Master-Projekt

Nicolai Böttger und Jon-Steven Streller
Hochschule Hannover
Fakultät IV - Abteilung Informatik
Masterstudiengang Angewandte Informatik (MIN)

***Zusammenfassung*—Diese Ausarbeitung beschäftigt sich mit dem Metaheuristikalgorithmus Whale Optimization Algorithm zum Lösen von Optimierungsproblemen. Dieser basiert auf dem natürlichen Jagdverhalten von Buckelwalen und besteht aus mehreren Phasen, wie dem Einkreisen, Annähern und Suchen nach Beute. Es handelt sich um einen iterativen Algorithmus, bei dem die Populationsgröße der Wale (Suchagenten) je nach Problem angepasst werden kann.**

Neben dem Whale Optimization Algorithm werden die Optimierungsprobleme Traveling Salesman Problem (TSP) und Sequential Ordering Problem (SOP) erklärt. Fortführend werden einige Änderungen am Whale Optimization Algorithm vorgenommen, sodass dieser TSP und SOP Probleme optimieren kann. Dafür repräsentiert ein Suchagent nicht mehr eine Position im Suchraum, sondern eine valide TSP bzw. SOP Route, bei welcher iterativ durch die vorher genannten Phasen Städte getauscht werden.

Mittels der Programmiersprache Java wurde der theoretische Algorithmus programmatisch umgesetzt und auf einige direkte Benchmark-Probleme angewandt. Abschließend werden noch einige Optimierungsansätze, wie eine Greedy-Heuristik und eine dynamische Iterationszahl, die zu einer besseren Laufzeit führen erläutert.

I. Einleitung

Optimierungsalgorithmen sind sehr bedeutend für viele wissenschaftliche Bereiche von Ingenieurwissenschaften bis zur Datenanalyse. Eine relativ neue und gleichzeitig vielversprechende Möglichkeit zum performanten Lösen von Optimierungsproblemen sind natürlich inspirierte Algorithmen, sogenannte Metaheuristiken. Diese setzen auf natürliche Vorbilder, wie Tierverhalten oder biologische Prozesse als Ansätze für Optimierungsalgorithmen. Ein solcher Algorithmus ist der ”Whale Optimization Algorithm”(WOA), welcher auf dem Verhalten von Buckelwalen bei der Nahrungssuche basiert [1].

In dieser Ausarbeitung wird der WOA-Algorithmus ausführlich und mit Bezug auf das Traveling Salesman Problem”(TSP), welches ein bekanntes Optimierungsproblem darstellt, erklärt. Dafür wird zuerst das TSP/SOP und anschließend der Whale Optimization Algorithm Stück für Stück erläutert. Anschließend wird verdeutlicht, wie der WOA genutzt werden kann, um das Traveling Salesman Problem zu lösen. Zur Bewertung des Algorithmus für den Anwendungsfall des TSP werden im 5. Kapitel experimentelle Ergebnisse für Benchmark-Probleme vorgestellt.

Ausarbeitung im Rahmen des Masterprojekts ”Wettbewerb neuer naturinspirierter Metaheuristiken” an der Hochschule Hannover im WS 23/24

Die Betrachtung des Algorithmus fand im Rahmen des Masterprojekts **Wettbewerb von Metaheuristiken** statt, in welchem mehrere Kleingruppen verschiedene Metaheuristiken bearbeitet haben.

II. Traveling Salesman Problem und Sequential Ordering Problem

Bei dem *Traveling Salesman Problem* sowie dem *Sequential Ordering Problem* handelt es sich um ein Optimierungsproblem. Optimierungsprobleme beschreiben eine mathematische Aufgabe, deren Ziel es ist, einen optimalen Wert zu ermitteln. In vielen Fällen handelt es sich bei dem optimalen Wert einer Funktion um das globale Minimum oder Maximum.

Das Bestimmen einer gültigen Lösung ist relativ einfach und ohne großen Rechenaufwand möglich, während Optimierungsprobleme immer rechenintensiver sind und oft abgewogen werden muss, ob die Berechnung des globalen Optimums den Zeitaufwand wert ist. Aus diesem Grund gibt es für Optimierungsprobleme verschiedene Approximationsalgorithmen, die sich dem globalen Optimum annähern, aber nicht immer das globale Optimum erreichen, dafür aber weniger rechen- und zeitintensiv sind.

A. Travelling Salesman Problem

Das *Traveling Salesman Problem* (TSP) ist wohl das meist verbreitete Optimierungsproblem. Das Problem lässt sich wie folgt beschreiben: Ein Handlungsreisender (Traveling Salesman) plant eine Tour durch beliebig viele Städte. Der Ausgangs- und Endpunkt der Tour sind identisch. Die übrigen Städte dürfen nur einmal besucht werden. Nun besteht das Optimierungsproblem darin, die Kosten, die durch die Tour entstehen, möglichst gering zu halten. Die Kosten können in verschiedenen Formen auftreten, in den meisten Fällen in Bezug auf die zurückgelegte Strecke. In anderen Szenarien könnte der Handlungsreisender entscheiden, mit dem Zug zu reisen und zu versuchen, so wenig wie möglich für Zugtickets zu bezahlen. [2] *TSP* tritt in der Praxis in vielen Anwendungsbereichen auf. Hierzu gehören beispielsweise Logistikunternehmen, die eine optimale Route für den Transportweg planen müssen.

