiten für das Bin Packing Problem. Ein Zufallswert wird generiert und ein Objekt entscheidet nun anhand von gegebenen Werten wie es sich bewegt. Ist der Zufallswert kleiner als der lokale Koeffizient, springt das Objekt nun an die Stelle, also den Container, in welchem es in seiner $\vec{pbest}_i$ Verteilung liegt. Die Koeffizienten c_p und c_g spiegeln also nun über Wahrscheinlichkeiten die Stärke zum Untersuchen lokaler Verteilungen oder zum konvergieren zur gefunden globalen Verteilung wieder. Ein Geschwindigkeitsvektor macht bei diesem Ansatz keinen Sinn, dadurch besitzt der Algorithmus aber keinen sonstigen Suchanteil. Nur mit Sprüngen in einen festen lokalen oder globalen Container oder durch Verharren in seiner aktuellen Position, würde sich das $\vec{gbest}$ vorgebende Partikel bis zu dessen Verbesserung von außerhalb nicht mehr bewegen und deshalb nicht weiter verbessern können. Es bietet sich die Einbindung einer Mutation in Form eines Koeffizienten c_m an. Die Mutation bestimmt zufällig oder mit Hilfe einer anderen Heuristik einen neuen Container für das Objekt. Die Wahrscheinlichkeit für c_m gibt die Anzahl der zusätzlichen Bewegungen an mit welchen die Partikel neue Lösungskandidaten finden. Für eine Implementierung mit nur legalen Verteilungen können auftretende Konflikte gelöst werden, indem die Objekte wiederholt über Mutationen verteilt werden falls diese nur die Bildung legaler Container garantiert.

Die neuen Verteilungen finden pro Partikel für jedes Objekt mithilfe von c_p, c_g, c_m und einem Zufallswert $r \in [0, 1[$ statt:

- falls $r < c_p$, das Objekt bewegt sich zu seiner Position im lokalen Bestwert $\vec{pbest}_i$
- falls $c_p \leq r < c_p + c_g$, das Objekt bewegt sich zu seiner Position im globalen Bestwert $\vec{gbest}$
- falls $c_p + c_g \leq r < c_p + c_g + c_m$, das Objekt bewegt sich zufällig oder über eine Hilfsheuristik in eine andere Position
- sonst, das Objekt bleibt an seiner Position

(Laurent und Klement, 2019)

3.3.1 Programmierung des Partikel

Für die Implementierung der Partikel ist eine Matrixdarstellung vorgesehen in welcher eine Zeile die Verteilung eines Objekts darstellt. Da diese Zeile nur aus Nullen und genau einer Eins besteht, welche den zugewiesenen Container über die Spalte vorgibt, wurde um Speicherplatz und Rechenzeit zu sparen eine vereinfachte Darstellung durch eine Liste gewählt. Diese Liste enthält die Objekten und jedes Element besitzt sein eigenes Gewicht und Volumen sowie einen Integerwert als Verweis für den aktuellen Container. Unterstützend existiert zum Auswerten für jedes Partikel eine Liste für die Container. Gespeichert sind ihre aktuellen Füllstände an Gewicht und Volumen sowie die Anzahl der

aktuell enthaltenen Objekte. Für Berechnungen sind dem Container sind sein maximales Gewicht und Volumen bekannt. Sollte ein Objekt bewegt werden bekommt der alte Container die Aufforderung das Gewicht und Volumen des Objektes von seinen Füllständen abzuziehen und die Anzahl der Objekte zu dekrementieren. Der neu ausgewählte Container bekommt das Gewicht und Volumen des Objektes übergeben und kontrolliert ob die neuen Füllstände die Maximalwerte überschreiten. Ist weder das Gewicht noch Volumen überschritten updated der Container die Füllstände und inkrementiert die Anzahl seiner Objekte. Falls aber mindestens einer der Maximalwerte überschritten wird, behält der Container seine alten Werte und gibt dem Partikel eine Rückmeldung um das Objekt in weiteren Schritten über Mutation zu verteilen. Über diese Implementierung wird die Objektbewegung vereinfacht und eine Garantie auf nur legale Lösungskandidaten gegeben.

Um die Verteilung des $\vec{pbest}_i$ zu speichern muss nur eine Kopie der darstellendern Objektliste gespeichert werden. Die einzige Information welche für ein Objekt aus dem $\vec{pbest}_i$ extrahiert werden muss sind ist der Integer welcher einen Container zuweist. Für vereinfachte Zugriffe speichert ein Partikel sonst nur seine aktuelle Fitness, die Anzahl der aktuell genutzten Container und die Fitness des $\vec{pbest}_i$.

3.3.2 Aufbau des Schwarms

Der Schwarm besitzt eine Liste an Partikeln welche in jedem Iterationsschritt für die Bewegung abgearbeitet wird. Über eine Liste ähnlich der $\vec{pbest}_i$ wird der $\vec{gbest}$ gespeichert, welcher nach den Bewegungen in einem Iterationsschritt angepasst wird. Die Koeffizienten c_p, c_g und c_m übergibt der Schwarm in jedem Schritt zusammen mit der Objektliste für die $\vec{gbest}$ Verteilung an die Partikel. Auf Basis dieser Daten kann das Partikel dann die Bewegung abschließen. Falls eine Veränderung der Koeffizienten vorgegeben wird, passiert dies linear über die Laufzeit.

3.3.3 Programmierung der Partikelbewegung

Bevor die interne Bewegung eines Partikel stattfindet wird jedem Objekt mithilfe eines Zufallswerts und den Koeffizienten die Bewegung zugeordnet. Nur die Objekte, welche eine Bewegung abschließen sollen, werden mit der Zuweisung aus ihren Containern entfernt. Dies senkt im Vergleich zur direkten Bewegung die Anzahl an Konflikten aufgrund überfüllter Container. Abhängig von dem zeitlich höheren Koeffizienten arbeitet das Partikel zuerst die lokalen oder globalen Bewegungen ab. Objekte die nach Abschluss der lokalen und globalen Bewegungen nicht zugeordnet sind werden noch vor den Objekten welchen Mutation zugewiesen wurde selbst über Mutation verteilt. Die internen Reihenfolgen der Objekte werden in jeder Iteration zufällig gewählt.

3.3.4 Fitnessfunktion

Für die Bewertung der Partikel zum Wählen von $\vec{pbest}_i$ und $\vec{gbest}$ gilt die Anzahl der benutzten Container als wichtigstes Element. Um auch Partikel mit gleicher Anzahl an Containern auswerten zu können ist in der Fitnessfunktion ein Vergleich an Füllständen eingebaut, Partikel mit mehr höheren Füllständen sollen besser bewertet werden.

$$f = \sum_{i=1}^{N} K + {volume_{max}}^2 - {volume_i}^2 + {weight_{max}}^2 - {weight_i}^2$$

Durch die oben beschriebene Fitnessfunktion wird ein Partikel mit hohen und niedrigen Füllständen niedriger bewertet als ein Partikel mit durchschnittlichen Füllständen. f wird von den Partikeln und dem Schwarm dementsprechend für die Bestwerte minimiert. N beschreibt die Menge an Containern der Partikel, welche mit mindestens einem Objekt belegt sind. Die Konstante K spiegelt die Bewertung der Anzahl Container wieder. K sollte ausreichend groß gewählt sein um seltene Randfälle zu verhindern, wo Verteilungen mit mehr Containern aufgrund ihrer Füllstände besser bewertet können als Verteilungen mit weniger benutzten Containern.

3.3.5 Hilfsheuristik für Zufallssprünge

Vier verschiedene Methoden zur Wahl des Containers bei den Mutationen wurden untersucht. Zufällig steht hier für zufällige Sprünge innerhalb der benutzten Container und einem leeren Container bis eine legale Verteilung gefunden wird. Eine Variation von Zufällig, hier als Random Fit bezeichnet, benutzt eine bestimmte Anzahl an Zufallssprüngen auf Container mit mindestens einem Element. Wird in der Anzahl der Sprüngen kein Container mit genügend Platz für das Objekt gefunden, öffnet das Objekt einen neuen Container. Weiterhin sind der First Fit und Best Fit Algorithmus als Hilfsheuristik implementiert.

Mutationswahrscheinlichkeit	Zufällig	Random Fit	First Fit	Best Fit
1%	4,0	4,0	3,0	7,7
10%	4,5	4,5	4,0	15,1
20%	5,6	5,7	5,4	21,7

Tabelle 1: Durchschnittliche Laufzeit (in Sekunden) aus 10 Durchläufen über ein Problem mit 500 Objekten

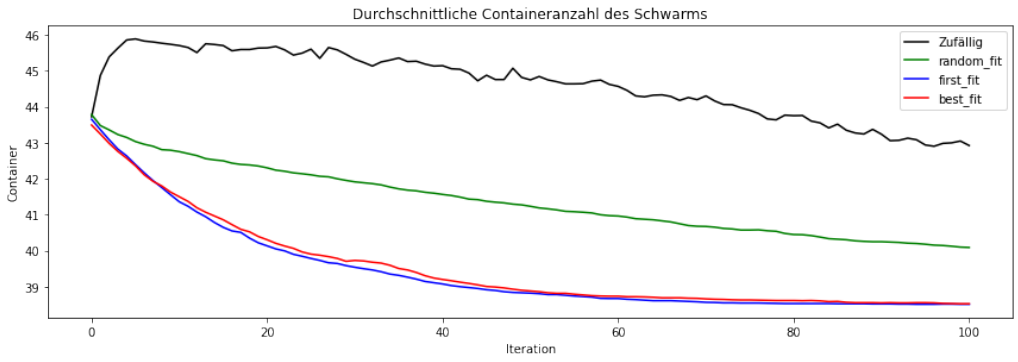

Abbildung 1: Hilfsheuristikvergleich über die durchschnittliche Containeranzahl des Schwarms mit einer Mutationswahrscheinlichkeit von 1% (Mittelwert aus 10 Durchläufen)

Im Vergleich geben Zufällig, Random Fit und First Fit ähnliche Laufzeiten aus. Best Fit verlängert die Dauer der Bewegungen durch viele zusätzliche Berechnungen deutlich. Trotz der längeren Laufzeit kann Best Fit im Durchschnitt keine besseren Ergebnisse als eine unterstützende First Fit Heuristik liefern. First Fit wird wegen einer Kombination aus

guter Laufzeit und guten Ergebnissen bei bereits niedrigen Mutationswahrscheinlichkeiten in weiteren Untersuchungen als Hilfsheuristik benutzt.

3.3.6 Initialverteilung

Für die Initialverteilung werden Objekte zufällig in eigene Container verteilt. Die Mutation erarbeitet in den ersten Iterationen erste Verteilungen in welchen die Bestwerte erstmals greifen. Die Ergebnisse sind bei genügend Laufzeit und Partikeln ausreichend gut um mit besseren Startverteilungen zu Beginn gleichzuziehen. Bessere Initialverteilungen können bei verschiedenen Problem mit unterschiedlichen Objektgrößen starke Startwerte liefern welche diese Implementation in seltenen Fällen nicht weiter verbessern kann.

3.3.7 Anzahl Partikel und Iterationen

Bei den Untersuchungen für die Anzahl von Partikeln und Iterationen zeigt sich die Instabilität des Algorithmus. Sobald zu einem Problem eine Hyperparameterwahl mit durchschnittlich guten Lösungen gefunden wird, kann in Folgeversuchen auf gleichen Hyperparametern oder sogar mit höherer Partikelanzahl oder mehr Iterationen weiterhin ein schlechteres Ergebnis auftreten. Für größere Untersuchungen wird dementsprechend ein Hyperparameterset gewählt, dessen Durchschnitt sich bei höheren Hyperparameterwerten nicht weiter verbessert und somit die Laufzeit nicht grundlos erhöht wird.

Die meisten Tests sind auf 50 Partikeln und 200 Iterationen durchgeführt, aber auch weniger Partikel und Iterationen können diesen gewählten Hyperparametern auf vielen Problemen gleichwertige Ergebnisse liefern. Trotz der schwachen Initialverteilung erreicht ein Schwarm sein bestes Ergebnis oft im ersten Abschnitt der Laufzeit.

3.3.8 Wahl der Koeffizienten

Bei der Wahl der Koeffizienten c_p und c_g kommen ebenfalls unbeständige Lösungen heraus. Gute Trends sind manchmal erkennbar, doch die Lösungsstärke im Verhältnis zu der Optimallösung auf verschiedenen Problemen bei unterschiedlichen Durchschnittsgrößen von Objekten variiert stark. Eine hohe Abhängigkeit von der Mutationswahrscheinlichkeit c_m ist bei vielen Problemen für bessere Ergebnisse ersichtlich. Aus den Tendenzen der Lösungsqualität auf verschiedenen Problemen wird die folgende Verteilung für $[c_p, c_g, c_m]$ gewählt: $[40\%, 20\%, 20\%]$, welches über die Laufzeit linear zu $[20\%, 40\%, 20\%]$ geändert wird. Diese Koeffizientenverteilung ist sehr allgemein gewählt und bei großen Problemen empfiehlt sich das Testen von verschiedenen Parametersets. Vor allem die Wahl der Mutationswahrscheinlichkeit c_m gilt es aufgrund der Mächtigkeit der Hilfsheuristik zu beachten. In Abbildung 2 sind die Abweichungen der Ergebnisse auf zwei Problemen mit verschiedenen Objektgrößen abgebildet. Problem 2 kommt bei niedriger Mutationswahrscheinlichkeit der optimalen Lösung nahe, aber Problem 1 kann sich auch bei hoher Mutationswahrscheinlichkeit über die Laufzeit dem optimalen Ergebnis schlecht nähern.

