Anwendung des Cuckoo Search Algorithmus auf das Traveling Salesman und das Sequential Ordering Problem

1st Lennard Rössig
Hochschule Hannover
Fakultät IV - Abteilung Informatik
Hannover, Niedersachsen
lennard.roessig@stud.hs-hannover.de

2nd Florian Rückauf
Hochschule Hannover
Fakultät IV - Abteilung Informatik
Hannover, Niedersachsen
florian.rueckauf@stud.hs-hannover.de

***Zusammenfassung*—In dieser Arbeit wurde der metaheuristischen Algorithmus Cuckoo Search (CS) implementiert, um damit zwei Optimierungsproblem zu lösen. Die grundlegende Idee des Algorithmus ist abgeleitet von dem Brutparasitismus einiger Kuckuck-Arten. Das Bewegungsmuster des Kuckucks wird in dem Algorithmus durch den Levy Flight erzeugt. Dieser versucht das Verhalten von Insekten und einigen Vögel zu imitieren. Es wurden die beiden NP-harten Probleme, des Travelling-Salesmans (TSP) und des Sequenc-Ordering (SOP) betrachtet und mit dem Algorihtmus gelöst. Da die Arbeit im Kontext eines Projektes erscheint, in dem mehrere metaheuristische Algorithmen verglichen wurden, sind zum Schluss nur die Benchmarks und Ergebnisse des hier angewandten Algorithmus gelistet.**

***Index Terms*—algorithm; cuckoo search; Levy Flight; metaheuritics; nature-inspired strategy; optimization**

I. Einleitung

Viele Optimierungsprobleme sind NP-hart weshalb die Existenz von effizienten Algorithmen, die diese Problem lösen, fraglich ist. Für solche Probleme können Metaheuristiken zur näherungsweisen Lösung eingesetzt werden. Metaheuristiken definieren dabei eine abstrakte Folge von Schritten, die auf beliebige Problemstellungen angewandt werden können. Die einzelnen Schritte müssen allerdings wieder problemspezifisch implementiert werden. Erfolg und Laufzeit hängen von der Definition und Implementierung dieser Schritte ab.

Ein Bereich der Metaheuristiken sind die naturanalogenen Algorithmen, deren Vorgehen sich aus der Natur ableitet. In diesem Zusammenhang wird der Cuckoo Search (CS) Algorithmus betrachtet und einmal auf das Traveling Salesman Problem (TSP) und Sequential Ordering Problem (SOP) angewandt. Der CS ist ein noch recht junger Algorithmus, der 2009 von Xin-She Yang und Suash Deb entwickelt wurde [1]. Die Experimente wurden für das euklidische, symmetrische TSP durchgeführt und die genutzten Daten stammen aus der TSPLIB Library [12].

II. Traveling Salesman Problem und Sequential Ordering Problem

A. Traveling Salesman Problem

Bei dem TSP wird ein Hamiltonkreis mit den minimalen Kosten H_{opt} in einem vollständigen, gewichteten Graphen $G = (V, E)$ gesucht [2], [3]. Die Städte werden hier durch die Knoten V repräsentiert und die Verbindungen zwischen den Städten u und v sind die Kanten $E = (u, v)$. Zu jeder Kante gibt es eine Länge $c_{u,v} \geq 0$. Da sich die Knoten hierbei in dem zweidimensionalen euklidischen Raum befinden, handelt es sich um das euklidische TSP. Die Länge $c_{u,v}$ ist somit für alle u, v in G die euklidische Entfernung zwischen den beiden Städten, die den Knoten u und v zugeordnet sind. Betrachtet wurde das symetrische TSP, bei dem die Strecke zwischen zwei Städten $c_{u,v}$ gleich der Strecke $c_{v,u}$ ist.

B. Sequential Ordering Problem

Beim Sequential Ordering Problem (SOP) handelt es sich wie beim TSP um einen Graphen im euklidischen zweidimensionalen Raum. Im Gegensatz zum TSP sind die Eigenschaften einer korrekten (engl. valid) Reihenfolge strenger. Hierbei müssen bestimmte Bedingungen (engl. Condition) hinsichtlich des Aufbaus einer solchen Reihenfolge eingehalten werden. Ist P die Menge aller Knoten des SOP's so lässt sich jeder Knoten mittels $P_i | i \in \mathbb{N}$ referenzieren. Um eine korrekte Reihenfolge zu konstruieren, muss diese mit P_0 starten und auf P_{n-1} enden. Bei P_0 und P_{n-1} handelt es sich um einen festen Start- und Endknoten. Der Endknoten ist dabei der letzte Knoten des Problems, daher ist $n = |P|$. Zusätzlich zu dieser Einschränkung besitzen die Knoten untereinander Beziehungen/Bedingungen. Eine Bedingung bedeutet, dass ein bestimmter Knoten vor einem anderen Knoten in der Reihenfolge liegt. Besitzt also P_x eine Bedingung zu P_y so muss P_y vor P_x in der Reihenfolge besucht werden. Jeder Knoten besitzt eine Bedingungsmenge $B_i | i \in \mathbb{N}$, wobei i die Bedingungsmenge des i-ten Knotens des Problems P meint. Für eine korrekte Reihenfolge müssen also folgende Bedingungen gelten:

- jeder Knoten wird nur einmal besucht
- alle Bedingungen werden eingehalten

In den Bedingungen lassen sich die festen Start- und Endknoten definieren. Die Bedingungsmenge des Startknotens (1) ist leer, da dieser ganz vorne stehen muss. Im Gegensatz enthält die Bedingungsmenge des Endknotens (2) alle anderen Knoten, da diese vor ihm besucht werden müssen.

$$B_0 = \emptyset \tag{1}$$

$$B_{n-1} = P \backslash \{P_{n-1}\} \tag{2}$$

Für die Bedingungen gilt zusätzlich, dass ein Knoten nicht zu sich selber eine Bedingung besitzen kann. Des Weiteren kann ein Knoten aber nicht eine Bedingung zu einem anderen Knoten besitzen, wenn dieser eine Bedingung zu ihm hat. Falls eine solche Beziehung erlaubt wäre, wäre das SOP unlösbar, da beide Knoten jeweils den anderen Knoten vor sich erwarten. Die Bedingungsmenge alle andere Knoten (3) darf nicht sich selbst und den Endknoten enthalten, aber muss den Startknoten besitzen.

$$B_i \subseteq P \backslash \{P_i, P_n\} | B_i \cap \{P_0\} \neq \emptyset \tag{3}$$

III. Grundlagen des Cuckoo Search

A. Brutverhalten des Kuckucks

Die Grundidee des Algorithmus wurde von dem Brutverhalten des namensgebenden Kuckucks abgeleitet. Das Brutverhalten des Kuckucks ist der sogenannter Brutparasitismus. Bei diesem Verhalten wird das eigene Gelege nicht selbst bebrütet, sondern es wird ein Ersatzwirt gesucht, der das Ei ausbrütet und sich auch um die Fütterung und Aufzucht des fremden Nachwuchses kümmert.

Die Brutschmarotzer müssen sich durch dieses Verhalten nicht um die zeitintensive Aufzucht der Nachkommen kümmern. Sie verringern somit den Aufwand für die Brutpflege und haben mehr Zeit Nahrung für sich selbst zu finden. Dadurch sind sie in der Lage mehr Eier zu legen und können so potenziell mehr Nachkommen erzeugen.

Unter den Wirtstieren gibt es aber auch solche, die die fremden Eier entdecken. Dann gibt es zwei verschiedene Strategien. Werden die fremden Eier entdeckt, so werden sie entweder aus dem Nest geschmissen oder das ganze Gelege wird verlassen und ein neues angelegt.

Gerade in dem interspezifischen Brutparasitismus haben sich verschiedene Anpassungen entwickelt, um das Risiko der Entdeckung zu minimieren. Bei einigen Arten erfolgt die Eireifung simultan zu der des Wirtstieres. Die Eier des Kuckucks besitzen meist eine kürzere Brutzeit, sodass diese als erstes schlüpfen um dann die anderen Eier aus dem Nest zu schmeißen. Die meisten Kuckuckskücken wachsen in den ersten Tagen schneller als die Küken der Wirtseltern und haben dadurch einen entscheidenden Fütterungs- und Wachstumsvorteil. Zudem erfolgt die Eiablage in beschleunigter Form, d.h. das Ei wird im Eileiter aufbewahrt und kann dann im Gelegenheitsfall schnell gelegt werden. Es wurden in der Natur Eiablagezeiten von nur 10 Sekunden gemessen. Manche Kuckucksarten haben die Größe und Farbe der Eier denen der Wirtseier angepasst, sodass fast kein Unterschied zwischen den Eiern mehr zu erkennen ist. Diese Tiere bevorzugen dann in der Regel eine Wirtsvogelart.

B. Basis Version

Das oben beschriebene Verhalten wird in dem Cuckoo Search genutzt, um den Suchraum nach einer optimalen Lösung zu durchsuchen. Der Algorithmus funktioniert dabei wie folgt [1]:

- Ein Set von Nestern mit einem Ei wird zufällig in dem Suchraum platziert. Die Anzahl der Nester ist dabei fest und ändert sich im Laufe des Algorithmus nicht.
- Eine Anzahl von Kuckucken durchfliegt den Suchraum und generiert damit eine neue Lösung. Diese neue Lösung wird in ein zufällig gewähltes Nest gelegt.
  In diesem Szenario repräsentiert jedes Ei in einem Nest eine Lösung und jeder Kuckuck repräsentiert eine neue Lösung. Das Ziel ist es, die neuen, potenziell besseren Lösungen zu nehmen und damit eine andere, nicht so gut Lösung aus dem Nest zu verdrängen.
- Die Wahrscheinlichkeit, dass ein Ei durch den Wirt entdeckt wird, liegt dabei in dem Bereich $p_a[0, 1]$. Tritt dieser Fall ein, so wird entweder das Ei aus dem Nest geschmissen oder das ganze Nest wird verlassen und ein neues wird gebaut. Für die Einfachheit wird die letztere Möglichkeit angenommen, d.h. mit der Wahrscheinlichkeit von p_a werden n Nester durch neue Nester ersetzt, die eine neue, zufällige Lösung besitzen.

Die besten Nester mit den besten Lösungen werden dann genutzt, um daraus neue Generationen zu erzeugen. Der Algorithmus kann noch dahin geändert werden, dass ein Nest mehrere Eier enthält und somit eine Menge an Lösungen beinhaltet.

Algorithm 1 zeigt den Pseudo Code für den Cuckoo Search.

