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

st 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 haben wir den 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 erzeugt ein Bewegungsmuster wie es bei einigen Vögeln und Insekten beobachtet wurde. 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, haben wir 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

This document is a model and instructions for LaTeX. Please observe the conference page limits.

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 sein 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.

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

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 haben dadurch den Vorteil, dass sie sich nicht um die zeitintensive Aufzucht der Nachkommen kümmern müssen. Sie verringern dadurch ihren Aufwand für die Brutpflege und haben mehr Zeit Nahrung für sich selbst zu finden. Sie sind dadurch 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 der Kuckuck als erster schlüpft und er dann die anderen Eier aus dem Nest schmeißt. 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
```

Um eine neue Generation von Lösungen von einem Kuckuck zu erzeugen, wird der Levy Flight genutzt. In verschiedenen Studien wurde gezeigt, dass das Flugverhalten verschiedener Tiere und Insekten die Eigenschaft eines Levy Flights besitzen [4], [5], [6]. Die Definition des Levy 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 Levy Flights beginnend im Punkt (0,0).

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

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

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 Lèvy Flight approximiert. In [8]

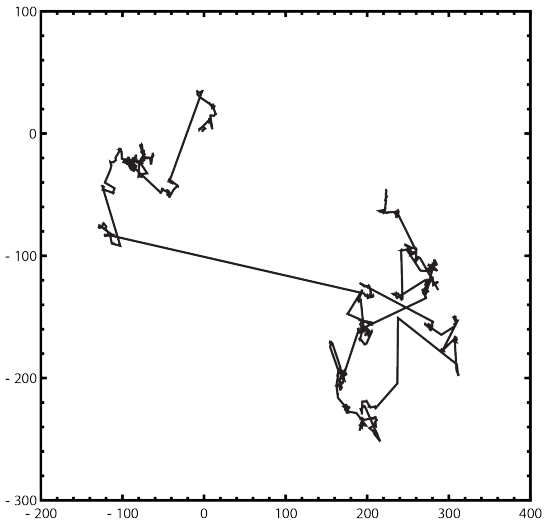

Abbildung 1. Levy Flight beginnend im Punkt (0,0).

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] beschreiben 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, führen ein Teil der Kuckucke in dem verbesserten CS 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

Der größte Unterschied für den Algorithmus von TSP zu SOP ist, dass der Algorithmus Lösungen produzieren kann die nicht mehr valide sind. Beim TSP musste nur darauf geachtet werden, dass jede zufällig erzeugte Lösung zu beginn valide war, danach konnten durch die Operationen DoubleBrideMove und TwoOptSwap keine invaliden Ergebnisse produziert werden. Mit SOP reicht es nicht mehr nur die Initialisierung valide zu gestalten, da auch die beiden Operationen zu invaliden Ergebnissen führen können. Es gibt mehrere Varianten mit invaliden Ergebnisse um zu gehen, die drei Bekanntesten sind dabei das Handicap, Reparieren und Operatoren Anpassung.

A. Handicap

Beim Handicap werden invalide Ergebnisse beibehalten in der Population. Ihre Fitness hingegen wird mittels eines Handicaps verschlechtert. Das Abschwächen ist also sogesehen die ”Bestrafung” dafür das es sich bei der Lösung um ein invalides Ergebnis handelt. Der Vorteil dieser Lösung ist, dass auch eine solche Lösung noch gute Bestandteile beinhaltet die vom Schwarm übernommen werden können. Hätte sie keine guten Bestandteile würde sich durch das Handicap so schlecht werden, dass sie aus der Population ausscheidet. Nachteil an diesem Ansatz ist, dass das Balancing des Handicaps enorm schwer ist. Ist das Handicap viel zu stark, werden invalide 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 invalides Ergebnis ist, was wiederrum auch nicht gut 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, da bedeutet, dass eine Fitness von 100 in dem einen Fall gut und im anderen Fall eine Fitness von 10.000 gut ist. Das Handicap müsste in diesem Fall variable gestaltet werden. Problem hierbei ist, dass das optimale Ergebnis zu beginn nicht bekannt ist, wobei ein relatives Handicap schwer umzusetzen ist. Aus dem Balancing Problem und dem Problem das Handicap auf SOP/TSP anzuwenden, haben wir uns gegen diese Methode entschieden.

B. Operatoren