Die Anzahl an auftretenden Konflikten welche mit Mutation behandelt wird spielt sich im niedrigen einstelligen Prozentbereich ab und Variationen wie eine andere Hilfsheuristik für diese Konflikte lieft keine Verbesserungen.

Die Mutationswahrscheinlichkeit hat einen großen Einfluss auf die Laufzeit der Implementierung. Sie sorgt für die meisten zusätzlichen Rechenschritte.

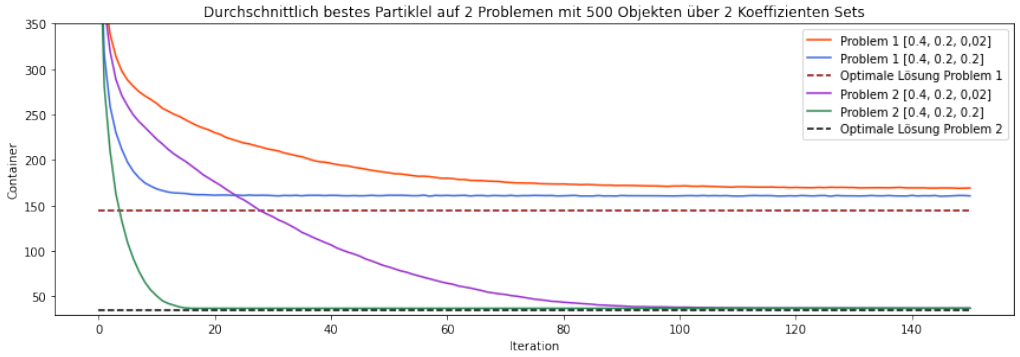

Abbildung 2: Vergleich von 2 Problemen mit 500 Objekten und unterschiedlichen Objektgrößen auf den statischen Parametersets $[40\%, 20\%, 2\%]$ und $[40\%, 20\%, 20\%]$ (Mittelwert aus 10 Durchläufen)

3.3.9 Stagnation des Schwarms

Durch die Verfahrensweise über $\vec{pbest}_i$ und $\vec{gbest}$ stagniert die Implementation oft an verschiedenen Punkten der Laufzeit und der Algorithmus kann über große Abschnitte der Laufzeit keine weiteren Verbesserungen bekommen. Die Umsetzung von Laurent und Klement, 2019 beschreibt gegen Stagnation eine Neubildung eines Teils des Schwarms vor. Test mit Implementierungen von solchen Neubildungen führten zu keinen Verbesserungen, aber sie wurden zu einem frühen Zeitpunkt in der Entwicklung des Algorithmus und nur auf wenigen Problemen mit ähnlichen Objektgrößen durchgeführt.

4 Ant Colony Optimization

Tobias

Ant Colony Optimization (ACO) ist ein Verfahren zum Lösen von kombinatorischen Optimierungsproblemen, welches sich an dem Verhalten von natürlichen Ameisen orientiert.

4.1 Inspiration

Bei Ant Colony Optimization (ACO) möchte man sich an der Art der Kommunikation von natürlichen Ameisen orientieren. Das Konzept des Informationsaustauschs welches natürliche Ameisen verwenden nennt man Stigmergie. Stigmergie ist eine indirekte Form der Kommunikation, Ameisen platzieren Pheromone in ihrer Umgebung die dann von anderen Ameisen wahrgenommen werden können. Dadurch werden Informationen wie zum Beispiel naheliegende Futterquellen oder Gefahren kommuniziert. Wenn eine Ameise eine Futterquelle findet, verteilt sie auf ihrem Weg zum Nest Pheromone mit der Information für Nahrung. Andere Ameisen nehmen diese Pheromone wahr und verteilen selbst weitere Pheromone der gleichen Information auf dem Weg, wodurch die Konzentration der Pheromone auf dem Weg steigt und so die Wahrscheinlichkeit erhöht wird, dass weitere Ameisen die Pheromone wahrnehmen. Dabei gilt, je kürzer der Weg, desto eher erreichen die Pheromone das Nest oder die Futterquelle. Daraus folgt, dass kurze Wege bei gleicher Anzahl Ameisen eine höhere Pheromonkonzentration erhalten als lange Wege. Eine höhere Pheromonkonzentration erhöht wiederum die Wahrscheinlichkeit weitere Ameisen anzulocken. Auf diese Art passen sich Ameisenstraßen an kürzere Wege an und bilden sehr effiziente Netzwerke.

Ein ACO Algorithmus möchtet diesen Effekt ausnutzen, man lässt eine Anzahl künstlicher Ameisen Lösungen für ein Optimierungsproblem suchen, je nach Qualität der gefundenen Lösung platzieren die Ameisen dann Pheromone auf ihrem Lösungsweg. Die nachfolgenden Ameisen sollen sich an den Pheromonen orientieren können und auf diese Weise noch bessere Lösungen finden.

4.2 Funktionsweise

Der erste Schritt bei der Implementation eines Ant Colony Optimization (ACO) Algorithmus ist die Konstruktion eines geeigneten Graphen für das zu lösende Problem. Die künstlichen Ameisen suchen ihre Lösungen auf dem Graphen, indem sie von Knoten zu Knoten wandern und sich dabei an den Pheromonwerten auf den jeweiligen Kanten orientieren. Der Weg den eine bestimmte Ameise auf dem Graphen zurücklegt soll einer gültigen Lösung des Problems entsprechen. Ich werde im Folgenden die Funktionsweise eines ACO Algorithmus beispielhaft an dem klassischen Traveling-Salesman-Problem(TSP) erläutern.

4.2.1 Das Traveling-Salesman-Problem (TSP)

Das klassische TSP, auch das Problem des Handlungsreisenden oder allgemeines kürzeste-Wege-Problem genannt, ist ein kombinatorisches Optimierungsproblem. Man hat eine bestimmte Anzahl von Städten gegeben, alle Städte sind durch Wege miteinander verbunden. Die Entfernungen zwischen den einzelnen Städten ist ebenfalls gegeben. Man möchte eine kürzeste Route finden bei der jede Stadt genau einmal besucht wird. Je nach Definition soll die Route in der Anfangsstadt enden. Auf die Schwierigkeit des Problems hat dies allerdings keinen Einfluss, deshalb beschränken wir uns für das Beispiel auf die Variante bei der eine Route gesucht wird die in der Anfangsstadt endet.

4.2.2 Ant Colony Optimization(ACO) Metaheuristik

1. *Set parameters, initialize pheromone trails*
2. **while** *termination condition not met* ***do****:*
3. *ConstructAntSolutions*
4. *LocalSearch* (*optional*)
5. *UpdatePheromones*
6. **endwhile**

(Dorigo, Birattari und Stutzle, 2006)
Diese ACO Metaheuristik ist ein allgemeines Schema für ACO Algorithmen an welchem die Implementation orientiert ist. Bei dem TSP ist die Wahl des Konstruktionsgraphen naheliegend, man ordnet jede Stadt einem Knoten zu und verbindet alle Knoten untereinander durch Kanten. Jeder Kante wird dann die Distanz zwischen den beiden Städten der inzidenten Knoten zugeordnet. Jede Kante erhält einen Pheromonwert.

Nach der Initialisierung werden in einer Schleife zwei wesentliche Schritte durchgeführt. In dem ersten Schritt, *ConstructAntSolutions*, wandern die Ameisen über den Lösungsgraphen und suchen gültige Lösungen. In dem zweiten Schritt wird *UpdatePheromones* ausgeführt. Die gefundenen Lösungen aus *ConstructAntSolutions* werden in diesem Schritt ausgewertet und je nach Qualität der Lösung werden entsprechend viele Pheromone auf den Kanten einer Lösung verteilt. In der folgenden Iteration sollen die Ameisen bei *ConstructAntSolutions* durch die Pheromonwerte zu besseren Lösungen geführt werden, welche

wiederum in das Update mit einfließen. Die gefundenen Lösungen in *ConstructAntSolutions* können vor dem Update der Pheromone durch eine lokale Suche verbessert werden. Für die Implementation des ACO Algorithmus für das Bin Packing Problem wurde auf eine lokale Suche verzichtet.

4.2.3 ConstructAntSolutions

Bei *ConstructAntSolutions* werden die künstlichen Ameisen über den Graphen geschickt um ihre Lösungen zu finden. Bei einer Implementierung des TSP genügt es alle Ameisen in einer beliebigen Stadt starten lassen, da alle Städte in einer gültigen Lösung besucht werden müssen. Um Lösungen zu bilden besuchen die künstlichen Ameisen schrittweise die einzelnen Knoten, ein folgender Knoten wird zufällig unter den noch unbesuchten Knoten ausgewählt. Für die Wahl des nächsten Knotens wird anhand der Heursitik und der Pheromonwerte eine Wahrscheinlichkeitsverteilung für die verfügbaren Knoten berechnet. Die einzelnen Wahrscheinlichkeiten können wie in **(4.1)** berechnet werden. (Dorigo, Birattari und Stutzle, 2006)

$$\textbf{(4.1)} \quad p_{ij}^k = \begin{cases} \frac{\tau_{ij} \cdot \eta_{ij}}{\sum_{c_{il} \in L(s^k)} \tau_{il} \cdot \eta_{il}} & \text{wenn } c_{ij} \in L(s^k) \\ 0 & \text{falls } a = 0 \end{cases}$$

Die Heuristik bestimmt die grundlegende Orientierung der Ameisen bei der Wahl des nächsten Knoten. Bei dem TSP sucht man eine möglichst kurze Route, demnach wird eine kurze Distanzen bevorzugende Heuristik gewählt. In **(4.1)** stellt η_{ij} die Heuristische Information dar und ist durch $\eta_{ij} = 1/d_{ij}$ definiert. Kurze Kanten erhalten einen höheren Wert und somit steigt auch die Wahrscheinlichkeit p_{ij}^k, dass die Kante c_{ij} von der Ameise k gewählt wird.

τ_{ij} stellt den Pheromonwert auf der Kante c_{ij} dar. Wenn die Kante c_{ij} zu einer bereits besuchten Stadt führt, gilt sie als nicht verfügbar und die Wahrscheinlichkeit p_{ij}^k für das Besuchen dieser Kante ist gleich Null. Ansonsten betrachtet man Pheromonwert τ_{ij} und Heuristische Information η_{ij} der Kante c_{ij} im Verhältnis zu der Summe der entsprechenden Werte aller Kanten in der Menge $L(s^k)$. $L(s^k)$ bezeichnet die Menge aller verfügbaren Kanten von dem zuletzt besuchten Knoten in der Teillösung s^k von Ameise k.

4.2.4 UpdatePheromones

Die im vorherigen Schritt gefundenen Lösungen werden ausgewertet und es werden je nach Qualität der Lösungen unterschiedlich viele Pheromone auf die Kanten gelegt. Eine mögliche Berechnung eines Pheromonupdates wird in **(4.2)** gezeigt. (Dorigo, Birattari und Stutzle, 2006)

$$\textbf{(4.2)} \quad \tau_{ij} \leftarrow p \cdot \tau_{ij} + \sum_{k=1}^{n} \Delta\tau_{ij}^k$$

Die Pheromonwerte τ_{ij} der vorherigen Iteration werden um einen Faktor der Zerfallsrate

p verringert. Für die in einer Iteration gefundenen Lösungen wird ein Wert $\Delta\tau_{ij}^{k}$ zum Pheromonwert addiert der $1/L_k$ beträgt, falls Ameise k die Kante $c_i j$ in ihrer Lösung gewählt hat, ansonsten wird 0 addiert. L_k bezeichnet die Länge der Route von Ameise k.

4.2.5 Weitere ACO Algorithmen

Die vorgestellte Variante von ACO ist an Ant System(AS) angelehnt. Ant System und zwei weitere ACO Varianten werden in dem Bericht von Dorigo, Birattari und Stutzle, 2006 vorgestellt. Bei der Variante *MAX-MIN* Ant System(*MM*AS) berücksichtigt man nur die beste gefundene Lösung für das Update der Pheromone. Diese Variante soll allgemein bessere Ergebnisse als ein einfacher AS Algorithmus erzielen. Für die Implementation des zweidimensionalen Bin Packing Problems wurde ein an *MAX-MIN* Ant System angelehnter Algorithmus gewählt. Ant Colony System ist eine Variante von AS bei welcher schon während der Lösungssuche der Ameisen Pheromone der besuchten Kanten verändert werden. Eine Ameise verringert den Wert auf einer besuchten Kante damit eine folgende Ameise sich mit höherer Wahrscheinlichkeit für einen anderen Weg entscheidet. So kann man mit höherer Wahrscheinlichkeit eine größere Anzahl an verschiedenen Lösungen finden.