Algorithm 1 Cuckoo Search

```
Objective function f(x), x = (x_i, ..., x_d)^T
Generate initial population of n host nests x_i (i = 1, ..., n)
while (t < MaxGeneration) or (stop criterion) do
    Get a cuckoo randomly by Lévy Flights
    Evaluate its quality/fitness F_i
    Choose a nest among n (say, j) randomly
    if F_i > F_j then
        replace j by the new solution
    end if
    A fraction (p_a) of worse nest are abandoned
    and new ones are buildt
    Keep the best solutions (or nest with quality solutions)
    Rank the solutions an find the current best
end while
Postprocess results and visualization
```

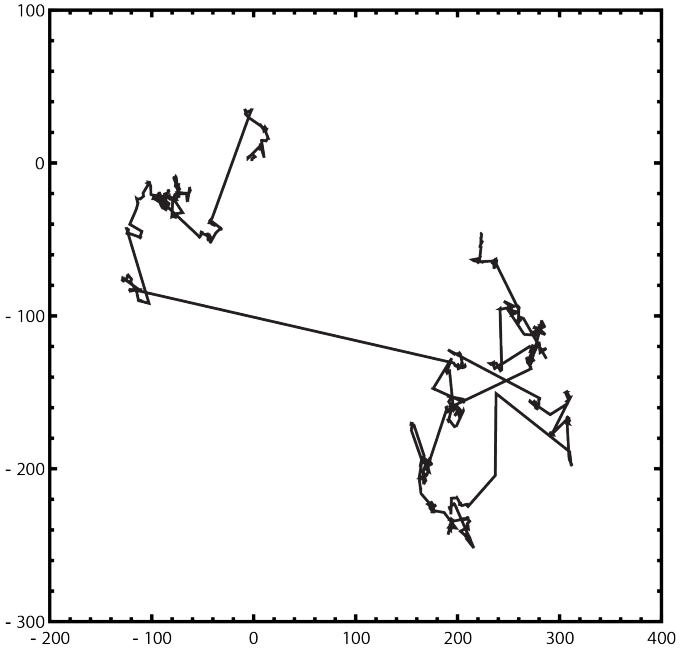

Abbildung 1. Lévy Flight beginnend im Punkt (0,0).

Um eine neue Generation von Lösungen von einem Kuckuck zu erzeugen, wird der Lévy Flight genutzt. In verschiedenen Studien wurde gezeigt, dass das Flugverhalten verschiedener Tiere und Insekten die Eigenschaft eines Lévy Flights besitzen [4], [5], [6]. Die Definition des Lévy Flights stammt von Mathematikern der Chaostheorie und ist sehr nützlich, um zufällige oder pseudo-zufällige natürliche Phänomena zu beschreiben. Abbildung 1 zeigt ein Beispiel eines Lévy Flights beginnend im Punkt (0,0).

Die Erzeugung einer neuen Lösung über den Lévy Flight wird mit der Gleichung

$$x_i^{t+1} = x_i^t + \alpha L(s, \lambda) \tag{4}$$

beschrieben. Der Parameter $\alpha > 0$ beschreibt den Skalierungsfaktor der Schrittweite. In [7] wird α durch $\alpha = O(L/10)$ berechnet. Bei Problemen in denen verhindert werden soll, das zu weit geflogen wird, kann $\alpha = O(L/100)$ genutzt werden. L ist dabei die Skalierung für das spezifische Problem.

Um den Lévy Flight zu implementieren ist ein schneller Algorithmus nötig, der den Lévy Flight approximiert. In [8] werden drei Algorithmen für die Approximation des Lévy Flights verglichen. Der Mantegna Algorithmus benötigt für die Aufgabe die unten aufgeführten drei Schritte. Die Schrittweite s wird mit

$$s = \frac{u}{|v|^{\frac{1}{\beta}}} \tag{5}$$

berechnet, wobei u und v durch die Normalverteilung

$$u = N(o, \sigma_u^2), v = N(o, \sigma_v^2) \tag{6}$$

gegeben sind. Die Varianz σ wird dabei mit

$$\sigma_u = \left(\frac{\Gamma(1+\beta)\sin(\pi\beta/2)}{\Gamma[(1+\beta)/2]\beta 2^{(\beta-1)/2}} \right)^{\frac{1}{\beta}}, \sigma_v = 1 \tag{7}$$

berechnet, wobei $1 \leq \beta \leq 2$ und Γ die Gammafunktion ist.

C. Verbesserte Version

In der Basisversion wird noch nicht das unterschiedliche Verhalten der Kuckucke bei der Auswahl des Nests in betracht gezogen. Quaarrad et al. [9] beschreibt eine Möglichkeit die Art und Weise, wie ein Kuckuck den Suchraum erkundet, zu verbessern. Ein Kuckuck kann eine gewisse Intelligenz besitzen, so dass er bessere Lösungen findet. Dadurch kann die Intensivierung und die Diversifikation der Suche durch den Kuckuck beeinflusst werden. Angepasst an das intelligenter Verhalten einiger Kuckucke, führt ein Teil der Kuckucke einen initialen Schritt zu einer neuen Lösung über den Lévy Flight durch und suchen von dort aus eine neue, bessere Lösung über eine lokale Suche.

Ausgehend davon, kann die Population der Kuckucke in dem verbesserten CS in drei Typen unterteilt werden.

