Seminararbeit

GRUNDLAGEN VON METAHEURISTIKEN UND LOKALE SUCHE

Simeon Andreev
Institut für Theoretische Informatik
Karlsruher Institut für Technologie (KIT)

Inhaltsverzeichnis

1 Einleitung **3**
- 1.1 Einführung . . . 3
- 1.2 Motivation . . . 4
- 1.3 Grundlagen von Metaheuristiken . . . 5

2 Untersuchung des Suchraums **7**
- 2.1 Intensifizierung . . . 7
- 2.2 Diversifizierung . . . 7

3 Bausteine von S-Metaheuristiken **9**
- 3.1 Auswahl der Zielfunktion . . . 9
- 3.2 Repräsentation der Lösungen . . . 10
- 3.3 Konstruktion der Startlösung . . . 11
- 3.4 Erstellung der Nachbarschaft . . . 12
- 3.5 Untersuchung der Nachbarschaft . . . 13

4 Anwendung der lokalen Suche **16**
- 4.1 Beispiele . . . 16
- 4.2 Umgang mit lokalen Optima . . . 19

5 Fitness-Landschaften **21**
- 5.1 Eigenschaften der Fitness-Landschaft . . . 22
- 5.2 Untersuchung der Fitness-Landschaft . . . 23

6 Zusammenfassung **25**

7 Literatur **26**

1 Einleitung

Dieses Kapitel bietet einen Einstieg im Thema. Es gibt einen Überblick an, was Metaheuristiken ungefähr sind, wo sie zu verwenden sind und wie eine Metaheuristik aussehen könnte. Dazu ist eine kurze Beschreibung von Metaheuristiken und die Motivation für deren Anwendung gegeben. Einige Anwendungsbeispiele und Grundlagen sind dann vorgestellt.

1.1 Einführung

Heutzutage wird es versucht, alle möglichen Probleme mit Hilfe von Rechner zu lösen. Bei vielen von diesen Problemen gibt es mehrere Lösungen, wobei manche Lösungen besser als andere sind.

Beispielsweise können wir bei der Routenplanung, ausgehend von dem Ort A, den Ort B mit Hilfe von verschiedenen Wegen erreichen. Wir wollen aber den schnellsten oder kürzesten Weg. Also ist hier unseres Problem einen Weg von A nach B zu finden, eine Lösung ist so ein Weg und wir wollen, dass die Reisezeit oder die Länge der gefundenen Weg minimal ist.

Solche Probleme werden Optimierungsprobleme genannt und bei denen wollen wir eine möglichst gute Lösung finden. Wie in der mathematischen Optimierung sind nur spezielle Klassen oder spezielle Instanzen von Optimierungsprobleme leicht lösbar. Im Allgemeinen sind diese Probleme *schwierig*. Informal heißt *schwierig* normalerweise:

(1) der Rechner schafft es schwierig oder gar nicht, das Problem zu lösen (Zeit- und Platzverbrauch)
(2) der Mensch schafft es nur schwierig oder gar nicht, das Problem zu lösen (Lösungsansatz zu kompliziert oder zu umfangreich).

Solche schwierige Optimierungsprobleme können wir mit Hilfe von Heuristiken lösen. Eine Heuristik liefert in der Praxis eine gute Lösung, allerdings können wir im Allgemeinen nicht sagen, wie gut diese gefundene Lösung im Vergleich zur besten Lösung des Problems ist. Wir unterscheiden zwischen den problemspezifischen Heuristiken, die für ein bestimmtes Problem zugeschnitten sind, und den Metaheuristiken. Die Metaheuristiken sind Strategien, wie für ein beliebiges Problem eine Lösung gefunden kann. Dabei müssen wir die problemspezifischen Teile der Metaheuristik selber definieren.

Neben Heuristiken können wir auch exakte Algorithmen und Approximationsalgorithmen verwenden. Ein exakter Algorithmus liefert für ein bestimmtes Optimierungsproblem immer die beste Lösung, wobei das für schwierige Probleme aufwendig ist.