4.3 Zweidimensionales Bin Packing Problem mit ACO

Im folgenden Abschitt wird die Struktur und Funktionsweise der Implentation eines ACO Algorithmus für das Bin Packing erklärt. Anschließend folgt eine Auswertung der Implementation.

4.3.1 Funktionsweise

Der Aufbau des Algorithmus ist an der ACO-Metaheuristik 4.2.1 orientiert. Der erste Schritt ist die Bildung des Konstruktionsgraphen für das BPP. Wenn man sich an dem bereits vorgestellten Ansatz für das TSP orientiert, dann wird ein Graph gewählt bei dem jeder Knoten eines der gegebenen Objekte darstellt. Eine künstliche Ameise bildet dann eine Lösung indem sie eine Reihenfolge der Objekte sucht in der die Objekte in den Containern platziert werden. Die Pheromone sind dann ein Maß dafür, wie oft ein bestimmtes Objekt direkt nach einem anderen Objekt in den guten Lösungen platziert wurde. Dies ist jedoch ein Ansatz der im Kontrast zur Definition des BPP steht. In jeder Lösung des BPP kann man die Container beliebig vertauschen und so eine gleichwertige Lösung erhalten obwohl die Reihenfolge der Objekte eine völlig andere ist. Die Objekte erhalten so, selbst wenn sie sich in unterschiedlichen Containern befinden, eine Abhängigkeit zueinander. Eigentlich sollen Objekte in verschiedenen Containern untereinander eigentlich unabhängig voneinander einsortiert werden. Im Gegensatz zu dem TSP wird bei dem BPP keine Reihenfolge der Knoten gesucht, sondern Gruppierungen von Knoten die in guten Lösungen besonders häufig in einem Container platziert werden. (Levine und Ducatelle, 2004)

Der Pheromonwert auf einer Kante c_{ij} gibt an, wie häufig ein Objekt mit Gewicht i in guten Lösungen zusammen mit einem Objekt mit Gewicht j in dem selben Container platziert wurde. Es kann mehrere Objekte mit gleichem Gewicht geben, alle Objekte mit Gewicht i greifen also auf die gleichen Pheromonwerte zu. Dies hat den Vorteil, dass die Pheromonwerte viel kompakter gespeichert werden können. In der Implementation sind die Pheromonwerte der einzelnen Kanten als Adjazenzmatrix gespeichert, jeweils eine Zeile und eine Spalte ist dabei einem bestimmten Gewicht zugeordnet. Für das zweite

Attribut Volumen wird analog eine zweite Pheromonmatrix verwendet. Die Größe dieser Matritzen entspricht somit jeweils der Anzahl unterschiedlicher Werte des entsprechenden Attributes unter den gegebenen Objekten zum Quadrat. Jede Kante in dem Graphen erhält einen Gewichtspheromonwert sowie einen Volumenpheromonwert. Diese Aufteilung der Pheromone in zwei Matritzen bedeutet mehr Aufwand, da im Gegensatz zum eindimensionalen BPP einige Berechnungen doppelt ausgeführt werden müssen. Jedoch ist die Aufteilung der Pheromone in Gewicht und Volumen sehr vorteilhaft gegenüber einer einzigen Matrix in der beispielsweise ein Koeffizient aus Volumen und Gewicht oder schlichtweg die Reihenfolge der Objekte als Index genutzt wird. Die Aufteilung ermöglicht viel effektivere Schritte bei der Auswahl der Objekte sowie bei der Bewertung durch eine Fitness-Funktion.

4.3.2 Initialisierung

Die Pheromonmatritzen können mithilfe der gegebenen Objekte initialisiert werden, Gewichte und Volumen sind einem entsprechenden Index zugeordnet. Dieses Mapping wird gespeichert um später dem jeweilige Gewicht oder Volumen den passenden Index zuordnen zu können. Die Einträge beider Pheromonmatritzen werden auf einen Startwert gesetzt welcher bei der Auswertung genauer besprochen wird. Ein wesentlicher Parameter ist die Anzahl der künstlichen Ameisen, er gibt an wie viele Lösungen in *ConstructAntSolutions* konstruiert werden sollen. Nach der Initialisierung wird die Funktion *ConstructAntSolutions* in der ersten Iteration ausgeführt. (s. Abschnitt 4.2.1)

4.3.3 Erstellen der Lösungen (BPP)

In diesem Schritt konstruieren die künstlichen Ameisen ihre Lösungen. Eine Ameise findet eine Lösung indem sie die Container der Reihe nach befüllt. Die Objekte werden nacheinander in den Containern platziert, ein neue Container wird erst geöffnet wenn keines der Objekte mehr hineinpasst. Die Auswahl eines Objektes wird mithilfe einer Wahrscheinlichkeitsverteilung bestimmt.

$$\textbf{(4.3)} \quad p_i^k = \begin{cases} \frac{X_{G_i}^k \cdot X_{V_i}^k}{\sum_{c_r \in L(s^k)} X_{Gr}^k \cdot X_{Vr}^k} & \text{wenn } c_i \in L(s^k) \\ 0 & \text{sonst} \end{cases}$$

$$\textbf{(4.4)} \quad X_{G_i}^k = \begin{cases} \frac{[\tau_g^i] \cdot [g_i]^b}{\sum_{c_r \in L(s^k)} [\tau_g^r] \cdot [g_r]^b} & \text{wenn } c_i \in L(s^k) \\ 0 & \text{sonst} \end{cases}$$

$$\textbf{(4.5)} \quad X_{V_i}^k = \begin{cases} \frac{[\tau_v^i] \cdot [v_i]^b}{\sum_{c_r \in L(s^k)} [\tau_v^r] \cdot [v_r]^b} & \text{wenn } c_i \in L(s^k) \\ 0 & \text{sonst} \end{cases}$$

p_i^k gibt die Wahrscheinlichkeit an, mit welcher ein Objekt i als nächstes Objekt in einem Container platziert wird. p_i^k ist gleich Null, wenn das Objekt i bereits in einem vorherigen Container von Ameise k platziert wurde oder wenn das Objekt nicht in den zu diesem Zeitpunkt geöffneten Container passt. Andernfalls erhält p_i^k einen Wert der aus zwei Wahrscheinlichkeitsverteilungen gebildet wird. Zum einen X_G für Gewicht und X_V für Volumen die jeweils aus den entsprechenden Attributen der Objekte und Pheromonmatritzen berechnet werden. X_G und X_V werden dabei jeweils auf die gleiche Art bestimmt

wie die Verteilung für das eindimensionale BPP in dem Artikel von Levine.(Levine und Ducatelle, 2004) $X^k_{G_i}$ ist die zu Objekt i zugehörige Wahrscheinlichkeit für die aktuelle Teillösung s^k von Ameise k und wird wie in (**4.4**) dargestellt berechnet. g_i stellt die heuristische Information des Gewichtsattributs dar, τ^i_g setzt sich wie in (**4.6**) beschrieben zusammen.

$$\textbf{(4.6)} \quad \tau^i_g = \begin{cases} \frac{\sum_{g_j \in C} \tau_g(i,j)}{|C|} & \text{wenn } C \neq \varnothing \\ 1 & \text{sonst} \end{cases}$$

$$\textbf{(4.7)} \quad \tau^i_v = \begin{cases} \frac{\sum_{v_j \in C} \tau_v(i,j)}{|C|} & \text{wenn } C \neq \varnothing \\ 1 & \text{sonst} \end{cases}$$

τ^i_g ist gleich der Summe aller Gewichtspheromonwerte von Objekt i zu allen Objekten die sich im aktuellen Container befinden, geteilt durch die aktuelle Anzahl von Objekten in dem Container $|C|$. τ^i_g entspricht 1 wenn der Container leer ist. $X^k_{V_i}$ und τ^i_v werden analog zu $X^k_{G_i}$ und τ^i_g in (**4.5**) und (**4.7**) beschrieben.

Aus den beiden Verteilungen X_G und X_V werden jedem Objekt zwei Wahrscheinlichkeiten zugeordnet, die zu einer gemeinsamen Wahrscheinlichkeit kombiniert werden müssen. Ein Mittelwert ist hier ungeeignet, da so die Information der einzelnen Verteilungen verloren geht. Aus einem arithmetischen Mittelwert kann man wenig Schlüsse auf die ursprünglichen Werte ziehen. Mehrere unterschiedliche Verteilungen werden auf den gleichen Wert abgebildet. Als Beispiel werden zwei Paare von Wahrscheinlichkeiten betrachtet, $Q_1 : (X^1_G, X^1_V) = (0.8, 0.2)$ und $Q_2 : (X^2_G, X^2_V) = (0.5, 0{,}5)$. Der arithmetische Mittelwert von beiden Paaren beträgt jeweils 0.5, sie stellen jedoch unterschiedliche Informationen dar. Bei Q_1 ist ein sehr guter Gewichtswert gepaart mit einem schlechten Volumenwert, Q_2 hat mäßige Werte bei beiden Attributen. Es stellt sich die Frage welches der Paare bevorzugt werden sollte. Wenn Fälle wie Q_1 mit ungleichmäßiger aber dafür mindestens einem sehr hohen Attribut bevorzugt werden, dann fallen die Pheromon und Heuristik Informationen des schwächeren Wertes weniger ins Gewicht. Das ist vermutlich ungünstig, man möchte die Container möglichst gleichmäßig im Hinblick auf beide Attribute füllen, deshalb ist es wahrscheinlich eher von Vorteil ausgeglichene Werte wie in Q_2 zu bevorzugen, so wird die Stärke der Informationen von Gewicht und Volumen möglichst gut ausbalanciert. Die Wahrscheinlichkeitwerte von X_G und X_V werden für jedes Objekt einzelnd miteinander multipliziert, Q_2 erreicht dann mit 0.25 einen höheren Wert als Q_1 mit 0.16. Die jeweiligen kombinierten Wahrscheinlichkeiten werden dann noch durch deren Gesamtsumme geteilt um eine echte Wahrscheinlichkeitsverteilung zu erhalten. Diese Methode wurde mit einer entgegengesetzten Methode verglichen, bei der durch Potenzieren der einzelnen Werte ungleichmäßige Fälle wie Q_1 bevorzugt werden. Die Vermutung, Fälle wie bei Q_2 sollten bevorzugt werde, hat sich nicht eindeutig bestätigt. Für die Tests wurde dennoch die Mulitplikationsvariante gewählt. Eine Variante bei der Fälle wie Q_1 vorgezogen werden würde in vielen Fällen hauptsächlich die Heuristik verstärken, was nicht zielführend ist. Die Stärke der Heuristik soll durch einen anderen Parameter bestimmt werden.

First Fit Decreasing (FDD) wurde als Heuristik gewählt. Die oben beschriebene Methodik zum Einsortieren der Objekte entspricht bereits der First Fit Heursistik. Die Werte g_i in (**4.4**) und v_i in (**4.5**) stellen die heuristische Information eines Objektes i für Gewicht und Volumen dar. Bei First Fit Decreasing möchte man schwere und voluminöse Objekte

bevorzugen, demnach wählt man g_i gleich dem Gewicht von Objekt i und v_i gleich dessen Volumen. Der Parameter b reguliert die Stärke der Heuristik gegenüber der Pheromoninformation.

Mit der bisher beschriebenen Implementierung hat man immer eine Chance eine optimale Lösung zu finden. Angenommen es existiert eine optimale Lösung für ein BPP Problem welche nicht durch den beschriebenen ACO-Algorithmus mit der First Fit Decreasing Heuristik gefunden werden kann. Man sortiere die Container einer optimalen Lösung absteigend nach ihrem Füllstand und erhält eine gleichwertige optimale Lösung mit einer anderen Containerreihenfolge. Wenn diese Lösung nicht auf die beschriebene Weise gefunden werden kann, dann muss es mindestens einen Container geben der noch genügend Platz für mindestens ein Objekt aus den nachfolgenden Containern hat. Dieses Objekt kann dann in den anderen Container verschoben werden um eine gültige Lösung mit gleicher Containeranzahl zu erhalten. Wenn dann noch Container übrig sind die noch genügend Platz für ein Objekt der nachfolgenden Container haben, wiederholt man den letzten Schritt so lange bis dies nicht mehr der Fall ist. Man erhält eine optimale Lösung die auch von dem beschriebenen ACO Algorithmus mit der First Fit Decreasing Heuristik gefunden werden kann. Es genügt den First Fit Aspekt zu betrachten, da der Decreasing Aspekt nur die Wahrscheinlichkeiten beeinflusst aber keine eindeutigen Lösungen erzwingt. Die Wahrscheinlichkeit dafür, dass ein passendes Objekt ausgewählt wird ist immer größer als Null. Auf diese Art kann für alle möglichen Lösungen für das BPP eine mindestens gleichwertig gute FFD Heuristik Lösung gefunden werden. Es ist einer Ameise also immer möglich eine optimale Lösung zu finden.