1) Der Kuckuck sucht von der besten Position aus neue Gebiete welche besser Lösung beinhalten können, durch eine zufällige Auswahl.
2) Ein Teil p_a der Kuckucke sucht eine Lösung weit weg von der besten Lösung
3) Ein Teil p_c der Kuckucke sucht nach Lösungen von der aktuellen Position und versucht diese zu verbessern. Sie bewegen sich von einer Region zu einer anderen durch den Lévy Flight um die beste Lösung in jeder Region zu bekommen, ohne in einem lokalen Minimum festsitzen zu bleiben.

Diese Anpassungen verbessern die Intensität der Suche um die aktuell besten Lösungen und gleichzeitig wird die Zufälligkeit mit der neue Gebiete erschlossen werden erhöht. Dadurch wird die Performanz und die Effizienz verbessert, da weniger Iterationen benötigt werden und er eine besser Resistenz gegenüber lokalen Minima besitzt [9].

IV. Anpassung des Cuckoo Search für die betrachteten Probleme

A. Cuckoo Search für das Traveling Salesman Problem

Bekanntlich befindet sich das TSP in einem kombinatorischen Raum. Der CS wurde entworfen, um Lösungen in einem kontinuierlichen Raum zu finden. Um jetzt den CS für die Lösung des TSP anzupassen, müssen die fünf Hauptbegriffe des CS (Ei, Nest, Zielfunktion, Suchraum und Lévy Flight) an das TSP angepasst werden.

Im CS repräsentiert ein Ei eine möglich Lösung das Problems. Beim TSP ist eine mögliche Lösung eine Route durch alle Städte. Diese Route ist ein Hamiltonkreis.

Ein Nest enthält die Eier, also enthält ein Nest ein oder mehrere mögliche Lösungen des Problems. In diesem Fall also ein oder mehrere Hamiltonkreise.

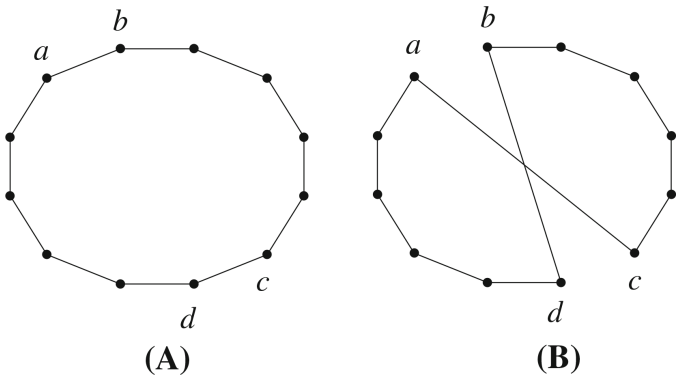

Abbildung 2. 2-opt-move. a initiale Tour. b durch 2-opt-move erzeugte Tour. Kanten (a,b) und (d,c) wurden durch Kanten (a,c) und (b,d) ersetzt

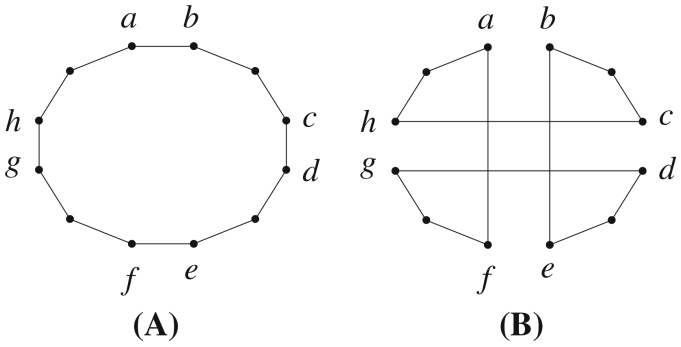

Abbildung 3. Double-Bridge-move. a initiale Tour. b durch Double-Bridge-move erzeugte Tour. Kanten (a,b), (c,d), (e,f) und (g,h) wurden durch Kanten (a,f), (g,d), (e,b) und (g,d) ersetzt

Beim TSP ist die kürzeste Route zwischen den Städten gesucht. Als Zielfunktion wird hier die Länge des Hamiltonkreises genutzt.

Da die Positionen der Städte fest sind und diese nicht verändert werden können, lässt sich die Lösungen nur durch die Reihenfolge der besuchten Städte verändern. Der Suchraum umfasst somit alle möglichen Anordnungen der Städte. Die Reihenfolge der besuchten Städte lässt sich mit dem 2-opt-move [10] und den Double-Bridge-move [10] verändern. Der 2-opt-move wird dabei genutzt, um eine kleine Änderung in dem Suchraum zu erzeugen und der Double-Bride-move wird für große Änderungen genutzt. Wie in Abbildung 2 zu sehen, entfernt der 2-opt-move zwei Kanten und erstellt zwei neue Kanten. Der Double-Bride-move entfernt vier Kanten und erstellt neue Kanten, so wie in Abbildung 3 zu sehen ist.