Ein Approximationsalgorithmus ist ein Kompromiss zwischen Heuristiken und exakten Algorithmen. Dieser Algorithmus hat wenigen Aufwand als ein exakter Algorithmus und liefert eine Lösung mit einer bekannten Qualität. D.h. wir wissen, wie viel die Ausgabe der Approximationsalgorithmus von der besten Lösung maximal abweichen kann.

In folgenden Kapiteln werden die Grundlagen von Metaheuristiken und eine Kategorie von Metaheuristiken, nämlich die Metaheuristiken mit Einzellösung, genauer erläutert.

1.2 Motivation

Wie in der Einführung erwähnt, können wir Metaheuristiken für schwierige Optimierungsprobleme verwenden, d.h. für Probleme wo die exakten oder approximativen Lösungsansätze nicht schnell genug sind. Also z.B. *NP*-Schwere Probleme, aber auch riesige Instanzen von leichteren Problemen.

Solche Probleme treten heutzutage sehr oft auf, z.B. bei weltweiter Routenplanung möglicherweise mit einem Navigationssystem. Die Suche nach Wegen minimaler Kost ist ein leichtes Problem, allerdings enthält ein Graph für die Routenplanung allein in Deutschland mehrere Millionen von Kanten und Knoten.

Ein weiteres Beispiel wäre das Lösen des *Traveling-Salesman-Problems* bei einem Logistikunternehmen. Das *Traveling-Salesman-Problem* ist *NP*-Schwer und entspricht die Suche nach einem Hamilton-Kreis minimales Gewichts in einem Graphen, wobei ein Hamilton-Kreis eine Tour ist, die alle Knoten im Graphen genau einmal besucht.

Es gibt auch andere Stellen, wo Metaheuristiken gut einsetzbar sind. Beispielweise können während der Entwicklung eines Verfahrenes sehr spezifische Probleme auftreten, oder noch schlimmer, Spezialfälle. Um diese schnell lösen zu können, und dann sich mit dem echten Problem weiterzubeschäftigen, benutzt man gerne Heuristiken.

Wie in der Einführung erwähnt, spielen Speicherbedarf und Implementierungsaufwand der Ansätze auch eine Rolle. Der Einbau einer Metaheuristik in einem schon existierenden (und kostenlosen) Framework sollte zumindest einfacher sein, als manchmal deutlich komplizierte und nicht intuitive exakte Algorithmen zu implementieren. Oder wenn es nicht einfacher ist, dann schneller und mit wenigem Aufwand.

Es gibt auch Probleme, für die es keine bekannten exakten Lösungsansätze gibt, wie z.B. Randerkennung von Sensornetze (ohne Ortsangaben). Weiter sind Fälle bekannt, wo die Güte der Lösung von einem Menschen bewertet werden sollte, wie beim Malen von den Netzwerken des öffentlichen Nahverkehrs.

Als letzter (für diese Seminararbeit) Eintrag dieser sehr langen Liste stehen die kombinatorischen Probleme. Auch dafür gibt es diverse Typen von Metaheuristiken, wie

z.B. *Tabu Search* und *Simulated Annealing*.

Ausführliche Beispiele werden in Kapitel **4**. betrachtet.

1.3 Grundlagen von Metaheuristiken

Hier werden einige Bausteine von den Metaheuristiken mit Hilfe der lokalen Suche vorgestellt. Die lokale Suche ist wahrscheinlich die einfachste Metaheuristik. Der Ablauf sieht folgendermaßen aus:

wähle die Startlösung S
do
    $S' \leftarrow S$
    $S \leftarrow$ beste Lösung aus $N(S)$
while $f(S)$ besser als $f(S')$

In diesen vier Zeilen steht die Strategie der Heuristik, also die Metaheuristik. Um diese erklären zu können, müssen zuerst die unbekannten Symbole und Begriffe definiert werden. Diese werden kurz erläutert, um einen allgemeinen Überblick zu geben, und später werden sie näher betrachtet.