4.3.4 Pheromone Update (BPP)

Die Pheromonupdates orientieren sich an dem Algorithmus *MAX-MIN* Ant System.(Dorigo, Birattari und Stutzle, 2006) Bei diesem wird nur die beste Lösung für das Update berücksichtigt. Die beste Lösung einer Iteration wird mithilfe Fitnessfunktion ermittelt, die in dem nachfolgenden Abschnitt erläutert wird. Der Wert an Pheromonen, den der Eintrag $\tau_g(i,j)$ der Gewichtspheromonmatrix erhält, wird in Formel **(4.8)** bestimmt.

$$\textbf{(4.8)} \quad \tau_g(i,j) = p \cdot \tau_g(i,j) + m.f_g(s^{update})$$

$$\textbf{(4.9)} \quad \tau_v(i,j) = p \cdot \tau_v(i,j) + m.f_v(s^{update})$$

Die Pheromonwerte der vorherigen Iteration werden zunächst um den Faktor p, der Zerfallsrate, verringert. Dann wird der Wert $m.f(s_g^{update})$ auf die Matrix addiert. Es wird für jedes Mal, wenn sich bei der Lösung s^{update} ein Objekt mit Gewicht i zusammen mit einem Objekt mit Gewicht j in dem selben Container befindet, jeweils eine 1 auf die Einträge $\tau_g(i,j)$ und $\tau_g(j,i)$ der Gewichtspheromonmatrix addiert. $\tau_v(i,j)$ wird analog zu $\tau_g(j,i)$ für die Volumenpheromone gebildet, wie in **(4.9)** beschrieben. (Levine und Ducatelle, 2004) In *ConstructAntSolutions* werden die Inhalte der Container als Lösungsparameter übergeben. Für das Update wird eine Matrix C_g verwendet deren Zeilen die Anzahlen der unterschiedlichen Objektgewichte eines Containers angeben. Durch Matrixmultiplikation von C_g mit seiner transponierten C_g.T erhält man eine co-occurence Matrix C_g^{occur}. Die Matrix C_g^{occur} enthält die Information wie oft ein Objekt mit Gewicht i zusammen mit einem Objekt mit Gewicht j in einem Container der Lösung s^{update} vorkommt. Man muss aber noch die jeweiligen Gewichte i von den zugehörigen Diagonaleinträgen $C_g^{occur}[i,i]$

abziehen um die gewünschte Matrix zu erhalten, da sonst jedes Objekt mit sich selbst auch eine Paarung in der besten Lösung darstellt. Analog wird eine Volumenmatrix C_v^{occur} berechnet.

Da nur die beste Lösung verwendet wird, ist die Suche sehr aggressiv. Das bedeutet der untersuchte Bereich des Suchraumes wird von Iteration zu Iteration sehr schnell kleiner. Im Kontext der Heuristik bedeutet ein kleinerer untersuchter Bereich des Suchraums lediglich, dass die Wahrscheinlichkeit eine Lösung in diesem Bereich zu erhalten deutlich größer wird. Es gibt keine strikte Abgrenzung zum Rest des gesamten Suchraumes. Wenn die optimale Lösung nicht in diesem Bereich liegt ist sie dennoch theoretisch erreichbar, wie in Abschnitt 4.3.3 gezeigt. Die Wahrscheinlichkeit diese zu finden kann jedoch so klein werden, dass man sie als praktisch unerreichbar bezeichnen könnte. Demnach sollte der untersuchte Bereich des Suchraumes nicht zu schnell kleiner werden. Dafür kann man einen Mindestwert t_{min} für Pheromoneinträge setzen, wie bei *MAX-MIN* Ant System.(Dorigo, Birattari und Stutzle, 2006) Ein Maximalwert ist für das BPP eher ungeeignet, da bestimmte Gewichte und Volumen unterschiedlich oft vorkommen können wachsen die Einträge der Pheromonmatritzen auch zwingend unterschiedlich schnell.

Eine weitere Möglichkeit um den Suchraum möglichst groß zu halten ist die Einführung eines Parameters μ. Für das Update der Pheromone kann man entweder den lokalen Bestwert der aktuellen Iteration oder den besten globalen Bestwert aller bisherigen Iterationen verwenden. Durch das Verwenden des lokalen Bestwerts kann der Suchraum größer gehalten werden, wenn jedoch nach μ Iterationen kein Wert der besser ist als der globale Bestwert, dann wird stattdessen der globale Bestwert für das Update genutzt bis ein neuer globaler Bestwert gefunden wird.(Levine und Ducatelle, 2004)

4.3.5 Fitnessfunktion

Für das Update der Pheromone sollte möglichst eine Lösung verwendet werden die Verbesserungspotential hat. Wenn man lediglich eine Lösung mit möglichst geringer Containerzahl verwendet ist dies oft nicht der Fall, da es meist sehr viele verschiedene Lösungen gibt die um einen Container schlechter als die optimale Lösung sind, jedoch nicht unbedingt in der Nähe einer optimalen Lösung liegen. Ein Update mit solchen Lösungen würde den betrachteten Bereich des Suchraum von der Optimalen Lösung ablenken. Um dies zu vermeiden wird die in (**4.10**) angegebene Fitnessfunktion verwendet.(Levine und Ducatelle, 2004)

$$\textbf{(4.10)} \quad Fitness(s) = \frac{[Fitness_g(s)]^k + [Fitness_v(s)]^k}{2}$$

$$\textbf{(4.11)} \quad Fitness_g(s) = \frac{\sum_{i=1}^{N}(F_g^i/C_g)^k}{N}$$

$$\textbf{(4.12)} \quad Fitness_v(s) = \frac{\sum_{i=1}^{N}(F_v^i/C_v)^k}{N}$$

Für Gewicht und Volumen wird jeweils eine eigene Fitness berechnet, wie in (**4.11**) und (**4.12**) beschrieben. Die Füllstände werden mit dem Parameter k potenziert. Für $k = 2$ werden dann Lösungen mit mehreren gut gefüllten und wenig gefüllten Containern besser bewertet als Lösungen mit gleichmäßig befüllten Containern. Solche Lösungen haben

ein größeres Verbesserungspotential. Für Werte von k größer 2 kann es allerdings vorkommen, dass eine suboptimale Lösung einen besseren Fitnesswert erhält als die optimale Lösung(Falkenauer, 1996) Dieser Fitness-Wert wird für jede Ameise getrennt jeweils für Gewicht und Volumen berechnet und dann wie in (**4.10**) beschrieben zusammengebracht. Hier hat man wie bei dem Zusammenführen der Wahrscheinlichkeitsverteilungen in Abschnitt 4.3.3 verschiedene Möglichkeiten. Allgemeine Mittelwerte sind auch hier ungeeignet. Ich habe zwei vielversprechende Ansätze ähnlich zu denen in Abschnitt 4.3.3 verglichen, bei denen jeweils entweder gleichmäßige Werte oder ungleichmäßige mit hohen Einzelwerten bevorzugt werden. Bei der Fitnessfunktion hat es sich, analog zu der Methodik bei den einzelnen Fitnessfunktionen der Attribute in (**4.11**) und (**4.12**), bewährt, hohe Einzelwerte besser zu gewichten. Daher werden die beiden Fitnesswerte in (**4.10**) einzelnd mit dem Parameter k potenziert und dann zusammengeführt.

4.4 Auswertung

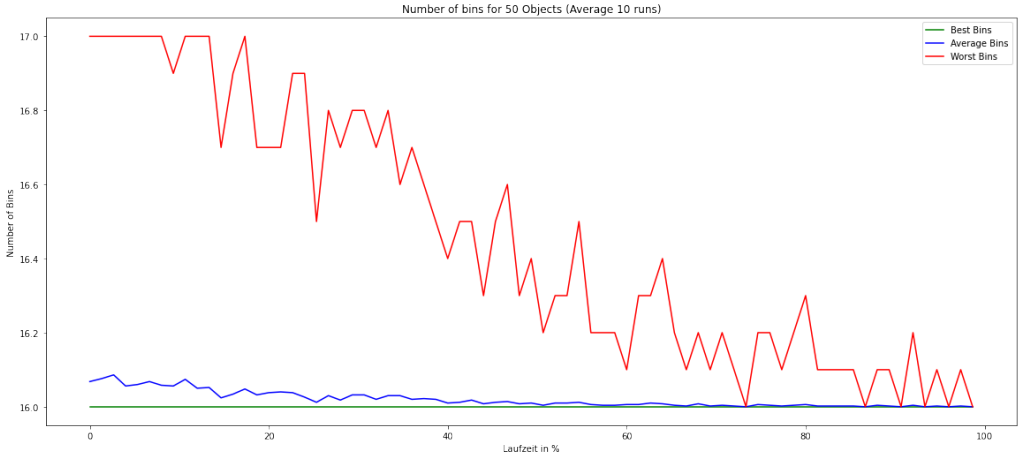

Abbildung 3: Ein Vergleich der besten, durchschnittlichen und schlechtesten Containeranzahl über die Laufzeit von ACO

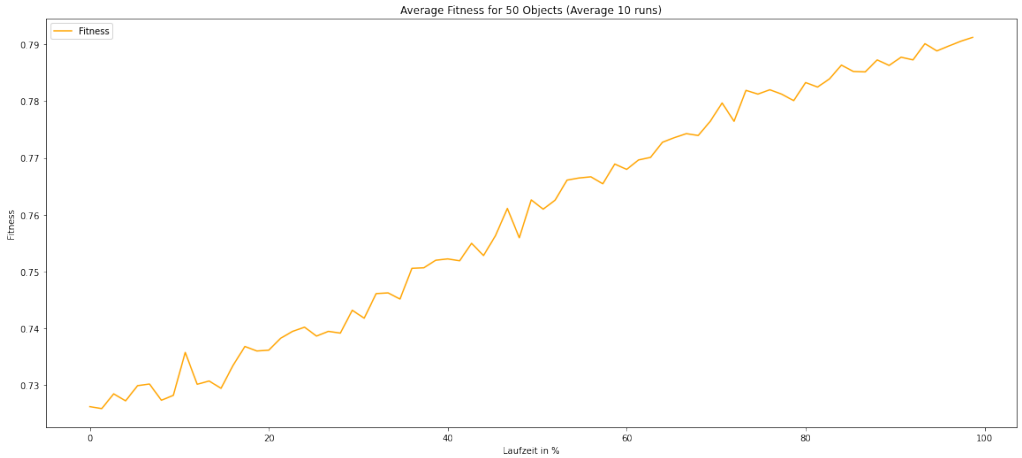

Abbildung 4: Verlauf der Fitness auf einem Problem mit ACO

In Abbildung 3 erkennt man sehr gut wie sich die durchschnittliche Containerzahl und

die Containeranzahl der schlechtesten Lösungen über die Laufzeit dem globalen Bestwert annähert. Der Graph für den besten Wert pro iteration verläuft konstant da der finale Bestwert bereits in der ersten Iteration gefunden wurde. In Abbildung 4 ist der durchschnittliche Fitnesswert für das gleiche Problem dargestellt. Es ist eine Verbesserung über die gesamte Laufzeit zu erkennen.

4.4.1 Laufzeit

Eine ausführliche Analyse der Laufzeit ist aufgrund der Komplexität der Implementation sehr schwierig. Es wird daher nur eine grobe Abschätzung aufgrund der beschriebenen Funktionsweise gezeigt. Die Laufzeit von *ConstructAntSolutions* lässt sich grob wie in (**4.13**) beschrieben abschätzen.

$$\mathcal{O}(ConstructAntSolutions) = \mathcal{O}(ants \cdot N \cdot Z) \tag{4.13}$$

N entspricht der Anzahl an gegebener Objekte. Für die Zuweisung eines Objektes wird Komplexität $\mathcal{O}(Z)$ benötigt, Z wird N-mal durchgeführt da jedes Objekt einem Container zugewiesen wird. Diese Prozedur wird für jede Ameise durchgeführt. Bei größeren Objektmengen werden mehr Ameisen zu dem Finden guter Lösungen benötigt, wenn $\mathcal{O}(Z)$ besonders aufwendig ist wirkt sich das stark auf die Laufzeit aus. Z beinhaltet unter anderem die Berechnung der Verfügbarkeit und der Wahrscheinlichkeitsverteilungen der Objekte, dabei müssen für jede Zuweisung die Gewicht und Volumenwerte berechnet werden, was sich bei der Laufzeit bemerkbar macht. Die Komplexität von *UpdatePheromones* ist deutlich günstiger. Das Update kann durch Bilden von co-occurence Matritzen schnell berechnet werden. Insgesamt ist eine hohe Laufzeit zu Erwarten.

4.4.2 Parameterwahl

In einem Bericht von Levine und Ducatelle, 2004 wird eine Anzahl Ameisen gleich der Anzahl gegebener Objekte empfohlen, bei dieser Implementierung wurden aber auch mit deutlich weniger Ameisen bereits sehr gute Lösungen gefunden. Aufgrund der hohen Laufzeit wurden deswegen die Tests der größeren Probleme mit 250 und 500 Objekt mit relativ wenigen Ameisen durchgeführt, circa 25 bis 30 Prozent der Anzahl Objekte.