TSP ist ein NP-schweres Problem. [2] Es handelt sich hierbei um eine Klasse von Problemen, die nur in nichtdeterministischer polynomieller Zeit gelöst werden können und für die

eine polynomielle Reduktion von einem bereits bekannten NP-vollständigen Problem erfolgen kann. Dies bedeutet, dass es unwahrscheinlich ist, dass das Problem auf effiziente Weise gelöst werden kann. [3]

Brute-Force Methode

Der intuitive Ansatz, um ein TSP zu lösen wäre, jede Stadt mit jeder anderen Stadt zu vergleichen, sich die Kosten zu merken und anschließend zu überprüfen, welche der Routen die niedrigsten Kosten verursacht. So einen Ansatz nennt man Brute-Force und erzielt die bestmögliche Lösung. Allerdings gehört dieser Ansatz zur Komplexitätsklasse $\mathcal{O}((n-1)!)$. [2]

Ein Beispiel: Angenommen, ein Handlungsreisender möchte vier ($n = 4$) Städte besuchen. Wie in Abbildung 1 ersichtlich, startet der Geschäftsreisende seine Reise am Knoten A und beendet sie auch am Knoten A. $(4-1)! = 3! = 6$ mögliche Routen, die der Handlungsreisende hinsichtlich der damit verbundenen Kosten zu eruieren hat. Auf den ersten Blick wirkt dies noch überschaubar, aber bereits bei einer größeren Menge von Städten, die besucht werden, steigt der Aufwand des Brute-Force-Algorithmus immens an. Unter der Annahme, dass eine CPU pro Sekunde 100.000 Rundreisen überprüfen kann, dauert die vollständige Berechnung bei einer Städteanzahl von $n = 15$ annähernd 25 Jahre.

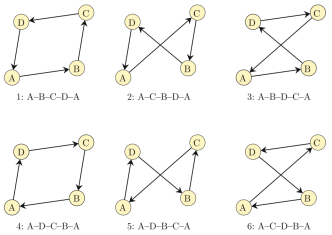


Abbildung 1. Alle möglichen Routen für eine Stadtanzahl von 4 und einem Start- und Endpunkt in Stadt A. *(Quelle: [2], S. 414)*

Es sollte nun nachvollziehbar sein, warum eine optimale Lösung nicht in angemessener Zeit gefunden werden kann und warum bis heute nach alternativen Lösungsansätzen, wie z.B. der Lösung eines TSP in Verbindung mit Metaheuristiken, gesucht wird.

B. Sequential Ordering Problem

Das *Sequential Ordering Problem* (SOP) kann in verschiedenen Arten und Ausprägungen auftreten. Ein SOP kann dazu beitragen, in sich geschlossene Abläufe optimal zu ordnen und somit den Zeitaufwand zu minimieren. Als kleines Beispiel im Industriekontext könnte es sein, dass eine Autotür hergestellt wird. Die Arbeitsabläufe können in unterschiedlicher Reihenfolge ausgeführt werden, um im Idealfall den Arbeitsaufwand zu reduzieren. Ein Negativbeispiel im Kontext der Autotür ist, dass zuerst die Dichtungen angebracht werden und danach Schweißarbeiten durchgeführt werden, wofür vorübergehend die Dichtungen wieder entfernt werden müssen. Hier wäre ein ausführen in umgekehrter Reihenfolge optimaler.

In dieser Arbeit beschäftigen wir uns ausschließlich mit einer anderen Ausprägung des SOP. Es wird eine optimale Lösung für ein asymmetrisches TSP gesucht. Das bedeutet, dass die Entfernung $A \to B \neq B \to A$ sein kann.

III. Grundlagen des "Whale Optimization Algorithm"

Bei dem *Whale Optimization Algorithm* handelt es sich um eine natur-inspirierte Metaheuristik. Metaheuristiken versuchen, die Lösung eines gegebenen Optimierungsproblems schrittweise zu verbessern, indem sie die bisher beste Lösung iterativ in vordefinierter Weise verändern. Die meisten dieser Algorithmen implementieren zusätzlich eine globale Suche, um nicht in lokalen Optima stecken zu bleiben. [4]

In diesem Kapitel wird ausführlich der WOA erläutert. Dafür wird zuerst das Jagdverhalten von Walen bzw. Buckelwalen in der Natur betrachtet. Anschließend wird beleuchtet, wie dieses auf einen mathematischen Algorithmus (WOA) abgebildet werden kann.

A. Natürliches Jagdverhalten von Buckelwalen

Das besondere Jagdverhalten der Buckwale ist die Hauptinspiration für die Entwicklung des Whale Optimization Algorithm. Bevor die Wale mit der eigentlichen Jagd beginnen, suchen sie ihre Beute, welche meist aus kleinen Fischen und Krill besteht, mittels Echoortung. Dies funktioniert so, dass die Buckelwale Schallwellen aussenden, welche dann anschließend an den Kleintieren abprallt, und ihnen so Informationen über die Entfernung und Größe derer liefert. [5]

Haben sie auf diese Art Beute gefunden, beginnen die Wale damit die Beute zu umkreisen. Dies tun sie meist in Gruppen, indem sie zuerst unter den Fischschwarm tauchen und dann in einer Spiralbewegung um diesen herum schwimmen. Dabei stoßen sie Luftblasen aus, um Ringe mit einem Durchmesser von bis zu 45 Metern zu verursachen, welche die Fische umzingeln und wie ein Gefängnis fungieren. [6]. Währenddessen bewegen sie sich hin zur Meeresoberfläche und gleichzeitig verringern sie den Durchmesser des "Blasen-Rings", indem sie kleinere Runden um die Beute schwimmen und treiben sie so immer enger zusammen. Zum Schluss öffnen sie nur noch ihre Mäuler und tauchen auf, um die Fische zu schlucken. [7]

B. Algorithmische Abbildung des Jagdverhaltens

Der von *Mirjalili* und *Lewis* entwickelte Algorithmus [1] basiert auf dem Prinzip des Umzingelns der Beute. Das Aufsuchen der Beute mittels Echoortung und Kreieren des Blasennetzes wird hierbei impliziert und nicht weiter algorithmisch behandelt.

Der Algorithmus funktioniert, indem iterativ die Position der Suchagenten (Wale) aktualisiert wird. Hierbei existieren, wie bei Metaheuristiken üblich, die 2 Phasen **Exploitation** und **Exploration**. Erstere setzt die Suche eines lokalen Maximums um und letztere sorgt für eine Beachtung globaler Ergebnisse.