Um den Suchraum zu durchsuchen, haben wir die Möglichkeit dies in kleinen Schritten, oder in großen Schritten zu tun. Um jetzt die Schrittweise an den Lévy Flight anzupassen, werden die Werte des Lévy Flight auf ein Intervall zwischen 0 und 1 abgebildet. Das Intervall kann dann in verschiedene Abschnitte unterteilt werden, so dass wir die Schrittlänge anpassen können. Je nach Wert des Lévy Flight ergibt sich dann die Schrittweite nach

1) $[0, i[$ ein Schritt durch 2-opt-move
2) $[(k-1) * i, k * i[$ k Schritte durch $k * 2 - opt - move$
3) $[k * i, 1[$ ein großer Schritt durch double-bridge-move

B. Cuckoo Search für das Sequential Ordering Problem

Der größte Unterschied für den Cuckoo-Algorithmus vom TSP zum SOP ist, dass der Algorithmus Lösungen produzieren kann die nicht mehr valide sind. Für das TSP wurden nur gültige Lösungen erstellt, wenn die Operatoren auf gültigen Reihenfolgen arbeiten. Daher ist es ausreichend dort, darauf zu achten, dass die Initialisierung nur gültige Reihenfolge erbrachte. Anschließend konnte durch die Operaten Double-Bridge-move 3 und 2-opt-move 2 keine ungültigen Ergebnisse erzeugt werden. Mit SOP reicht es nicht mehr nur die Initialisierung gültig zu gestalten, da auch die beiden Operationen zu ungültigen Ergebnissen führen können. Im folgenden werden drei Varianten vorgestellt, die den Umgang mit ungültigen Reihenfolge auf unterschiedliche Art lösen. Anschließend wird die Implementierung einer dieser Varianten für den Cuckoo's Algorithmus gezeigt.

C. Handicap

Beim Handicap werden ungültige Ergebnisse in der Population beibehalten. Ihre Fitness hingegen wird mittels eines Handicaps verschlechtert. Das Abschwächen ist die "Bestrafung" dafür, dass es sich bei der Lösung um ein ungültiges Ergebnis handelt. Der Vorteil dieser Variante ist, dass auch eine solche Lösung noch gute Bestandteile beinhalten, die vom Schwarm noch sinnvoll übernommen werden können. Besitzen diese aber keine guten Bestandteile, verschlechtern sie sich durch das Handicap stark, sodass diese aus der Population ausscheiden. Nachteil des Ansatzes ist, dass das Balancing des Handicaps enorm schwer ist. Ist das Handicap viel zu stark, werden ungültige Lösung nicht wirklich betrachten und könnten auch direkt gelöscht werden. Ist wiederrum das Handicap zu schwach, kann es passieren, dass die beste Lösung des Algorithmus ein ungültiges Ergebnis ist, was wiederrum unerwünscht ist. Erst recht ist das Handicap im Falle von SOP/TSP schwer einzubringen, da die Fitness hierbei die Länge der Pfade ist. Die Problemstellung äußert sich also direkt in der Fitness, was wiederrum bedeutet, dass eine Fitness von 100 in dem einem Fall gut und in einem anderen Fall schlecht ist. Das Handicap müsste daher variable gestaltet werden. Problem hierbei ist, dass das optimale Ergebnis zu Beginn nicht bekannt ist, wodurch ein relatives Handicap schwer umzusetzen ist. Des Weiteren besitzt der Cuckoo's Algorithmus nicht das klassische Verhalten eines Schwarms, da die einzelnen Individuen sich nicht stark aneinander Orientieren. Aufgrund des fehlenden Schwarm-Verhaltens, dem Balancing und dem Problem des schwer an zu passenden Handicaps, haben wir uns gegen diese Methode entschieden.

D. Operatoren

Eine weitere Variante ist es, die Operatoren *Double-Bridge-Move* und *2-opt-swap* soweit anzupassen, dass diese nur noch gültige Ergebnisse erzeugen. Die beiden Operatoren des Cuckoo's Algorithmus besitzen eine einfache Struktur ohne

weitere Metriken, wodurch eine solche Veränderung schwer ist. Des Weiteren würde eine solche Anpassung die Komplexität der Operatoren erhöhen, wodurch die Grundidee eines einfach zu verstehenden Optimierungsalgorithmus verloren gehen würde. Grad die Einfachheit des Cuckoo's Algorithmus macht diesen aus, weshalb wir uns gegen diese Anpassung entschieden haben.

E. Reparierung

Im Repair-Schritt wird ein ungültiger Weg wieder in einem gültigen Weg transformiert, wobei versucht wird den gültigen Wissensgehalt, innerhalb der Lösung, nicht zu verändern. Dies bedeutet, dass gerade so viel geändert wird, dass die Reihenfolge wieder korrekt ist. Veränderungen darüber hinaus, könnte dazu führen, dass kleiner Veränderungen durch die Operanten sonst verloren gehen würden. Zusätzlich ist der *Repair-Schritt* als ein unnatürlicher Eingriff in die natürliche Lösungsfindung des Cuckoo's Algorithmus zu sehen, weswegen dieser möglichst klein ausfallen sollte. Der *Rapair-Schritt* schien uns für den Cuckoo's Algorithmus am passendsten, da dieser die Komplexität des Algorithmus nicht erhöht, aber gleichzeitig keine weiteren Probleme mit sich bringt. Die genaue Implementierung des Schrittes wird im folgenden Abschnitt beschrieben.