Die Anzahl Iterationen sind ebenfalls aufgrund der hohen Komplexität für die Tests eher klein gewählt worden, meist etwa 10 bis 20 Iterationen weniger als die Anzahl Ameisen. So wurde der Algorithmus zwar häufig beendet wenn noch eine Verbesserung möglich war, aber dennoch eine gute Verbesserung über die Laufzeit erzielt.

Der Parameter b ist ein Maß für das relative Gewicht der Heuristik gegenüber den Pheromonen. Hohe Werte von b erzielten eine bessere initiale Lösung, niedrige Werte bewirkten eine gleichmäßigere und stärkere Verbesserung über die Laufzeit. Es haben sich Werte für b von 2 bis 7 als geeignet erwiesen. Größere Werte für b haben auf größeren Problemen besser abgeschnitten, dies könnte aber auch an der laufzeitbedingten geringen Anzahl Ameisen und Iterationen liegen.

Der Parameter k bestimmt die Gewichtung der Fitnessheuristik. Ein Wert von k gleich 2 hat gute Ergebnisse erzielt.

Der Startwert der Pheromonmatritzen und den Pheromonmindestwert $t_{\min}$ wurden dem Bericht von Levine und Ducatelle, 2004 entnommen. Abweichende Werte haben schlechtere Ergebnisse erzielt.

Der Parameter μ für die Anzahl Iterationen nach denen wieder die globale beste Lösung für das Update verwenden sollte, wurde für kleine Probleme mit Werten zwischen

5 bis 10 gewählt, für große Probleme mit Werten zwischen 1 bis 2. Diese Wahl ist jedoch eher von der geringen Iterationszahl abhängig als von der Lösungsqualität.

Für die Zerfallsrate p hat ein Wert von 0.9 bis 0.95 auf den meisten Problemen gut funktioniert.

5 Genetischer Algorithmus

Tim

In den folgenden Abschnitten werden, anhand eines Basisalgorithmus, kurz die theoretischen Grundlagen genetischer Algorithmen beschrieben. Anschließend betrachten wir die Implementierung eines genetischen Algorithmus zur Lösung des zweidimensionalen Bin Packing Problems.

5.1 Vorstellung des Basisalgorithmus

Viele Probleme in der realen Welt besitzen einen großen Lösungsraum. Die Nachfrage nach Algorithmen, die diese großen Lösungsräume möglichst effizient durchsuchen, ist hoch. Dabei orientieren sich genetische Algorithmen an der biologischen Evolution, um eine effiziente Suchstrategie zu entwickeln. In der Biologie ist der Lösungsraum durch alle möglichen genetischen Sequenzen gegeben und die Lösungen sind in diesem Fall die am besten angepassten Organismen. Die biologische Evolution kann somit als eine Strategie aufgefasst werden, um diese Lösungen zu designen. (Mitchell, 1998)

5.1.1 Repräsentation der genetischen Konzepte

Genetische Algorithmen bilden biologische Konzepte ab und übernehmen daher viele biologische Begriffe. Wir betrachten an dieser Stelle die Funktion der biologischen Konzepte nicht genauer, sondern fokussieren uns auf die für die folgende Ausarbeitung wichtigsten Begriffe.

Um ein Problem zu lösen kodiert der genetische Algorithmus Lösungskandidaten. Diese bezeichnen wir als Chromosome und werden im genetischen Algorithmus normalerweise als Bit Strings abgebildet. Chromosome können wiederum in Gene unterteilt werden. Im Algorithmus entsprechen diese Blöcken von Bits innerhalb eines Chromosoms und beschreiben bestimmte Elemente einer Lösung. Mehrere Chromosome zusammen bilden eine Population. Durch die Rekombination von Elternchromosomen entstehen Nachkommen. Diese werden durch eine zufällige Mutation verändert und ersetzen anschließend die alte Population. Es entsteht eine neue Generation. (Mitchell, 1998)

5.1.2 Fitnesslandschaften

Ziel des genetischen Algorithmus ist es, durch die parallele Anwendung einfacher Regeln immer bessere Chromosome zu erstellen. Jedes Chromosom beschreibt die Kodierung einer Lösung eines gegebenen Problems. Wir möchten jedem Chromosom eine Bewertung hinsichtlich seiner Eignung zur Lösung des vorliegenden Problems zuordnen. Dies geschieht durch eine Fitnessfunktion. Je höher der Wert der Fitnessfunktion eines Chromosoms ist, desto besser sollte dieses zur Lösung des Problems geeignet sein. Werden die Lösungen durch l-dimensionale Chromosome kodiert, so entsteht zusammen mit der Fitnessfunktion eine $l + 1$-dimensionale Fitnesslandschaft.

Durch die Beschreibung der Fitnesslandschaft kann die Funktionsweise des genetischen Algorithmus auch anders aufgefasst werden. Der Algorithmus lässt eine Populati-

on die Fitnesslandschaft erkunden und bewegt die Population Generation um Generation hin zu lokalen Optima. (Burke u. a., 2014)

5.1.3 Komponenten genetischer Algorithmen

Nachdem die grundlegenden Begriffe vorgestellt wurden, lässt sich ein einfacher Basisalgorithmus vorstellen. Genetische Algorithmen folgen dieser Strukture und passen die einzelnen Komponenten entsprechend an. In Burke u. a., 2014 wird der folgende Algorithmus und die einzelnen Komponenten beschrieben:

1. Initialisierung
2. Evaluation
3. Selektion
4. Rekombination
5. Mutation
6. Ersetzen
7. Wiederhole Schritte 2-6 bis ein Stopp Kriterium erreicht ist.

Die Initialisierung erzeugt eine initiale Population von Lösungskandidaten. Diese bildet den Startpunkt des Algorithmus. Die Initialisierung erfolgt normalerweise zufällig, in manchen Fällen kann domänenspezifisches Vorwissen genutzt werden. Die Evaluation bewertet jedes Chromosom hinsichtlich seiner Eignung zur Lösung des Problems, dafür wird die vorher definierte Fitnessfunktion ausgewertet.

Die Selektion wählt nun eine bestimmte Anzahl an Kandidaten aus der gegebenen Population. Diese Auswahl hängt von der Fitness der Kandidaten ab und kann auf verschiedene Weisen erfolgen. Die Selektion überträgt das Konzept *Survival of the fittest*, da in den verschiedenen Selektionsmethoden Chromosome mit höherer Fitness bevorzugt werden. Diese Methode soll, in Kombination mit den anderen Operatoren, die durchschnittliche Fitness im Verlauf des Algorithmus steigern. Eine Kategorie von Selektionsmethoden ist die Fitness Proportionate Selection. Bei diesen Methoden steigt die Wahrscheinlichkeit, dass ein Chromosom ausgewählt wird proportional mit der Fitness des Chromosoms.

Die in der Selektionsmethode ausgewählten Chromosomen, die Eltern, werden in der Rekombination zu neuen Chromosomen, den Nachkommen, kombiniert. Dabei sollen Bausteine guter Lösungen der Eltern auf die Nachkommen übertragen werden. Die sogenannte Crossover Probability entscheidet, ob die Eltern überhaupt rekombiniert werden. Falls diese nicht eintritt wird eine Kopie der Eltern als Nachkommen gewählt. Auch wenn generische Rekombinationsoperatoren existieren wird die Rekombination meist problemspezifisch angepasst.

Die Mutation verändert ein einzelnes Chromosom. Dabei werden meist zufällig bestimmte Teile des Chromosoms verändert. Diese zufällige Veränderung soll sicher stellen, dass andere Teile des Lösungsraums erkundet werden. Der Mutationsoperator ist problemspezifisch, beinhaltet aber häufig das durchlaufen eines Chromosoms, wobei einzelne Stellen beim eintreten einer Mutation Probability verändert werden.

Im nächsten Schritt wird die alte Population mit den erzeugten Nachkommen ersetzt. Hier gibt es verschiedene Strategien. Diese reichen von einfachen Methoden, bei denen alle Mitglieder der Population ersetzt werden, bis hin zu Methoden, bei denen sicher gestellt wird, dass eine bestimmte Elite mit Sicherheit in der nächsten Generation vertreten ist.

5.2 Umsetzung des Algorithmus

Im folgenden betrachten wir die einzelnen Bausteine erneut und passen diese zur Lösung des Bin Packing Problems entsprechend an. Dazu muss zunächst eine Kodierung der Lösungskandidaten gewählt werden.

5.2.1 Encoden der Lösung

Das Bin Packing Problem gehört zu der Gruppe der Grouping Problems. Grouping Problems sind Probleme, bei denen Elemente einer Menge in Teilmengen partitioniert werden sollen um eine Kostenfunktion zu optimieren. Gleichzeitig müssen bestimmte Nebenbedingungen eingehalten werden. (Falkenauer, Delchambre u. a., 1992)

Falkenauer et al. zeigen für verschiedene Encodings, dass diese nicht für Grouping Problems geeignet sind. Die beschriebenen Encodings orientieren sich zu stark an den Objekten und nicht an den Gruppen. Aus diesem Grund schlagen die Autoren ein Encoding vor, das aus einem Objektpart und einem Gruppenpart besteht. Das folgende Beispiel veranschaulicht das Encoding.

ABCDEEDE:ACBDE

Der erste Part (Objektpart) gibt die Containerzugehörigkeit der jeweiligen Objekte an. Das erste Objekt ist in Container A, das zweite Objekt in Container B, das dritte in C und so weiter. Der zweite Part (Gruppenpart) gibt an, welche Container verwendet werden. In diesem Fall werden die Container A,B,C,D und E verwendet. Dabei werden Rekombination und Mutation nur auf dem zweiten Teil operieren. (Falkenauer, Delchambre u. a., 1992)

Wir übernehmen diese Idee und passen die Form für die Implementierung leicht an. Jedes Chromosom wird durch eine einzelne Liste repräsentiert, die aus einer variablen Anzahl an Containern besteht. Die Container enthalten wiederum eine Liste in der die im Container vorhandenen Objekte gespeichert werden.

5.2.2 Initialisierung

Zu Beginn des Algorithmus muss eine initiale Population erzeugt werden. Der Algorithmus kann nur mit Chromosomen arbeiten, die valide Lösungsvorschläge kodieren. Somit muss eine Anfangsverteilung der Objekte auf die Container gefunden werden, die die Gewichts- und Volumenbeschränkungen einhalten.

Wir verwenden eine Kombination der First-Fit Heuristik und dem zufälligen erstellen neuer Container. Zunächst wird die Reihenfolge der Objekte gemischt. Für jedes Objekt wird mit einer Wahrscheinlichkeit von 60% ein neuer Container erstellt. Andernfalls wird das Objekt in den ersten Container eingefügt, der noch genug Platz besitzt. Falls alle Container voll sind wird, auch in diesem Fall, ein neuer leerer Container erstellt.

Die beschriebene Methode erstellt valide Lösungskandidaten. Durch die alleinige Verwendung von First-Fit, kann die Anzahl der Container in der initialen Population verringert werden. Wir verwenden diese Methode nicht, um die Lösungskandidaten zu Beginn nicht zu stark einzuschränken.

5.2.3 Fitness Funktion und Selektion

Die Fitness Funktion bewertet ein Chromosom hinsichtlich seiner Eignung zur Lösung des Problems. Eine naive Fitness Funktion ergibt sich direkt aus der Anzahl der verwendeten Container. Sei x ein Chromosom und N die Anzahl der verwendeten Container, in

der durch das Chromosom kodierten Lösung, so beschreiben wir die Fitness durch

$$f_{\text{amount bins}}(x) = N.$$

Die Fitness Landschaft besteht für das gegebene Problem möglicherweise nur aus einzelnen Spitzen und vielen flachen Ebenen, da es viele einfache Lösungen gibt, jedoch möglicherweise nur wenige optimale Lösungen. In diesem Fall fällt es dem Algorithmus schwer die Fitness Funktion zu nutzen. Eine Lösung für das Bin Packing Problem besteht darin die einzelnen Füllstände der Container in der Fitness Funktion zu berücksichtigen. Um eine bessere Verteilung für zukünftige Objekte zu ermöglichen sollten einzelne Container möglichst wenig verbleibendes Restvolumen und Restgewicht besitzen. Das führt dazu, dass zwei Container die ähnliche Füllstände besitzen schlechter bewertet werden sollen, als zwei Container bei denen einer nahezu voll ist und der andere viel freien Platz besitzt. (Falkenauer, Delchambre u. a., 1992)

Falkenauer, Delchambre u. a., 1992 beschreiben eine solche Fitness Funktion für den eindimensionalen Fall. Wir erweitern diese Fitness Funktion auf den zweidimensionalen Fall,

$$f_{\text{fill}} = \frac{\sum_{i=1}^{N} \left(\frac{fill_{vol_i}}{C_{vol_i}}\right)^2 + \left(\frac{fill_{weight_i}}{C_{weight_i}}\right)^2}{N},$$

wobei N wieder die Anzahl der verwendeten Container, $fill_{vol_i}$ bzw. $fill_{weight_i}$ den Füllstand bezüglich des Volumens/Gewichts im i-ten Container und C_{vol_i} bzw. C_{vol_i} die Kapazität des Containers bezüglich des Volumens/Gewichts beschreiben.