Der Algorithmus definiert 2 Bewegungsmuster der Suchagenten. Bei dem ersten bewegen sie sich gerade auf ihr Ziel zu, während sie sich bei dem zweiten in einer Kreisbewegung um

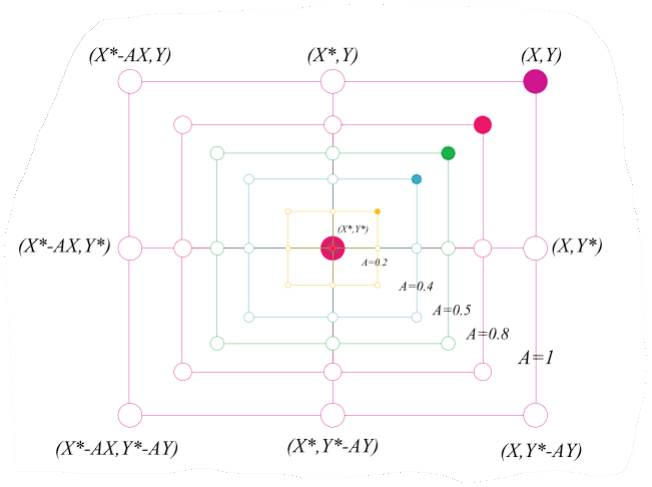


Abbildung 2. Umkreisungs-Mechanismus des Whale Optimization Algorithm *(Quelle: [1], Fig. 4)*

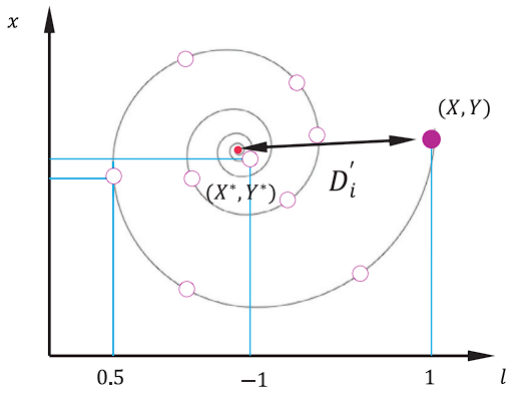


Abbildung 3. Umkreisungs-Mechanismus des Whale Optimization Algorithm *(Quelle: [1], Fig. 4)*

die Beute herum bewegen.

Alle in den Formeln dargestellten Vektoren haben, außer explizit anders angegeben, die gleiche Dimensionalität wie der Positionsvektor $\overrightarrow{X}$, welcher die Position eines Suchagenten im Suchraum darstellt.

1) Umkreisungs-Mechanismus (Exploitation): Da die Position des Ziels bzw. der Beute nicht bekannt ist, wird angenommen, dass der "besteSSuchagent in jeder Iteration das Ziel, welches abhängig vom Optimierungsproblem ist, ist. Auf Basis dessen wird die Position der anderen Suchagenten aktualisiert.

$$\overrightarrow{X}(t+1) = \overrightarrow{X}_{best}(t) - \overrightarrow{A} * \overrightarrow{D} \quad (1)$$

$$\overrightarrow{D} = |\overrightarrow{C}.\overrightarrow{X}_{best}(t) - \overrightarrow{X}(t)| \quad (2)$$

- $\overrightarrow{X}$ ist der Positionsvektor eines Suchagenten, z.B. (X,Y) für 2D-Suchraum
- $\overrightarrow{X}_{best}$ ist der Positionsvektor des momentan besten Suchagenten
- $\overrightarrow{A} = 2\overrightarrow{a} * \overrightarrow{r} - \overrightarrow{a}$
- $\overrightarrow{C} = 2 * r$
- $\overrightarrow{a} = [0, 2]$ und wird pro Iteration linear verringert
- $\overrightarrow{r} = [0, 1]$ und ist ein Zufallsvektor
- $\cdot$ ist die element-weise Multiplikation

Die Auswirkungen von (1) sind in III-B1 dargestellt. Durch $\overrightarrow{D}$ wird die Entfernung zwischen Suchagent und dem Ziel berechnet. Mittels $\overrightarrow{A}$ wird bestimmt, wie stark sich der jeweilige Suchagent auf den besten Suchagenten zubewegt. Da sich $\overrightarrow{a}$ und daraus resultierend auch $\overrightarrow{A}$ mit jeder Iteration verringert und $\overrightarrow{A}$ mit $\overrightarrow{D}$ multipliziert wird, hat $\overrightarrow{D}$ einen immer geringeren Einfluss auf das Endergebnis und die Suchagenten nähern sich immer weiter dem besten Suchagenten an.

2) Spiralförmige Positionsaktualisierung (Exploitation):

$$\overrightarrow{X}(t+1) = \overrightarrow{D'} * e^{bl} * cos(2\pi l) + \overrightarrow{X}_{best}(t) \quad (3)$$

$$\overrightarrow{D'} = |\overrightarrow{X}_{best}(t) - \overrightarrow{X}(t)| \quad (4)$$

- b ist Konstante zum Festlegen der Spiralgröße
- $l = [-1, 1]$ und ist eine Zufallszahl

Abbildung 3 verdeutlicht, wie sich die Suchagenten durch Anwendung dieser Funktion nicht gerade sondern mit der in der Natur zu beobachtenden Spiralbewegung um das Ziel herum bewegen. Auch hier ist $\overrightarrow{D}$ die Distanz zwischen dem betrachteten und dem besten Suchagenten.