F. Implementierung

Im Cuckoo's Algorithmus, muss an drei Orten festgestellt werden, dass nur gültige Reihenfolgen erstellt werden. Die einzigen Punkte im Algorithmus wo ungültige Ergebnisse auftreten können, ist bei der *Initialisierung* und nach den beiden Operatoren. Da die *Initialisierung* TSP Reihenfolgen erstellt, werden diese im Nachhinein so transformiert, dass sie dem SOP entsprechen. Der *Two-Opt-Swap* kann zu ungültigen Reihenfolgen führen, muss dies aber nicht. Daher muss nach jeder Ausführung geprüft werden, ob es sich noch um eine gültige Reihenfolge handelt. Ausnahme ist hierbei, dass wenn mehrere *Two-Opt-Swaps* hintereinander ausgeführt werden, nur das letzte Ergebnis wieder zu einem gültigen transformiert wird. Dies folgt wieder dem Prinzip, dass nur möglichst wenig verändert/eingegriffen werden soll. Der *Double-Bridge-Move* führt immer zu einer ungültigen Reihenfolge, dies folgt aus der Definition der Methode. Daher wird hier nach jeder Durchführung die Reihenfolge wieder repariert. Der Ablauf des *Repair-Schrittes* ist immer der gleiche und wird im Pseudocode 2 dargestellt.

Die Laufzeit vom *Repair-Schritt* beträgt dabei O(n^2), wobei n die Länge der Reihenfolge R ist. Dies ergibt sich aus der Tatsache, dass jede $swap$-Operation wieder zu einer neuen Verletzung der Bedingungen führen kann, weswegen ab Stelle i erneut begonnen wird. Um also alle möglichen Verletzungen der Bedingungen zu finden, muss jeder Knoten n_x mit jedem nachfolgendem Knoten n_y geprüft werden. Wie oben beschrieben, ist die Laufzeit abhängig von der Länge der Reihenfolge und dadurch direkt auch von der Größe des SOP's. Durch diese Abhängigkeit, die zuvor nicht bestand, wird die Auswertungen von großen SOP's erheblich langsamer.

Algorithm 2 Repair

```
Erhalte Reihenfolge R mit Länge n
while (i < n) do
    y = i + 1
    B_i ist die Bedingungsmenge des Knoten i
    while (y < n) do
        if B_i ∩ {P_y} ≠ ∅ then
            tausche Knote i mit Knoten y
            breche innere Schleife ab
        end if
    end while
    Erhöhe i + 1, falls innere Schleife nicht abgebrochen
end while
gültige Reihenfolge R
```

V. Evaluierung

In diesem Abschnitt gehen wir auf die Parameter des Cuckoo's Algorithmus ein und prüfen ihren Einfluss auf die Qualität des Ergebnisses. Dabei untersuchen wir verschiedene Anzahlen an Nester und Iterationen sowie unterschiedliche Wahrscheinlichkeiten P_a dafür, dass ein Ei entdeckt und aus dem Nest geworfen wird. Die Ergebnisse werden anschließend genutzt um den Cuckoo's Algorithmus auf mehreren Datensätzen zu testen.

A. Parameter

Alle Durchgeführten Tests arbeiten dabei auf dem Datensatz *a280.tsp* der 280 Knoten besitzt. Beginnen tun wir mit dem Parameter P_a, dieser beschreibt die Entdeckungswahrscheinlichkeit eines Eies und somit das Ausscheiden aus der Population. Dafür haben wir den Algorithmus mit fünf unterschiedlichen Werten laufen lassen, wobei diese von 10% bis zu 30% reichen. Die Anzahl der Nester lag bei 50 und die Generationen bei 1000. Anhand der Ergebnisse (Abb 4) lässt sich erkennen, dass der Einfluss des Parameters auf die Güte der Lösung sehr gering ist. Eine niedrigere Entdeckungswahrscheinlichkeit sorgt geringfügig für ein schnelleres Anpassungsverhalten, hingegen sich dies im Endergebnis nicht wiederspiegelt.

Für ein gutes Anpassungsverhalten sollte P_a daher nicht zu niedrig gesetzt werden, aber auch nicht zu hoch, damit noch eine Anpassung stattfindet. Die weiteren Tests werden daher mit einer $P_a = 0.15$ durchgeführt. Die Veränderung des P_a Faktors hat keinerlei Einfluss auf die Laufzeit des Algorithmuses.

Als nächsten wurde der Einfluss der Nester auf das Ergebnis geprüft. Dafür wurden acht Durchläufe mit steigender Anzahl an Nestern von 25 bis 200 gemacht. Hierbei wurden eine P_a von 0.15 und 1000 Generationen genutzt. Das Ergebnis (Abb. 5) zeigt, dass mit steigerender Anzahl der Nester wieder ein schnelleres Anpassungsverhalten vorliegt, dass aber wiederrum mit ansteigender Anzahl an Generationen verschwindet. Damit hat die Anzahl der Nester auch wieder nur einen geringen Einfluss auf die Endqualität des Ergebnisses. Daher setzen wir für weitere Test die Anzahl an Nestern auf 50