Um die einzelnen Fitnessfunktionen zu vergleichen führen wir eine weitere Fitnessfunktion ein. Diese liefert immer einen konstanten Wert. Die Wahl widerspricht damit den Anforderungen an eine Fitnessfunktion, da bessere Lösungskandidaten keinen höheren Fitnesswert erhalten. Wir verwenden die Fitnessfunktion nur für den Vergleich und werden diese im späteren Algorithmus nicht benutzen. Die Fitnessfunktion ist definiert durch

$$f_{constant} = 1.$$

Der Vergleich der verschiedenen Fitnessfunktionen zeigt vor allem für das kleine Problem eine bessere Performance der f_{fill} Funktion. Wie in Abbildung 5 zu sehen, verbessert die f_{fill} Funktion die durschnittlichen Werte, bis das Optimum erreicht ist, während die anderen beiden Funktionen eine durchschnittliche Anzahl von 4 Containern halten. Es ist anzumerken, dass trotzdem alle Fitnessfunktionen die optimale Lösung finden.

Für das mittlere und große Problem findet die f_{fill} Funktion früher bessere Lösungen als die beiden anderen Funktionen. Auch hier finden alle Funktionen die gleiche beste Anzahl an Containern. Je nach Problem hat die Fitnessfunktion kaum einen Einfluß auf den Verlauf des Algorithmus, wie die Auswertung auf einem neuen Problem mit 500 Objekten zeigt. Wie in Abbildung 6 zu sehen, unterscheiden sich die Fitnessfunktionen f_{fill} und $f_{amountbins}$ in ihrem Einfluss auf die Anzahl der verwendeten Container kaum. Auffällig ist außerdem, dass die $f_{constant}$ Fitnessfunktion zu nahezu gleichen Ergebnissen führt.

Eine mögliche Erklärung für dieses Verhalten kann die Wahl des Rekombinations- und Mutationsoperators sein. Da beide Operatoren stark an das Problem angepasst sind, verbessern diese in den meisten Fällen die Lösung und können diese nur in wenigen Fällen verschlechtern. Außerdem benutzen beide Operatoren die First Fit Decreasing Heuristik. Wie in Abschnitt 6.5 beschrieben, liefert diese Heuristik bei vielen Problemen bereits nahezu optimale Lösungen. Dementsprechend hat die Wahl der Eltern keinen großen Einfluss auf die Lösung, da diese sich unabhängig von der Wahl verbessert.

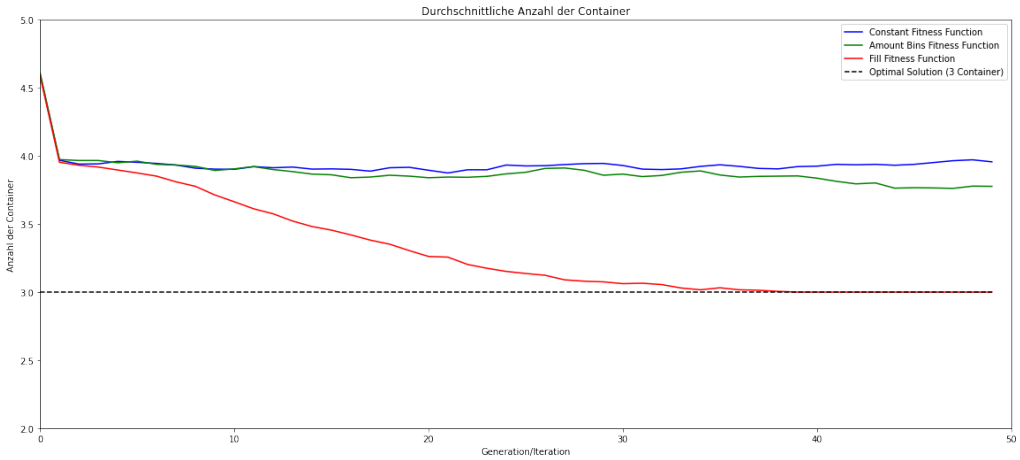

Abbildung 5: Einfluss der verschiedenen Fitnessfunktionen auf die Anzahl der durchschnittlichen Container für ein Problem mit 10 Objekten. Die Ergebnisse wurden über 10 Durchläufe gemittelt.

Alle unsere Tests zeigen entweder einen positiven Einfluß oder eine gleiche Performance der f_{fill} Funktion im Vergleich zu den anderen Fitnessfunktionen. Aus diesem Grund und den oben beschriebenen theoretischen Vorüberlegungen entscheiden wir uns in unserem Algorithmus für die f_{fill} Fitnessfunktion.

Als Selektionsmethode wird eine Fitness Proportionate Selection, genauer das Roulette Wheel Sampling, verwendet. Den Fitnesswert des i-ten Chromosoms beschreiben wir mit f_i. Wir ziehen aus allen Mitgliedern der Population N Mal mit zurücklegen. Die Wahrscheinlichkeit p_i, dass das i-te Chromosom gezogen wird, errechnet sich dabei als Anteil der Fitness an der Gesamtfitness der Population

$$p_i = \frac{f_i}{\sum_{i=1}^{N} f_i}.$$

5.2.4 Rekombination

Für die Rekombination benutzen wir den sogenannten Bin Packing Crossover (BPCX) aus Falkenauer, Delchambre u. a., 1992. Der BPCX kombiniert zwei Chromosome und produziert einen Nachkommen. Dabei sollen relevante Informationen aus beiden Elternteilen vererbt werden.

Beschreibung des BPCX: Die genaue Funktionsweise lässt sich am besten anhand des folgenden Beispiels darstellen. Seien die Gruppenparts der Elternchromosome ElterA und ElterB gegeben.

ElterA: ABCDEFG
ElterB: abcde

Zunächst wird ein zusammenhängender Teil der Container aus ElterB ausgewählt. Dazu werden im Gruppenpart des Chromosoms zwei Positionen bestimmt und alle Container zwischen diesen beiden Positionen kopiert. Die zwei Positionen werden gleichverteilt und ohne zurücklegen aus der Menge aller möglichen Positionen gezogen. In unserem Beispiel

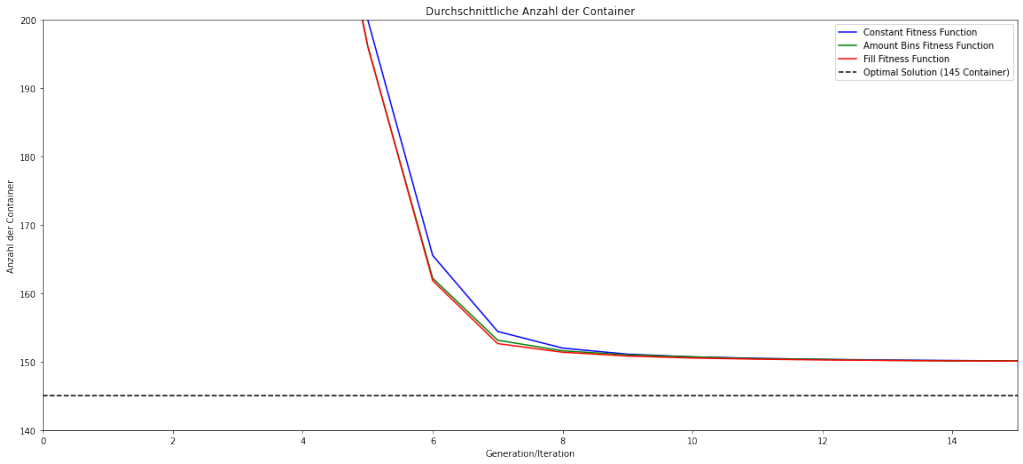

Abbildung 6: Einfluss der verschiedenen Fitnessfunktionen auf die Anzahl der durchschnittlichen Container für ein Problem mit 500 Objekten. Die Ergebnisse wurden über 10 Durchläufe gemittelt.

werden die Container b und c aus ElterB ausgewählt:

ElterA: ABCDEFG
ElterB: a|bc|de
.

Die aus ElterB ausgewählten Container sollen nun in ElterA eingefügt werden. Um den Nachkomme zu erstellen und ElterA nicht zu verändern wird das Chromosom dupliziert. Durch das Einfügen der Container in ElterA könnten nun Objekte doppelt vorkommen. Da der Rekombinationsoperator nur gültige Lösungen erstellen soll, müssen die doppelten Objekte zusätzlich betrachtet werden. Dazu wird vor dem Einfügen der Objekte jeder Container aus ElterA auf doppelte Objekte überprüft. Enthält ein Container ein Objekt, das sich bereits in einem der Container befindet, die neu eingefügt werden sollen, wird der gesamte Container gelöscht. Die verbleibenden Objekte des gelöschten Containers werden gespeichert, um diese am Ende über die Container zu verteilen. Angenommen die Objekte die in den Containern b und d enthalten sind kommen auch in den Containern B, D und E vor. Diese werden entsprechend gelöscht.

Nachkomme: ACFG

Wurden alle Container mit doppelten Objekten entfernt, wird eine Position bestimmt, an dem die Container aus ElterB eingefügt werden. Dieser wird gleichverteilt aus der Menge aller möglichen Positionen gezogen.

Nachkomme: A|CFG

Im letzten Schritt müssen die verbliebenden Objekte wieder über Bins verteilt werden. Dazu verwenden wir eine angepasste Version der First Fit Decreasing Heuristik aus Falkenauer, Delchambre u. a., 1992. Die Objekte werden absteigend nach ihrer Summe aus Gewicht und Volumen sortiert und dann nacheinander in den ersten passenden Bin eingefügt. Eine genaue Beschreibung dieser und weiterer Heuristiken befindet sich im Abschnitt „Vergleich der Heuristiken“. In unserem Beispiel erhalten wir das neue Chromosom mit dem folgenden Gruppenpart:

Nachkomme: AbcCFGx.

Dabei können allen Containern neue Objekte zugewiesen worden sein. Insbesondere beschreibt x ein oder mehrere Container, die durch First Fit Decreasing entstanden sind.

Funktionsweise des BPCX: Wir sehen, dass der Nachkomme Informationen aus den Eltern übernimmt. Es können ganze Container und möglicherweise auch mehrere gut gefüllte Container an den Nachkomme vererbt werden. Aufgrund der Kodierung der Container und der darin enthaltenen Objekte, können gut gefüllte Container ebenso vererbt werden, da diese nicht mehr direkt von der Länge innerhalb des Chromosoms abhängen. Der Rekombinationsoperator arbeitet somit nicht entgegen der zu optimierenden Fitnessfunktion. (Falkenauer, Delchambre u. a., 1992)

Vergleich der Heuristiken: Falkenauer, Delchambre u. a., 1992 beschreiben den BPCX nur für das eindimensionale Bin Packing Problem. Die Grundstruktur des Operators kann für den zweidimensionalen Fall übernommen werden. Um die Objekte im letzten Schritt ein zu sortieren schlagen die Autoren die First Fit Decreasing Heuristik vor. Die Objekte werden absteigend nach ihrer Größe sortiert und anschließend in den ersten Container eingefügt, der genügend freie Kapazität besitzt. Die Sortierung ist im zweidimensionalen Fall nicht eindeutig und es ergeben sich verschiedene Möglichkeiten. Wir vergleichen die vier Heuristiken Random Fit, First Fit No Sort, First Fit Decreasing Combined und First Fit Decreasing Chance.

Random Fit wählt für jedes Objekt einen zufälligen Container und prüft, ob dieser über genügend Platz verfügt um das Objekt auf zu nehmen. Dabei werden maximal $0.01 \cdot$ Anzahl der Objekte $+ 1$ Container geprüft bevor ein neuer Container eröffnet wird. First Fit No Sort, sortiert die Objekte in den ersten passenden Container. Wenn das Objekt in keinen der Container passt, wird ein neuer Container erstellt und das Objekt in diesen eingefügt. First Fit Combined sortiert die Objekte absteigend nach ihrer Summe aus Gewicht und Volumen und fügt diese anschließend mit First Fit ein. First Fit Chance wählt zu Beginn zufällig ob die Objekte nach Gewicht oder nach Volumen sortiert werden sollen und fügt diese mit First Fit ein.

Der Vergleich der Heuristiken zeigt, dass Random Fit am schlechtesten performt. Wobei im Test nur der oben beschriebene feste Parameter für die Anzahl der Versuche getestet wurde. Eine höhere Anzahl an Versuchen könnte diesen Wert verbessern. Wir wählen für unseren Algorithmus First Fit Combined, da diese Heuristik in den meisten Tests besser bzw. gleich gut wie First Fit Chance und First Fit No Sort abschließt.

Ein ausgewähltes Problem mit einer Darstellung der verschiedenen Heuristiken befindet sich in Abbildung 7. Man sieht vor allem, dass die Anzahl der besten Lösung bei Random Fit deutlich höher liegt, als bei den anderen Heuristiken. Das Problem zeigt einen sehr deutlichen Vorteil der First Fit Combined Heuristik, bei den anderen Tests fällt dieser Unterschied geringer aus. Weiterhin ist zu beachten, dass die Heuristik nicht nur in der Rekombination, sondern auch bei der im nächsten Abschnitt beschriebenen Mutation verwendet wird.