In der Natur führen die Wale beide Bewegungen gleichzeitig aus. Algorithmisch wird dies umgesetzt, indem in jeder Iteration zufällig bestimmt wird, welche Bewegung ausgeführt wird. Die endgültige Positionsaktualisierung ist definiert durch

$$X(t+1) = \begin{cases} \overrightarrow{X}_{best}(t) - \overrightarrow{A} \cdot \overrightarrow{D} & p < 0.5 \\ \overrightarrow{D'} \cdot e^{bl} \cdot cos(2\pi l) + \overrightarrow{X}_{best}(t) & p \geq 0.5 \end{cases} \quad (5)$$

- $p = [0, 1]$ und ist eine Zufallszahl

3) Beutesuche (Exploration): Die globale Suche funktioniert sehr ähnlich wie der *Umkreisungs-Mechanismus*, nur mit dem Unterschied, dass sich die Suchagenten voneinander entfernen. Dabei wird als Referenz-Suchagent nicht der beste, sondern ein zufälliger gewählt.

$$\overrightarrow{X}(t+1) = \overrightarrow{X}_{rand} - \overrightarrow{A} \cdot \overrightarrow{D} \quad (6)$$

$$\overrightarrow{D} = |\overrightarrow{C}.\overrightarrow{X}_{rand} - \overrightarrow{X}| \quad (7)$$

Abhängig von $|A|$ wird entweder die *Beutesuche* ($|A| \geq 1$) oder der *Umkreisungsmechanismus* ($|A| < 1$) durchgeführt.

4) Kompletter Algorithmus: Zu Anfang wird eine voreingestellte Menge Suchagenten initialisiert und zusätzlich deren Fitness berechnet, also wie nah die Lösung des jeweiligen Agenten an der Optimallösung des Problems ist. In einer Iterationsschleife werden anschließend in jeder Iteration (a, A, C, l, p) aktualisiert und abhängig von p und $|A|$ durchläuft jeder Suchagent einen der Aktualisierungspfade. Es kann vorkommen, dass Suchagenten durch die Aktualisierung den definierten Suchraum verlassen, weswegen sie im nächsten Schritt, falls dies der Fall war, angepasst werden müssen, z.B. durch Eingrenzen der Werte. Zuletzt werden wieder die

Fitnesswerte berechnet.

Dieser Ablauf wird solange wiederholt, bis die maximale Anzahl Iterationen erreicht ist.

IV. Anpassung des WOA für TSP und SOP

Damit der Whale Optimization Algorithm auf ein diskretes Problem, wie das TSP, angewendet werden kann, müssen einige Anpassungen sowohl an dem Positionsvektor X und somit auch den anderen Vektoren, als auch der generellen Struktur der mathematischen Formeln vorgenommen werden. Die Berechnung der einzelnen Vektoren und Zahlen bleibt, wenn nicht weiter angegeben, gleich.

Die angepassten Formeln aus IV-A stammen aus der Ausarbeitung "*Greedy WOA for Travelling Salesman Problem*" [8].

A. WOA für TSP

Anstatt dass der Positionsvektor eines jeden Suchagenten Koordinaten eines Punktes im Suchraum repräsentiert, stellt er nun eine gültige TSP-Route für ein gegebenes TSP dar. Sei n die Anzahl der Städte des TSPs und m die Anzahl der definierten Suchagenten, so kann eine Permutationsmatrix $P_{m,n}$ gebaut werden, welche die Route aller Suchagenten beinhaltet. Nachfolgend wird eine beispielhafte Permutationsmatrix abgebildet.

$$P = \begin{bmatrix} P_{1,1} & P_{1,2} & \dots & P_{1,n} \\ P_{2,1} & P_{2,2} & \dots & \dots \\ \dots & \dots & \dots & \dots \\ P_{m,1} & \dots & \dots & P_{m,n} \end{bmatrix}$$

P_i bezeichnet dabei die (valide) Route des i. Suchagenten und $P_{i,j}$ die j. Stadt des i. Suchagenten. Auf Basis dieser Matrix werden die Formeln aus III-B modifiziert. Anstatt dass sich die Suchagenten in einem Koordinatenraum abhängig voneinander bewegen, wird die Aktualisierung beim *WOA* für *TSP* durch Tauschen einer oder mehrerer Städte durchgeführt. Diese Aktualisierung funktioniert durch die Formel (8).

$$P_{i,j}(t+1) = \begin{cases} P_{rand,k}(t) & \text{if } |A| \geq 1 \\ P_{leader,k}(t) & \text{if } |A| < 1 \end{cases} \tag{8}$$

- i ist eine ganze Zahl im Intervall [1, m]
- j, k sind ganze Zahlen im Intervall [1, n]
- t ist eine ganze Zahl im Intervall [0, maxIter], wobei *maxIter* die maximale Anzahl Iterationen ist

Wie auch beim SStandardWOA werden die Suchagenten in den anfänglichen Iterationen ($|A| \geq 1$) abhängig von einem Zufallsagenten und in den späteren ($|A| < 1$) abhängig von dem Suchagenten mit der besten Fitness aktualisiert. Die Variablen k und j sind die Positionen der Städte in der jeweiligen Route eines Suchagenten, welche getauscht werden sollen. Sei $P_1(t) = [3, 5, 2, 4, 1]$ und $j = 2, k = 3$, so würden die Städte mit den Indizes 2 und 5 getauscht werden, da sie an diesen Positionen liegen, sodass $P_1(t+1) = [3, 2, 5, 4, 1]$.

j ist die Laufvariable, auf Basis derer k mit folgenden Formeln berechnet wird:

$$k = \left| j + \left\lfloor \frac{C}{A} \cdot n \right\rfloor \right| - n \cdot \left\lfloor \frac{\left| j + \left\lfloor \frac{C}{A} \cdot n \right\rfloor \right|}{n} \right\rfloor + 1 \tag{9}$$

$$k = \left\lfloor D_{leader} e^{bl} \cos(2\pi l) + j \right\rfloor + \left\lfloor \frac{D_{leader} e^{bl} \cos(2\pi l) + j}{n} \right\rfloor + 1 \tag{10}$$

- $\cdot$ ist elementweise Multiplikation

Diese Formeln sind die diskreten Formen von (1) und (3).

Dies sind alle grundlegenden Änderungen, damit der WOA auf TSPs angewendet werden kann.