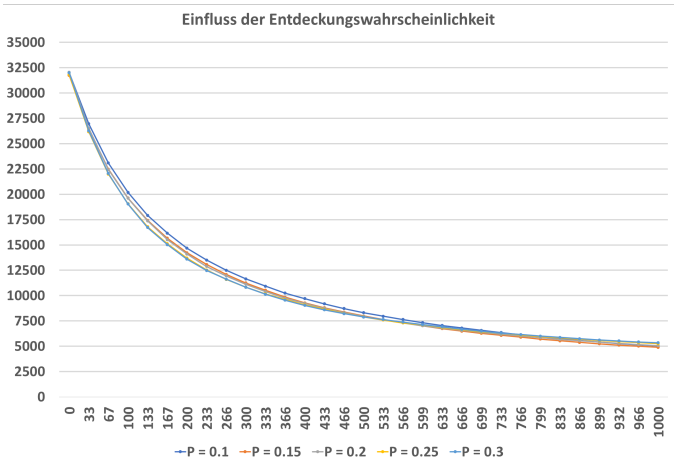

Abbildung 4. Einfluss des Parameter P_a auf das Ergebnis

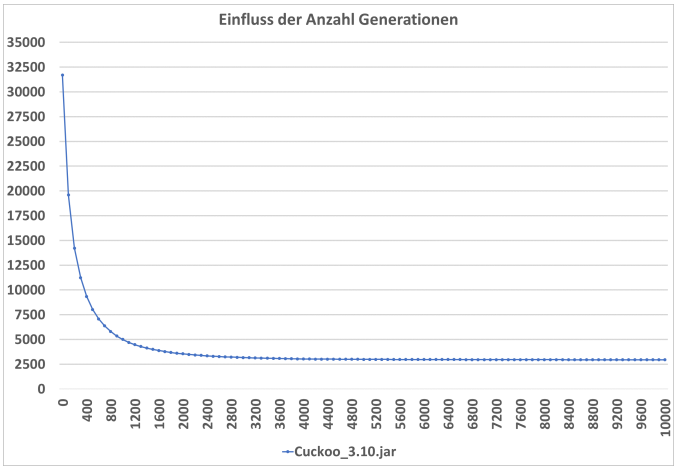

Abbildung 6. Einfluss Anzahl der Generationen auf das Ergebnis

fest. Durch Erhöhung der Nester erhöht sich die Laufzeit proportional linear.

In Abbildung 6 ist der Einfluss des Generation Parameters zu erkennen. Für den Durchlauf wurden 50 Nester und eine P_a von 0.15 genutzt. Zu erkennen ist, dass der Algorithmus bis 3500 Iterationen sich langsam immer weiter anpasst. Ab 3500 nimmt diese Kurve immer weiter ab, bis so gut wie keine Verbesserung mehr stattfindet. Diese Schranke lässt sich mittels Anpassung der Anzahl der Nester und des Werte P_a weiter nach links, und somit < 3500, oder rechts > 3500 verschieben. Generell lässt sich daher sagen, dass mit den von uns gewählten Parametern nur noch geringe Verbesserungen über 4000 Iterationen eintreten. Der Einfluss auf die Laufzeit ist linear und dementsprechend auch mit hohen Zahlen vertretbar.

Daher sind unsere Parameter für den Cukoo's Algorithmus:

- $P_a = 0.15$
- Nester = 50
- Generationen = 4000

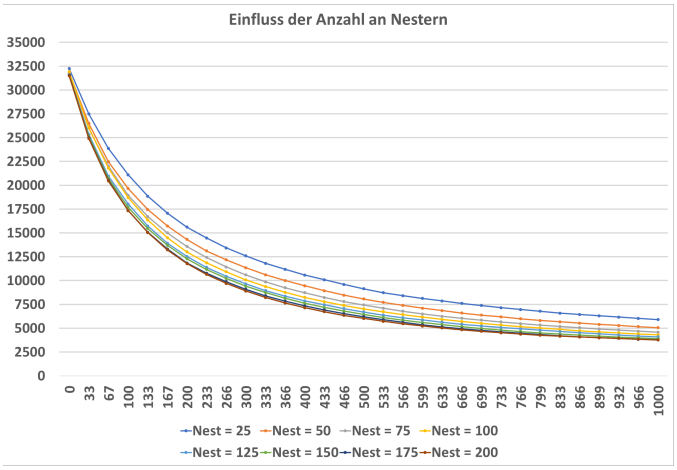

Abbildung 5. Einfluss Anzahl der Nester auf das Ergebnis

B. *Ergebnisse*

Im folgendem werden das SOP sowie das TSP anhand einiger Beispiele aus der Datensatzsammlung der Uni-Heidelberg [12]getestet. Dabei werden die Parameter aus dem vorherigen Abschnitt genutzt. Die Ergebnisse sind der Mittelwert aus 10 Durchläufe auf dem jeweiligen Datensatz.

Der Cuckoo's Algorithmus für kleine TSP erbringt gute Ergebnisse in linearer Zeit. Hingegen in der Tabelle auch zu erkennen, dass mit wachsender Anzahl an Knoten, das Ergebnis immer schlechter wird. Für den Datensatz *d1291* beträgt der Abstand zwischen dem besten und den vom Cuckoo erzeugten $222502 - 50801 = 171701$. Das bedeutet, dass der Cuckoo's Algorithmus mehr als $3x$ vom besten Ergebnis abweicht, was enorm ist. Wiederrum bei kleineren TSP's performt der Algorithmus gut. Durch die Unabhängigkeit des Algorithmuses gegenüber der Anzahl an Knoten in einem TSP, ist die Laufzeit auch bei großen TSP's noch sehr gut. Nur die Berechnung der Fitness bremst hier den Algorithmus aus.