5.2.5 Mutation

Die durch die Rekombination erstellten Nachkommen werden durch die Mutation verändert. Wir orientieren uns an dem in Falkenauer, Delchambre u. a., 1992 vorgestellten Mutationsoperator. Die Autoren beschreiben, dass mindestens 3 Container gelöscht werden müssen, um die Fitness der Chromosome zu verbessern.

Wir löschen in jeder Mutation mindestens 3 Container, dabei befindet sich der am wenigsten gefüllte Container immer unter den gelöschten Containern. Um die Füllstände der Container zu bewerten, benutzen wir wieder die Summe aus Gewicht und Volumen der in dem Container enthaltenen Objekten.

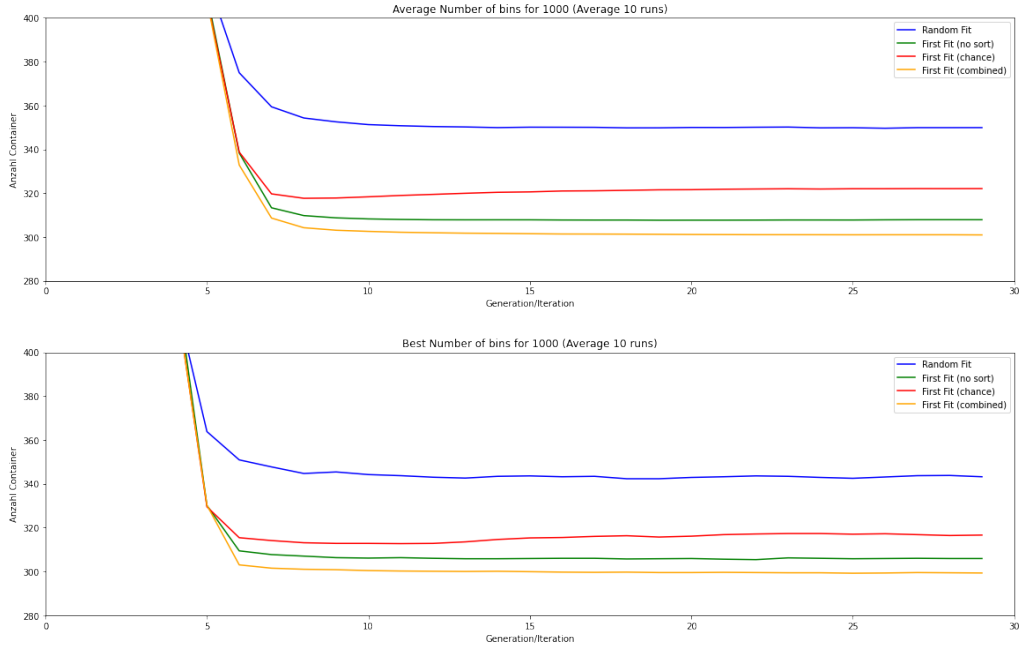

Abbildung 7: Einfluss der Heuristiken auf die Anzahl der durchschnittlichen und die Anzahl der besten Container für ein Problem mit 1000 Objekten.

Anschließend wird jeder der verbleibenden Container mit einer Mutationswahrscheinlichkeit von 0.1 gelöscht. Alle so gelöschten Objekte werden anschließend mit der First Fit Combined Heuristik einsortiert. Nach der Mutation mischen wir die Reihenfolge der Container.

5.2.6 Ersetzen und Abbruchkriterium

Nachdem alle Nachkommen erstellt sind ersetzen wir die gesamte alte Population durch die erzeugten Nachkommen. Diese *delete-all* Methode ist aufgrund ihrer Einfachheit weit verbreitet und hat den Vorteil, dass sie keine neuen Parameter einführt. (Burke u. a., 2014)

Als Abbruchkriterium definieren wir eine vorher festgelegte Anzahl der Generationen. Dies vereinfacht die Auswertung des Algorithmus.

5.3 Wahl der Hyperparameter

Aus den zuvor beschriebenen Komponenten des genetischen Algorithmus ergeben sich die folgenden Hyperparameter: Number of Generations, Population Size, Crossover Probability und Mutation Probability. Für die verschiedenen Probleme wurde eine Grid Search über 200 Parameterkombinationen durchgeführt, die die durchschnittliche sowie die beste Anzahl der Container auf 3 verschiedenen Problemgrößen untersucht. Die Auswertung dieser Grid Search gibt eine grobe Richtung für die Wahl der Parameter, ist aber aufgrund von unterschiedlichen Ergebnissen auf verschiedenen weiteren Problemen nicht aussagekräftig. Für die gewählten Probleme findet der Algorithmus schnell eine Lösung, da bereits durch First Fit Decreasing schnell gute Lösungen gefunden werden können. Um eine aussagekräftiges Ergebnis zu erhalten, müsste die Grid Search auf mehrere verschiedene Probleme angewendet werden. Dies wurde aufgrund der Komplexität nicht durchgeführt.

Wir orientieren uns an den Ergebnissen der Grid Search, obwohl diese möglicherweise nicht optimal gewählt sind. Wir wählen die Populationsgröße als 60, da wir in keinem unserer Tests durch eine größere Population bessere Werte erzielt haben. Gleichzeitig beeinflusst die Populationsgröße die Laufzeit und durch diese Wahl beschränkt sich die Laufzeit auf einen akzeptablen Rahmen. Die Anzahl der Generationen beschränken wir auf 30. Auch hier konnten wir kein Problem finden, auf dem der Algorithmus mit einer höheren Generationszahl bessere Ergebnisse erzielte. Die Mutations- und Rekombinationswahrscheinlichkeit wählen wir als 0.1 bzw. 0.7. Zwar zeigt die Grid Search, dass höhere Werte schneller konvergieren. Hier möchten wir aber vor allem eine zu schnelle Konvergenz vermeiden.

6 Ergebnisse

Im folgenden werden die Ergebnisse der verschiedenen Algorithmen vorgestellt. Um die Ergebnisse vergleichbar zu machen wurden die Algorithmen für die jeweiligen Testprobleme auf dem gleichen Computer ausgeführt. Dazu müssen zunächst zusätzliche Testprobleme erstellt werden.

6.1 Testprobleme

Das Ziel bei der Erstellung der Probleme ist es Testprobleme zu erhalten für die es deutlich schwieriger ist eine Lösung zu finden die nahe an dem Optimum liegt. Bei den zuvor verwendeten Problemen mit Problemgrößen bis 250 Objekten findet die mehrfache Anwendung der First Fit Heuristik bereits eine Lösung die um einen Container schlechter ist als die optimale Lösung.

Wir erstellen jeweils 10 Probleme für die Objektanzahlen 50, 100, 250 und 500. Dazu wurden volle Container der Größe (100, 100) zufällig in 2 bis 6 Objekte zerteilt. Damit ist die optimale Lösung der jeweiligen Probleme bekannt. Im Gegensatz zu den zuvor verwendeten Problemen sind in diesem Fall Gewicht- und Volumenwerte von 1-99 möglich. Gewicht und Volumen der Objekte wurden dabei so gewählt, dass Werte von größer als 70 seltener erstellt werden als Objekte mit Werten von 5 bis 40. Volumen und Größe eines Objekts können sich beliebig stark unterscheiden.

Problemgröße	GA	PSO	ACO	FF	FFD
50	14,52	14,52	14,52	14,52	14,52
100	7,22	13,65	7,22	10,43	9,01
250	4,37	10,77	4,10	6,77	4,80
500	3,10	9,48	3,00	5,51	3,20

Tabelle 2: Durchschnittliche Prozentuale Abweichung der Algorithmen (Genetischer Algorithmus (GA), Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), First Fit (FF), First Fit Decreasing (FFD)) vom optimalen Wert.

6.2 Ergebnisse der Particle Swarm Optimization

Lars

Wie in den durchschnittlichen Abweichungen in Tabelle 2 zu sehen, liefert die Implementation von Particle Swarm Optimization im Vergleich zu den optimalen Lösungen schlech-

te Ergebnisse. Es zeigte sich auf vielen verschiedenen Problemen ab frühen Phasen bereits eine Stagnierung. Vor allem bei großen Problemen und viel Variation in den Objektgrößen weist der Algorithmus Schwierigkeiten auf. Bessere Lösungen lassen sich mit problemspezifischen Parameterwerten erreichen, aber auch diese Lösungen fallen mit einem großen Unterschied zum optimalen Wert nicht gut aus. Das Finden besserer Parameter für ein Problem erweist sich aufgrund des hohen Rechenaufwands des Algorithmus über eine Grid Search als unpraktikabel, die Grid Search liefert sich im Ergebnis ähnelnde Parameterwerte. Die ein bis zwei besseren Ergebnisse über mehrere Durchläufe erscheinen durch zufällige Mutationen zu entstehen. Das dies trotz einer starken Hilfsheuristik mit First Fit und einer hohen Mutationswahrscheinlichkeit c_m auftritt, welche zusätzlich mit Objekten die Konflikte verursacht haben noch weiter vergrößert wird, weist auf größere Probleme in der gewählten Implementation oder einer schlechten Eignung von PSO als Hauptheuristik auf das Bin Packing Problem hin.

Problemgröße	GA	PSO	ACO	FF	FFD
50	0,41	2,22	18,52	1,07	0,00
100	0,96	5,25	26,31	2,20	0,00
250	3,60	18,64	62,36	6,47	0,01
500	12,48	57,49	99,73	26,17	0,03

Tabelle 3: Durchschnittliche Laufzeiten der Algorithmen in Sekunden (Genetischer Algorithmus (GA), Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), First Fit (FF), First Fit Decreasing (FFD))

6.3 Ergebnisse der Ant Colony Optimization

Tobias

Der ACO-Algorithmus findet meist bereits in der ersten Iteration eine sehr gute Lösung und die durchschnittliche Qualität der Lösungen steigt dann über die restlichen Iterationen nur um einen geringen Wert, wie in Abbildung 9 zu sehen. Durch die in Abschnitt 4.3.3 beschriebene First Fit Decreasing Heursitik werden alle Container so gut wie möglich gefüllt und es entstehen wenige schlechte Lösungen. Dennoch war bei jedem Test deutlich eine Verbesserungskurve in der durchschnittlichen Fitness und den durchschnittlichen Containerzahlen der einzelnen Iterationen zu erkennen, wenn man sie in einem geeignet kleinen Maßstab betrachtet. Durch die Heuristik wird auf einer kleinen Teilmenge des Suchraums gesucht, dies ist jedoch nicht unbedingt schlecht da für alle Lösungen eines Problems eine mindestens gleichwertig gute First Fit Decreasing Heuristik Lösung gefunden werden kann, wie in Abschnitt 4.3.3 gezeigt. Die Suche auf diesem kleineren Bereich des Suchraumes kann vorteilhaft sein, da weniger Lösungen untersucht werden müssen. Es ist aber auch gut möglich, dass in dieser kleineren Teilmenge einige der Lösungen weit voneinander entfernt sind, und es so für die Ameisen schwieriger wird durch Pheromonupdates von einer Lösung eine bessere Lösung zu erreichen. Der ACO Algorithmus ist in der Lage, auf manchen Problemen bereits in der ersten Iteration Lösungen zu finden die besser sind als die First Fit Decreasing Lösung, wie in Abbildung 9 zu sehen. Das liegt an der Implementation des ACO Algorithmus mit der First Fit Decreasing Heuristik. Bei den Ameisen ist die Wahl der Objekte nicht festgelegt wie bei FFD, die Objekte werden lediglich durch eine Wahrscheinlichkeitsverteilung gewählt die sich an FFD orientiert. So können auch bereits in der ersten Iteration Lösungen gefunden werden die besser sind

als die FFD Lösung. Die Laufzeit ist sehr hoch, was aber auch aufgrund den in Abschnitt 4.4.1 aufgeführten Punkten zu erwarten war. Die Laufzeit des ACO Algorithmus skaliert auf größeren Problemen etwas besser, zum Beispiel werden für die Probleme der Größe 500 im Vergleich zu Größe 250 nur circa 50% mehr Laufzeit benötigt obwohl die Objektanzahl doppelt so hoch ist. 3 Insgesamt ist die Laufzeit des ACO Algorithmus trotzdem die schlechteste Laufzeit unter den verglichenen Algorithmen auf allen untersuchten Problemen.

6.4 Ergebnisse des genetischen Algorithmus

Tim

Der genetische Algorithmus erzeugt zunächst eine große Anzahl an Containern und verbessert diese dann innerhalb kurzer Zeit. Dieses Verhalten lässt sich bei allen Problemen beobachten und ist in Abbildung 9 beispielhaft dargestellt. Die Lösung, die der Algorithmus berechnet hängt dabei von den Problemen ab. Bestehen die Container aus vielen Objekten mit geringem Gewicht und Volumen und gibt es eine optimale Lösung bei der alle Container vollständig gefüllt sind, dann findet der Algorithmus eine Lösung die nahezu optimal ist. Eine beispielhafte Verteilung der Objekte ist in Abbildung 8 dargestellt. Man erkennt, dass Container 9 einen geringen Füllstand besitzt und damit viele mögliche Lösungen der Größe 17 möglich sind. Diese Lösung wird bereits schnell mit den Heuristiken First Fit und First Fit Decreasing gefunden. Die optimale Lösung von 16 Containern wird dabei nicht erreicht.