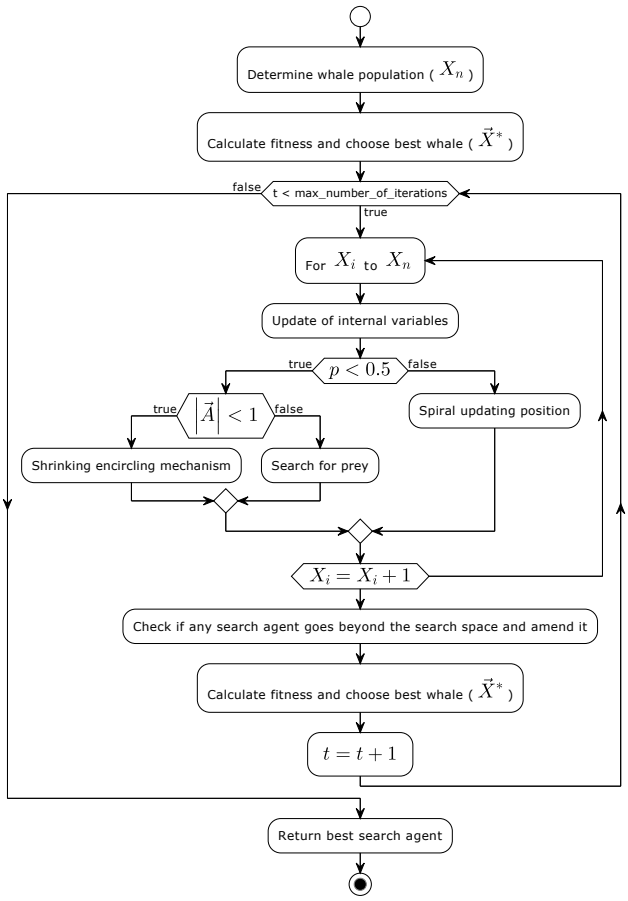


Abbildung 4. Das Aktivitätsdiagramm veranschaulicht den Ablauf des WOA.

B. WOA für SOP

Um den in IV-A beschriebenen WOA so zu adaptieren, dass darin auch SOP-Probleme behandelt werden können, müssen an einigen Stellen des Algorithmus Änderungen vorgenommen werden, die in den folgenden Abschnitten

erläutert werden.

1) Initialisierung der Suchagenten: Bei der Implementierung des SOP-WOA hat jeder Suchagent eine randomisierte Route P_i als Startpunkt. Jeder Knoten $P_{i,j}$ in der Route P_i hat eine Menge V_i an Knoten, die vor dem aktuellen Knoten $P_{i,j}$ besucht sein müssen. Falls die Menge $V_{i,j}$ leer ($V_{i,j} = \emptyset$) ist, deutet dies darauf hin, dass der Knoten $P_{i,j}$ entweder keine Bedingungen hat oder dass alle Bedingungen bereits erfüllt sind. Formal ist die Permutationsmatrix P SOP-Constraint-konform, wenn gilt:

$$\forall i \forall j \in P_{i,j} : |V_{i,j}| = 0$$

Routen, die diesen Bedingungen entsprechen, werden als Constraint-konforme Routen bezeichnet.

Um die zufällige Route in eine konforme Route zu überführen, wird über jede Route eine While-Schleife ausgeführt, die erst stoppt, wenn die Route konform ist. Der folgende Pseudocode veranschaulicht das Vorgehen:

Algorithm 1 Überführung einer randomisierten Route zu einer Constraint-konformen Route.

```
for i ← 0 to |P| do
    while not IsConstraintCompliant(P_i) do
        for j ← 0 to |P_i| do
            for node in V_{i,j} do
                if index(node) > i then
                    Swap(j, index(node))
                end if
            end for
        end for
    end while
end for
```

Abbildung 5 zeigt, wie die Route P_i (in der Abbildung als *Tour* bezeichnet) mit dem Algorithmus 1 so verändert wurde, um eine Constraint-konforme Route zu erhalten. In der Abbildung 5 wird der Knoten $P_{i,1}$ betrachtet. Im ersten Schritt ist der Constraint Stack $V_{i,1}$ noch mit Bedingungen gefüllt, die nach und nach in den darauf folgenden Schritten aufgelöst werden, bis $V_{i,1}$ schließlich $V_{i,1} = \emptyset$ die leere Menge ist und somit per Definition als Constraint-konforme Route gilt.

2) Modifikation der Aktualisierung der Routen: Die Route P_i verändert sich an insgesamt zwei Stellen und muss an diesen auf Contraint-Konformität überprüft werden. Bei der Initialisierung wird zufällig eine Route P_i gewählt, was bereits in IV-B1 behandelt wurde. Um die Konformität der Route P_i bei der Stadttausch-Berechnung des WOAs sicherzustellen muss die Formel 8 entsprechend angepasst werden. Diese Anpassungen sind nachfolgend dargestellt:

$$P_{i,j}(t+1) = \begin{cases} P_{r,j}(t) & \text{wenn } |A| \geq 1 \ \ \& \ \ V_{r,j} = \emptyset \\ P_{l,j}(t) & \text{wenn } |A| < 1 \ \ \& \ \ V_{l,j} = \emptyset \end{cases} \quad (11)$$

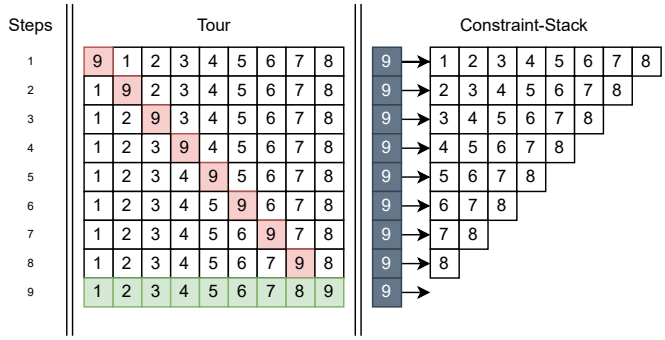


Abbildung 5. Die Abbildung zeigt, wie eine Route P_i in eine gültige, constraint-konforme Route überführt wird.