Mit der Anpassung der Operatoren ist gemeint, dass die Operatoren nur valide Ergebnisse erzeugen können. Im Falle des Cuckoo´s Algorithmus ist dies auch extrem schwer, da die Operationen an sich sehr einfach sind. Es werden keine weiteren Metriken verwendet, mit denen man eine solche Anpassung umsetzen könnte. Um den Algorithmus und damit die Operatoren nicht zu weit weg von der Idee eines einfachen Optimierungsalgorithmus zu bringen, haben wir uns auch gegen diesen Ansatz entschieden.

C. 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 soviel geändert wird, dass die Reihenfolge wieder valide ist. Veränderungen darüber hinaus, könnte dazu führen, dass kleiner Veränderungen durch die Operaten (Swap, CrossBridge) sonst verloren gehen würden. Des weiteren ist der Repair-Schritt ein unnatürlicher Eingriff in die natürliche Lösungsfindung des Cuckoo's Algorithmus, weswegen dieser möglichst klein Ausfallen sollte. Der Rapair-Schritt schien uns für den Cukoo's Algorthimus am passensten, da dieser die Komplexität des Algorithmus nicht erhöht, aber gleichzeitig keine **Balancing** Probleme mit sich bringt. Die genaue Implementierung des Schritte wird im folgenden Abschnitt IV-D beschrieben.

D. Cuckoo Search für das Traveling Salesman Problem

Bekanntlich befindet sich das TSP in einem kombinatorischen Raum und 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.

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

Da die Position der Städte fest ist und diese somit nicht verändert werden können, lassen 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 der Lösung zu machen und der Double-Bride-move wird für große Änderungen genutzt. Wie in Abbildung 2 zu sehen, entfernt der 2-opt-move zwei Kanten und verbindet die zwei Pfade neu. Der Double-Bride-move entfernt vier Kanten und erstellt neue Kanten, so wie in Abbildung 3 zu sehen ist.

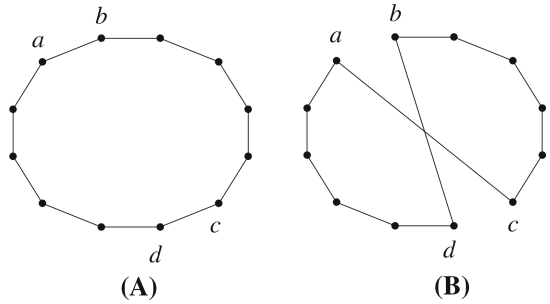

Abbildung 2. 2-opt-move. a initiale Tour. b durch 2-opt-move erzeugte Tour. Kanten (a,b) und (d,c) wurden durch Kanten (a,c) und (b,d) ersetzt

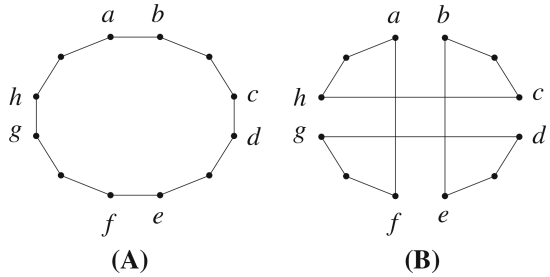

Abbildung 3. Double-Bridge-move. a initiale Tour. b durch Double-Bridge-move erzeugte Tour. Kanten (a,b), (c,d), (e,f) und (g,h) wurden durch Kanten (a,f), (g,d), (e,b) und (g,d) ersetzt

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 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

E. Cuckoo Search für das Sequential Ordering Problem

Im Cuckoo's Algorithmus, muss an drei Orten festgestellt werden, dass nur valide Ergebnisse erstellt werden. Die einzigen Punkte im Algorithmus wo unvalide Ergebnisse auftretten, ist bei der Initialisierung und den beiden Operatoren. Da die **Initialisierung** TSP Reihenfolgen erstellt, werden diese im Nachhinein so transformiert, dass sie dem SOP entsprechen. Beim **2-Opt-Swap** können invalide Reihenfolgen erstellt werden, müssen aber nicht. Daher muss auch hier nach jeder Operation geprüft werden. Ausnahme ist hierbei, dass wenn mehrere **2-Opt-Swaps** hintereinander ausgeführt werden, nur das letzte Ergebnis wieder zu einem validen Transformiert wird. Dies folgt wieder dem Prinzip, dass nur möglichst wenig verändert/eingegriffen werden soll. Beim **DoubleBridgeMove** entstehen immer invalide Ergebnisse, dies ist aus der Definition der Operation ersichtlich. Daher wird hier nach jeder Durchführung die Reihenfolge wieder repariert. Der

Ablauf des Repair-Schrittes ist dabei immer der gleiche und wird im Pseudocode Algorithmus **??** dargestellt.

Algorithm 2 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
```

Die Laufzeit von **??** beträgt dabei O(n^2), wobei n die Länge der Reihenfolge R ist. Dies ergibt sich darauß, 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_1 mit jedem nachfolgendem Knoten n_2 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 dieses Abhängigkeit, die zuvor nicht bestand, wird die Auswertungen von großen SOP's erheblich langsamer.

V. Ergebnisse

Datensatz	Start	End	Best	Abweichung
berlin52.tsp	31757	2920	2670	0,08
ch130.tsp	31757	2920	2670	0,08
a280.tsp	31757	2920	2670	0,08
d1291.tsp	31757	2920	2670	0,08

Tabelle I
Ergebnisse TSP

Datensatz	Start	End	Best	Abweichung
br17.10	31757	2920	2670	0,08
ESC25	31757	2920	2670	0,08
ESC63	31757	2920	2670	0,08
rgb174a	31757	2920	2670	0,08
rgb358a	31757	2920	2670	0,08

Tabelle II
Ergebnisse SOP

VI. Fazit

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).