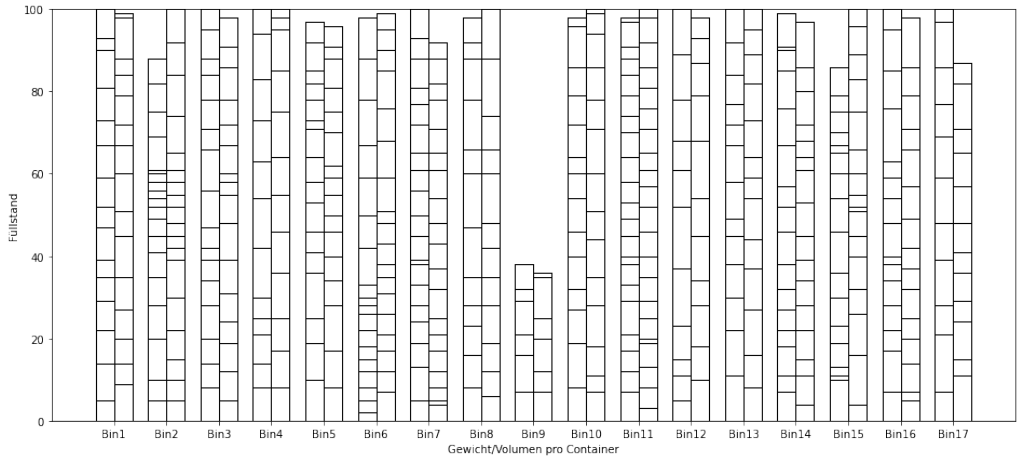

Abbildung 8: Eine Verteilung der 250 Objekte über die Container, wie sie beispielsweise der genetische Algorithmus berechnet.

Die Lösung des genetischen Algorithmus hängt dabei nicht alleine von der verwendeten Heuristik ab. Die Ergebnisse aus Abschnitt 6.5 zeigen, dass der Algorithmus im Durchschnitt bessere Lösungen als First Fit Decreasing findet. Dabei gibt es Probleme, bei denen der Algorithmus eine schlechtere Lösung als First Fit Decreasing berechnet. Der Unterschied zu First Fit Decreasing ist, wie in Tabelle 2 zu sehen, gering. Es ist es fraglich, ob die besseren Lösungen allein aus der parallelen Anwendung der vielen Ein- und Umsortierungen in Rekombination und Mutation entstehen oder ob diese auf die Vererbung von guten Bausteinen innerhalb der Elternchromosome zurückzuführen sind.

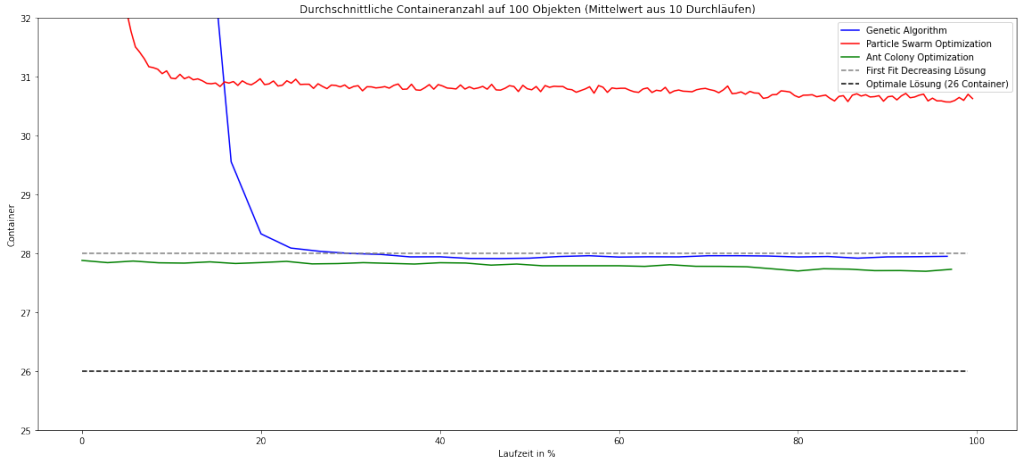

Abbildung 9: Durchschnittliche Containeranzahl der drei Algorithmen auf einem Problem mit 100 Objekten.

6.5 Vergleich der Algorithmen

Alle drei vorgestellten Algorithmen benutzen auf verschiedene Weise die First Fit bzw. die First Fit Decreasing Heuristik. Aus diesem Grund betrachten wir nicht nur die besten Lösungen und die Laufzeiten der drei Algorithmen, sondern vergleichen diese zusätzlich mit den beiden Heuristiken.

Für die First Fit Heuristik wurde eine Implementierung gewählt, die eine feste Anzahl an Iterationen durchführt und anschließend den besten gefundenen Wert ausgibt. Vor jeder Iteration werden die Objekte neu gemischt. Die Anzahl der Iterationen wurde so gewählt, dass die Laufzeit zwischen den Laufzeiten der PSO und des genetischen Algorithmus liegt. Die First Fit Decreasing Heuristik, sortiert die Objekte zunächst absteigend nach ihrer Summe aus Gewicht und Volumen. Anschließend werden diese mit First Fit einsortiert und die gefundene Containeranzahl ausgegeben. First Fit Decreasing liefert auch bei mehreren Durchläufen für ein gegebenes Problem immer die selbe Lösung, es muss nur einmal ausgeführt werden.

Als Testprobleme verwenden wir die in Abschnitt 6.1 beschriebenen neu erzeugten Probleme. Dementsprechend testen wir jeden Algorithmus, für jede der 4 Problemgrößen auf jeweils 10 Problemen. Jeder Algorithmus wird dabei 10 mal ausgeführt und die Ergebnisse anschließend gemittelt. Es wurde versucht die Parameter jeweils möglichst gut auf das entsprechende Problem abzustimmen. Einzelne Hyperparameter über alle Algorithmen gleich zu setzen und die Algorithmen anhand dieser zu vergleichen ist nicht sinnvoll, da die Algorithmen sehr unterschiedlich strukturiert sind und die Parameter individuell abgestimmt werden müssen um gute Lösungen zu finden.

Tabelle 2 zeigt eine Übersicht der Ergebnisse. Für jede Problemgröße wurde die prozentuale Abweichung von der optimalen Lösung berechnet. Die Auswertung zeigt, dass die ACO die besten Ergebnisse bezüglich der Anzahl der benötigten Container erzielt. Die Werte liegen dabei knapp über den Werten des genetischen Algorithmus und der First Fit Decreasing Heuristik. Die PSO liefert vor allem auf den größeren Problem schlechtere Ergebnisse.

Vergleicht man die Ergebnisse bezüglich der Laufzeiten, sieht man, dass vor allem First Fit Decreasing wesentlich schneller ist als die vorgestellten Algorithmen. Nach der First Fit Decreasing Heuristik erzielt der genetische Algorithmus die beste Laufzeit. Auf-

fällig ist, dass die ACO wesentlich längere Laufzeiten als die anderen Algorithmen benötigt. Die Ergebnisse sind in Tabelle 3 zusammengefasst.

7 Schlussfolgerung

Es wurden drei verschiedene Lösungsansätze für das zweidimensionale Bin Packing Problem vorgestellt und ausgewertet. Die Auswertung zeigt, dass die ACO und der genetische Algorithmus geeignete Lösungen für das Problem liefern. In den durchgeführten Tests berechnen beide Algorithmen durchschnittlich bessere Lösungen als die Heuristiken First Fit und First Fit Decreasing. Die Algorithmen geben keine Garantie eine bessere Lösung als die Heuristiken zu finden.

Die vorgestellte Implementierung der PSO eignet sich nicht für die gegebenen Probleme und liefert schlechtere Lösungen als die ACO, genetische Algorithmen, First Fit und First Fit Decreasing. Die PSO scheint durch den restriktiven Suchraum ungeeignet um alleine als Hauptalgorithmus gute Lösungskandidaten zum Bin Packing Problem zu erschaffen. In dieser Implementation sind Variationen mit Neuverteilungen nicht ausgiebig untersucht worden, doch es ist fraglich, dass ihre Einbindung selbst mit mehr Iterationen und Zeitaufwand der ACO und dem genetischen Algorithmus gleichwertige Lösungen liefert. Es empfiehlt sich eher stärker spezialisierte Implementierungen mit diversen Methoden gegen Stagnierung zu entwickeln, welche auf anderen Heuristiken aufbauend dann die Stärken der PSO wie die Suche und Konvergenz aller Partikel ausnutzt. Dies wurde bereits erfolgreich in der Arbeit Laurent und Klement, 2019 durchgeführt.

Durch die schlechte Laufzeit ist der reine ACO-Algorithmus leider nicht gut für das zweidimensionale Bin-Packing-Problem geeignet. Er hat jedoch gezeigt, dass er in wenigen Iterationen sehr gute Lösungen finden kann und meist auf diesen auch eine Verbesserung über die Gesamtlaufzeit erreicht. Der implementierte ACO-Algorithmus für das zweidimensionale Bin-Packing-Problem könnte gut mit einer lokalen Suche kombiniert werden, ähnlich zu dem Hybrid-Algorithmus von Levine und Ducatelle für das eindimensionale Bin-Packing-Problem.(Levine und Ducatelle, 2004) Ein solcher Hybrid-Algorithmus kann meist mit deutlich weniger Ameisen und Iterationen bessere Lösungen als ein reiner ACO Algorithmus finden. So könnte ein Hybrid-Algorithmus für das zweidimensionale Bin Packing Problem womöglich trotz der sehr hohen Laufzeit des alleinstehenden ACO Algorithmus eine praktikable Laufzeit erzielen.

Für den vorgestellten genetischen Algorithmus ergeben sich offene Fragestellungen. Zunächst können die Parameter durch eine umfangreichere Grid Search über verschiedene Probleme untersucht und entsprechend angepasst werden. Außerdem zeigt die Auswertung eine schnelle Konvergenz zu den lokalen Optima der Fitnesslandschaft. Hier kann untersucht werden, ob diese verlangsamt werden kann um andere Teile des Lösungsraums zu untersuchen. Der Algorithmus basiert auf der First Fit Decreasing Heuristik und es ist fraglich, wie sich der Algorithmus auf Problemen verhält für die First Fit Decreasing schlechte Lösungen berechnet. Da es für die First Fit Decreasing Heuristik, wie in Dósa, 2007 beschrieben, scharfe obere Schranken für die Anzahl der verwendeten Container gibt, wäre es interessant zu sehen ob entsprechende ähnliche Probleme auch für den zweidimensionalen Fall konstruiert und anschließend getestet werden können.

Literatur

Burke, Edmund K u. a. (2014). *Search methodologies: introductory tutorials in optimization and decision support techniques*. Springer.

Dorigo, Marco, Mauro Birattari und Thomas Stutzle (2006). „Ant colony optimization“. In: *IEEE Computational Intelligence Magazine* 1.4, S. 28–39. DOI: 10.1109/MCI.2006.329691.

Dósa, György (2007). „The tight bound of first fit decreasing bin-packing algorithm is FFD (I) $\leq$ 11/9OPT (I)+ 6/9“. In: *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer, S. 1–11.

Falkenauer, Emanuel (1996). „A hybrid grouping genetic algorithm for bin packing“. In: *Journal of heuristics* 2.1, S. 5–30.

Falkenauer, Emanuel, Alain Delchambre u. a. (1992). „A genetic algorithm for bin packing and line balancing.“ In: *ICRA*, S. 1186–1192.

Kennedy, J. und R. Eberhart (1995). „Particle swarm optimization“. In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. Bd. 4, 1942–1948 vol.4. DOI: 10.1109/ICNN.1995.488968.

Laurent, Arnaud und Nathalie Klement (Jan. 2019). „Bin Packing Problem with priorities and incompatibilities using PSO: application in a health care community“. In: *IFAC-PapersOnLine* 52, S. 2596–2601. DOI: 10.1016/j.ifacol.2019.11.598.

Levine, J und F Ducatelle (2004). „Ant colony optimization and local search for bin packing and cutting stock problems“. In: *Journal of the Operational Research Society* 55.7, S. 705–716. DOI: 10.1057/palgrave.jors.2601771. eprint: https://doi.org/10.1057/palgrave.jors.2601771. URL: https://doi.org/10.1057/palgrave.jors.2601771.

Mitchell, Melanie (1998). *An introduction to genetic algorithms*. MIT press.

Poli, Riccardo, James Kennedy und Tim Blackwell (Okt. 2007). „Particle Swarm Optimization: An Overview“. In: *Swarm Intelligence* 1. DOI: 10.1007/s11721-007-0002-0.

Shi, Yuhui und B.Gireesha Obaiahnahatti (Juni 1998). „A Modified Particle Swarm Optimizer“. In: Bd. 6, S. 69–73. ISBN: 0-7803-4869-9. DOI: 10.1109/ICEC.1998.699146.