Hierbei steht r für einen zufällig gewählten Suchagenten und l für den in der Iteration t besten Suchagenten.

Durch die Anwendung der angepassten Formel 11 wird die Route nur dann angepasst, wenn keine Constraints verletzt werden. Dadurch wird sichergestellt, dass die Route jederzeit gültig und konform zu den Constraints ist.

V. Verbesserungen des Basisalgorithmus

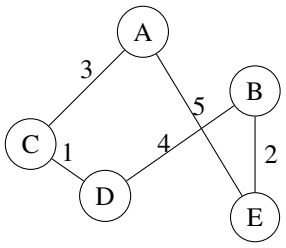
Die zuvor vorgestellten Algorithmen für TSP und SOP sind vollständig und liefern ein iterativ optimiertes Ergebnis für beide Probleme.

In diesem Abschnitt werden einige Verbesserungen für den Basisalgorithmus WOA für TSP bzw. SOP vorgestellt und erläutert. Diese Änderungen beeinflussen nicht den grundlegenden Ablauf des Algorithmus, sondern setzen nur an einigen Stellen an, um ein besseres Ergebnis zu erzielen.

A. Greedy Swap Technique

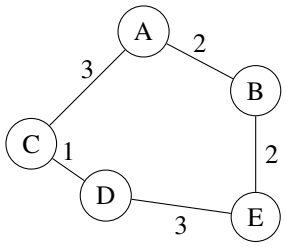
Der erste Ansatz ist die ”Greedy Swap Technique”. Hierbei handelt es sich um einen *greedy* Algorithmus, welcher also die lokal beste Option in der jetzigen Iteration auswählt und zukünftige Iterationen dabei nicht betrachtet.

Greedy Swap führt dazu, dass ein Tausch von Städten in einer Route eines Suchagenten nur durchgeführt wird, wenn das Ergebnis nach dem Tausch zu einer besseren Fitness des Suchagenten führt.



Sei der Graph eine valide TSP-Route eines Suchagenten mit Stadt A als Startpunkt. Angenommen für $j = 2$ wurde mittels Eq. (9) $k = 3$ berechnet, sodass Stadt B mit Stadt E getauscht werden soll. In der Praxis bedeutet das, dass zuerst B und danach E besucht wird. Durch *Greedy Swap* wird an dieser Stelle nun erstmal geprüft, ob sich dies positiv auf die

Fitness der Gesamtroute auswirkt. Dafür wird die Entfernung zwischen den Nachbarn der beiden Städte vor und nach dem Tausch berechnet. Zur Veraunschaulichung wird der Graph nun nach dem potentiellen Tausch dargestellt.



Die Entfernung vor dem Tausch ist $(E,A)+(E,B)+(B,E)+(B,D)$ $=>$ $5+2+2+4=13$ und nach dem Tausch $(B,A)+(B,E)+(E,B)+(E,D) => 2+2+2+3=9$. Da die kumulierte Entfernung nach dem Tausch geringer ist, wird dieser durchgeführt.

Die Hauptauswirkung von *Greedy Swap* ist, dass sehr viel weniger Tauschoperationen ausgeführt werden müssen. Stattdessen müssen arithmetische Berechnungen durchgeführt werden, welche allerdings in der Regel weniger rechenintensiv sind. Allerdings kann *Greedy Swap* auch potentiell gute zukünftige Tauschmöglichkeiten verhindern, da es immer nur den derzeitigen Stand betrachtet.

In der Praxis hat *Greedy Swap* zu einer besseren Laufzeit und einem ähnlichen Ergebnis geführt, weswegen es beibehalten wird.

B. Dynamische Wahl der Konfigurationsparameter durch generierte Heuristik

Allerdings stellt sich spätestens jetzt die Frage, mit welchen Konfigurationsparametern der Algorithmus gestartet werden muss, um ein optimales Ergebnis hinsichtlich der Laufzeit und der Strecke zu erzielen. Es gibt zwei Variablen, die das Ergebnis maßgeblich bestimmen: zum einen die Anzahl der Iterationen und zum anderen die Anzahl der Suchagenten.

Das Ziel dieser Optimierung besteht darin, für ein beliebiges Problem eine dynamische Wahl der Iterationsanzahl und der Anzahl der Suchagenten zu ermöglichen, da eine statische Wahl nicht für alle Probleme geeignet ist. Bei Problemen mit wenigen Knoten ist das Ergebnis ausreichend, wenn eine niedrige Anzahl von Iterationen und Suchagenten gewählt wird. Allerdings kann der Algorithmus bei einer Route mit vielen Knoten nicht sein volles Potential ausschöpfen. Der Algorithmus würde mitten in der Optimierung beendet werden, obwohl noch bessere Lösungskandidaten gefunden werden könnten. Dasselbe gilt analog für die Wahl größerer Werte für die Anzahl der Iterationen sowie für die Suchagenten. Bei kleinen Problemen würde der Algorithmus unnötig lange laufen, ohne wesentliche Verbesserungen zu erzielen.

Der angewendete Lösungsansatz bestand darin, vorab viele verschiedene Routen mit unterschiedlichen Iterations- und Suchagentenanzahlen zu starten und die Werte in der Matrix H_i zu speichern. Die gespeicherten Werte von H_i können verwendet werden, um für ein beliebiges Problem die optimale Anzahl an Iterationen und Suchagenten zu definieren. Die Matrix H_i ist wie folgt definiert:

$KA =$ Knotenanzahl
$IA =$ Anzahl der Iterationen
$SAA =$ Anzahl der Suchagenten
$AV =$ Absolute Verbesserung
$VPI =$ Verbesserung pro Iteration $= \frac{AV}{IA}$

$$H_i = \begin{pmatrix} KA & IA & SAA & AV & VPI \end{pmatrix}$$

Wie die Matrix H_i befüllt wird, wird im folgenden Pseudocode 2 verdeutlicht: Die Anzahl der Iterationen beginnt bei 200 und wird pro Iteration um 100 erhöht, bis 1000 erreicht werden. Die Anzahl der Suchagenten beginnt bei 20 und wird pro Iteration um 20 erhöht, bis 100 erreicht sind.