Im Gegensatz zum TSP, lässt sich beim SOP die Güte des Ergebnisses nicht anhand der Komplexität des Problemes ablesen. Die Komplexität eines SOP's liegt nicht nur innerhalb der Anzahl an Knoten sondern in der Bedingungsmenge. Dies lässt sich gut in der Tabelle erkennen, da die Ergebnisse nicht gleichmäßig schlechter werden mit zunehmender Anzahl an Knoten. Damit scheint die Güte des Ergebnisses unabhängig von der Anzahl an Knoten zu sein. Da die Laufzeit quadratisch

Datensatz	Start	End	Best	∼%-Abweichung
berlin52	26174	7949	7542	0,05
kroA100	151054	22742	21282	0,07
eil101	3094	679	629	0,08
ch130	42378	6683	6110	0,09
kroA150	233162	29204	26524	0,10
kroB200	303928	32340	29386	0,10
a280	31874	2974	2579	0,15
d493	430752	43334	35003	0,23
d1291	1682530	222502	50801	3,38

Tabelle I
ERGEBNISSE TSP

Datensatz	Start	End	Best	$\sim$ %-Abweichung
ESC07	3200	2125	2125	0,00
ESC11	3524	2075	2075	0,00
ESC12	2352	1675	1675	0,00
br17.10	119	55	55	0,00
ESC25	9477	2881	1681	0,71
ESC47	17911	3513	1288	1,72
ESC63	228	96	62	0,59
ESC78	30728	19949	18230	0,09
ft70.1	67388	48212	39313	0,22
rgb109a	1889	1262	1038	0,21
rgb174a	2940	2342	2053	0,14
rgb358a	6626	4292	[2518, 2758]	[0,7 - 0,54]

Tabelle II
ERGEBNISSE SOP

zunimmt, dies liegt am *Repair-Schritt*, und die Güte des Ergebnisses nicht eingeschätzt werden kann, eignet sich der Cuckoo's Algorithmus nicht gut für das SOP.

VI. FAZIT

Das Konzept des Cuckoo's Algorithmus hinsichtlich des TSP und SOP ist einfach und schnell verstanden. Durch wenige freie Parameter, die zudem nur einen geringen Einfluss auf den Algorithmus haben, sind nicht viele Anpassungsmöglichkeiten gegeben. Dies hat den Vorteil, dass nach Implementierung nur wenig Zeit aufgebracht werden muss, bis die ersten guten Ergebnisse vorhanden sind. Deshalb lässt sich sagen, dass der Cuckoo's Algorithmus durch seine geringe Komplexität, Laufzeit und Robustheit gegen Änderungen, überzeugt.

Gleichzeitig ist die Stärke des Algorithmus aber auch seine Schwäche. Durch den Verzicht auf komplexe Metriken, sind Anpassung und damit Einwirkungen auf den Ablauf des Algorithmus schwierig. Der Algorithmus lässt sich daher, wenn keine guten Ergebnisse erzielt werden, nur schwer auf das Problem weiter anpassen. Die fehlenden Metriken und damit verbunden die Einfachheit sorgen dafür, dass Erweiterungen wie das SOP mit seinen Bedingungen, sich nicht elegant in das Verfahren einbauen lassen.

Generell eignet sich der Cuckoo's Algorithmus daher in Bereichen, wo schnell für kleiner Optimierungsproblem ein relativ gutes Ergebnis gefunden werden muss. Für perfekte Resultate oder größere/komplexere Probleme ist die Anpassungsmöglichkeit nicht gegeben, weshalb auf komplexere Algorithmen zurückgegriffen werden sollte.

LITERATUR

[1] Yang, X.Ss, Deb, S.: Cuckoo search via lévy flights. In: Nature and biologically inspired computing, 2009. NaBIC 2009. World congress on, IEEE, pp 210–214, (2009) .

[2] Lawler, E.L., Lenstra, J.K., Kan, A.R., Shmoys, D.B.: The traveling salesman problem: a guided tour of combinatorial optimization, vol 3. Wiley, New York (1985) .

[3] Gutin, G., Punnen, A.P.: The traveling salesman problem and its variations, vol 12. Springer, New York (2002).

[4] Pavlyukevich, I.: Lévy flights, non-local search and simulated annealing, J. Computational Physics, 226, 1830-1844 (2007).

[5] Pavlyukevich, I.: Cooling down Lévy flights, J. Phys. A:Math. Theor., 40, 12299-12313 (2007).

[6] Reynolds, A.M., Frye, M.A.: Free-flight odor tracking in Drosophila is consistent with an optimal intermittent scale-free search, PLoS One, 2, e354 (2007).

[7] Yang, X.S.: Cuckoo Search and Firefly Algorithm: Theory and Applications, chap. Cuckoo Search and Firefly: Overview and Analysis, pp. 1-26. Springer Publishing Company, Incorporated (2013).

[8] Leccardi, M.: Comparison of three algorithms for Lévy noise generation. In: Proceedings of Fifth EUROMECH Nonlinear Dynamics Conference, Mini Symposium on Fractional Derivates and their Applications (2005).

[9] Quaarab, A., Ahiod, B., Yang, X.S.: Discrete cuckoo search algorithm for the travelling salesman problem. Neural Comput and Applic (2014) 24:1659–1669, DOI 10.1007/s00521-013-1402-2.

[10] Croes, G.A.: A Method for Solving Traveling-Salesman Problems. Operation Research, 6, 791-812, (1958).

[11] Martin, O., Otto, S.W., Felten, E.W.: Large-Step Markov Chains for the traveling salesman problem. Complex Syst. 5(3), 299-326 (1991).

[12] https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/ Zuletzt aufgerufen: 30.01.2019