Algorithm 2 Generierung von H_i

```
for i ← 0 to GETROUTESCOUNT() step 1 do
    for ia ← 200 to 1000 step 100 do
        for saa ← 20 to 100 step 20 do
            APPENDTOH_i(WOA(Routes[i], IA, SAA))
        end for
    end for
end for
```

Die äußere Schleife (ia) wird insgesamt 9-mal ausgeführt, während die innere (saa) Schleife 5-mal ausgeführt wird. Das bedeutet, dass für jede Route $9 \cdot 5 = 45$ Ergebnisse in der Matrix H_i gespeichert werden.

In H_0 wird exemplarisch dargestellt, wie H_i nach Ausführung des Algorithmus aussieht. Es ist wichtig zu erwähnen, dass im Beispiel von H_0 nur eine Route mit 575 Knoten erkennbar ist. In der realen Implementierung besteht H_0 aus $r \cdot 9 \cdot 5$ Zeilen, wobei r für die Anzahl der Routen steht.

$$H_0 = \begin{pmatrix} 575 & 200 & 20 & 18556 & 91 \\ 575 & 200 & 40 & 23833 & 188 \\ 575 & 200 & 60 & 16187 & 79 \\ 575 & \dots & \dots & \dots & \dots \\ 575 & 1000 & 100 & 17441 & 86 \end{pmatrix}$$

Da der WOA-Algorithmus einen Teil der internen Variablen zufällig wählt, kann es vorkommen, dass zufällig ein sehr gutes Ergebnis erreicht wird. Dieses Ergebnis würde bei erneuter Ausführung unter gleichen Bedingungen jedoch nicht oder nur mit einer niedrigen Wahrscheinlichkeit wieder auftreten. Um diesen Zufall auszugleichen, wird die Berechnung von H_i insgesamt dreimal durchgeführt und der Mittelwert berechnet. Daraus ergibt sich:

$H_M = \frac{H_0+H_1+H_2}{3}$.

Die Werte von H_M werden nun aufsteigend nach der Knotenanzahl (KA) und absteigend nach Verbesserung pro Iteration (VPI) sortiert. Die Matrix H_M wird durch dieses Vorgehen so geordnet, dass die Route mit der geringsten Anzahl an

Knoten KA und der besten VPI an erster Stelle steht. Genau dieser erste Eintrag ist für die Heuristik relevant. Dies gilt entsprechend für jede Route r in H_{MD}.

Anschließend wird der erste Wert von KA von r in die sortierte Matrix H_{MD} eingefügt. Die Matrix H_{MD} sieht nach der Ausführung wie folgt aus:

$$H_{MD} = \begin{pmatrix} 52 & 200 & 20 & 6955 & 31 \\ 99 & 200 & 40 & 2559 & 11 \\ 200 & 200 & 100 & 99138 & 494 \\ 575 & 200 & 40 & 23833 & 188 \\ 657 & 200 & 20 & 99138 & 494 \end{pmatrix}$$

Die Matrix H_{MD} enthält nun nur noch die optimalen Iterations- bzw. Suchagentenanzahlen aus H_M für ein beliebiges Problem, und gibt Aufschluss darüber, welche Konfigurationsparameter gewählt werden müssen, um ein optimales Ergebnis hinsichtlich Laufzeit und Strecke zu erzielen.

Bei einem Problem mit 52 Knoten wird die Iterationsanzahl auf 200 und die Anzahl der Suchagenten auf 20 dynamisch gesetzt.

Da die Matrix H_{MD} aufsteigend nach KA sortiert ist, lässt sich hierauf die Binärsuche ausführen, um die optimalen Konfigurationsparameter zu bestimmen. Die Laufzeit dafür beträgt $O(log \cdot N)$. [2]

C. Dynamische Anpassung der Iterationsanzahl während der Laufzeit

Als weitere Optimierung wurde eine dynamische Anpassung der Anzahl der Iterationen zur Laufzeit eingeführt. Konkret bedeutet diese Optimierung, dass in der letzten Iteration überprüft wird, ob in den letzten 15% der Iterationen eine Verbesserung der Route gefunden wurde. Wenn eine solche gefunden wurde, werden weitere 15% Iterationen durchgeführt.

Hierdurch soll das Verhalten des WOA optimiert werden, da im letzten Bereich der Iterationen die meisten Verbesserungen erfolgen. So wird sichergestellt, dass der Optimierungsprozess nicht vorzeitig abgebrochen wird.

Die 15 Prozent erwiesen sich als guter Kompromiss zwischen Laufzeit und Verbesserung. Bei einer höheren Prozentzahl ($> 15\%$) ist die Anzahl der Iterationen zu stark angestiegen, während bei einer niedrigeren Prozentzahl ($< 15\%$) die Optimierung zu selten ausgeführt wurde.

VI. Experimentelle Ergebnisse für Benchmark-Probleme

Für die Experimente wurden einige Benchmark-Probleme für TSP aus der TSPLIB [9] ausgewählt und der Algorithmus auf diese Probleme angewandt. Die Ergebnisse sind in der Tabelle 6 nachstehend aufgeführt. $\overline{x}$ ist das arithmetische Mittel und σ die Standardabweichung. Der Algorithmus wurde jeweils mit 1000 Iterationen und 50 Suchagenten über 10 unabhängige Läufe berechnet. Es ist zu erkennen, dass der Algorithmus bei kleineren Problemen, d.h. bei Problemen mit weniger Städten (z.B. Bays29), eine größere Verbesserung erzielt als bei Problemen mit vielen Städten.

Datensatz	Knotenanzahl	GWOA			Optimum
		Startwert	Endwert		Distanz
		x̄	x̄	σ	Best
bays29	29	23042,9	**13009,6**	1130,2	2020
eil51	51	1573,6	**970,7**	72,6	426
st70	70	3396,7	**2110,8**	84,5	675
eil101	101	3220,5	**2224,3**	104,6	629

Abbildung 6. Benchmark-Ergebnisse des GWOA

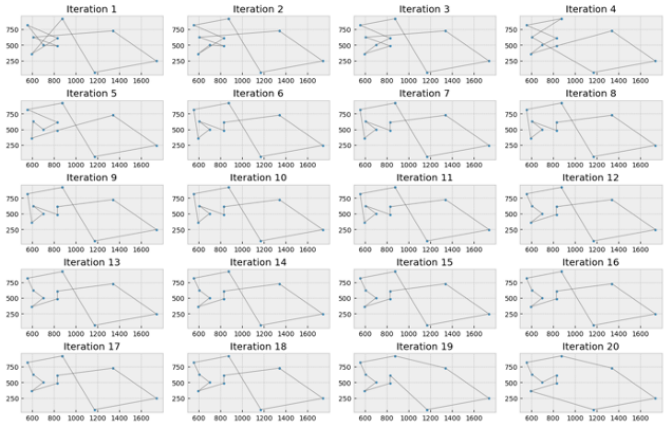


Abbildung 7. Veranschaulichung der besten Route in 20 Iterationen

Abbildung 7 veranschaulicht, wie sich die beste Route im Laufe der Iterationen verbessert. Die Anzahl der Städte beträgt in dem Beispiel 10. Zu erkennen ist, wie sukzessiv Überkreuzungen aufgelöst werden.

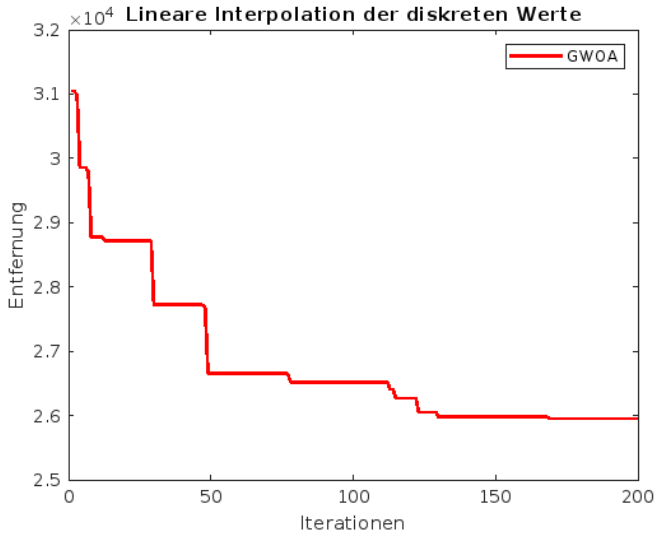


Abbildung 8. Fitness der besten gefundenen Route über 200 Iterationen.

In Abbildung 8 wird der iterative Verlauf zur Optimierung eines TSPs über 200 Iterationen dargestellt. Eine deutliche

Verbesserung, also Verringerung der kumulativen Entfernung zwischen den Städten, ist in den ersten Iterationen erkennbar, welche mit der Zeit jedoch immer geringer wird.

VII. Ausblick und Fazit

Da die Abgabe durch das Modul in unserem Studiengang zeitlich begrenzt war, konnten einige Punkte nicht vertieft behandelt werden, die jedoch vielversprechend sein könnten und daher im Folgenden kurz erläutert werden.

Ein vielversprechender Ansatz wäre die Verwendung einer Heuristik, um die Start-Route zu bestimmen, anstatt sie wie in der Standardimplementierung vollständig zufällig zu wählen. Eine effiziente Heuristik, die hierfür oft genannt wird, ist die k-opt-Heuristik. Die Heuristik könnte dann als approximative Vorberechnung dienen, während (G)WOA dann mit dem Ergebnis der Heuristik fortfährt, um ein besseres Ergebnis zu erzielen.

Eine Möglichkeit, den Algorithmus zur Verarbeitung von SOP zu verbessern, wäre die Erweiterung der Formel 11. Dadurch könnte vermieden werden, dass die Berechnung durchgeführt wird, um anschließend überprüfen zu müssen, ob der Constraint-Stack mit der leeren Menge übereinstimmt. Es wäre sinnvoller, wenn die Berechnung der Formel 11 garantiert zu einer Constraint-konformen Route führt. Dadurch wird die Überprüfung des Constraint-Stacks obsolet. Die Umsetzung gestaltete sich leider sehr komplex und unsere bisherigen Ansätze hatten nicht zum gewünschten Ergebnis geführt.

Der (G)WOA eignet sich gut für Szenarien, in denen nicht unbedingt das optimale Ergebnis erzielt werden muss. Der Algorithmus findet keine optimale Lösung, aber eine Lösung in akzeptabler Laufzeit.

Literatur

[1] S. Mirjalili and A. Lewis, "The whale optimization algorithm," *Advances in Engineering Software*, vol. 95, pp. 51–67, 2016.

[2] B. Voecking, *Taschenbuch der Algorithmen - Kap. Das Travelling Salesman Problem*. Berlin: Springer-Verlag, 1 ed., 2008.

[3] D. W. Hoffmann, *Theoretische Informatik*. München: Hanser, 5 ed., 2022.

[4] Prof. Dr. Marco Lübbecke, "Metaheuristik." [Accessed 20.01.2024].

[5] Liz Langley (National Geographics), "Wie funktioniert echolokation?." [Accessed 14.01.2024].

[6] Pro Wildlife, "Springender akrobat." [Accessed 14.01.2024].

[7] D. W. C. W. A. B. D. C. A. F. M. T. M. Weinrich, "Underwater components of humpback whale bubble-net feeding behaviour," *Brill*, p. 575–602, 2011.

[8] R. Gupta, N. Shrivastava, M. Jain, V. Singh, and A. Rani, *Greedy WOA for Travelling Salesman Problem*. Singapore: Springer Singapore, 2018.

[9] "Universität heidelberg: Tsplib." (accessed: 29.01.2023).