(1) eine Startlösung

Die lokale Suche geht von einer Startlösung aus und versucht sie zu verbessern. Wenn man bereits eine Lösung hat, die unbefriedigend ist, kann man diese als Startlösung verwenden. Ein Beispiel für so einen Fall wäre der bisherige (suboptimale) Plan eines Logistikunternehmens. Außerdem gibt es Probleme, wo diese Startlösung schon gegeben ist. Da diese beiden Fälle nicht immer eintreten, kann man die Startlösung auch konstruieren. Mehr zur Startlösung im Abschnitt **3.3**.

(2) die Funktion N

Die Funktion N beschreibt die Nachbarschaft einer Lösung S. Diese Nachbarschaft enthält alle Lösungen S', die mit Hilfe von einem Operator aus S erzeugt werden können. Ein solcher Operator wäre z.B. das Vertauschen zweier Knoten in einem Hamiltonkreis, um einen neuen Kreis zu bekommen. Abschnitt **3.4** enthält weitere Informationen über solche Operatoren und über auf diesem Weg erzeugte Nachbarschaften.

(3) die Funktion f

Damit man über die Güte einer Lösung sprechen kann, muss man diese Lösungen bewerten können. Die Funktion F macht genau das, sie nimmt eine Lösung, und ordnet dazu eine Zahl aus R, d.h. $f : M \to \mathbb{R}$, mit M als die Lösungsmenge.

f wird auch die Zielfunktion genannt. Ähnlich zur mathematischen Optimierung sollte die Ausgabe von f nicht mehrdimensional sein, damit man zwei Zielfunktionswerte einfacher vergleichen kann. Fälle, wo man mehrere Kriterien für die Güte einer Lösung hat, und wie man damit umgehen kann, werden im Abschnitt **3.1** besprochen.

Offensichtlich ist es wichtig, dass alle Lösungen des Suchraums einen Zielfunktionswert haben. Sonst kann die Metaheuristik auf einer Lösung landen, die sich nicht bewerten lässt. Wie man diese Zielfunktion wählen sollte, wird auch im Abschnitt **3.1** näher betrachtet.

(4) die beste Lösung aus $N(S)$

Nachdem die Nachbarschaft und die Zielfunktion definiert sind, ist die beste Lösung $S^* \in N(S)$ mit: $\forall S' \in N(S)$: $f(S^*)$ ist besser als $f(S')$. Alternative Möglichkeiten, wie man die nächste Lösung wählt und deren Vor- und Nachteile, werden im Abschnitt **3.5** besprochen.

(5) $f(S)$ besser als $f(S')$

Je nach Definition der Zielfunktion will man diese entweder maximieren oder minimieren. Beim Maximieren ist S besser als S', falls $f(S) \geq f(S')$. Entsprechend, beim Minimieren, ist S besser als S', falls $f(S) \leq f(S')$.

Man sieht, dass eine Lösung gewählt wird und solange verbessert wird, bis ein lokales Optimum gefunden ist. Das heißt, dass zu einem bestimmten Zeitpunkt nur eine Lösung weiterentwickelt wird. Eine Metaheuristik, die so vorgeht, wird eine Metaheuristik mit Einzellösung (S-Metaheuristik) genannt. Dagegen betrachten populationsbasierte Metaheuristiken (P-Metaheuristiken) mehrere Lösungen, d.h. eine Population von Lösungen. Die Elemente dieser Population werden schrittweise verbessert. Wenn man eine S-Metaheuristik wählt, sollte man ihre Vor- und Nachteile gegenüber P-Metaheuristiken beachten:

- schlecht parallelisierbar, da der Vorgang sehr sequenziell ist

- größere Probleme mit lokalen Optima, da wenigere Teile des Suchraums untersucht sind

+ leichter zu verfolgen/debuggen, da zu einem Zeitpunkt nur einen Pfad im Suchraum betrachtet wird

+ weniger Rechenaufwand, da einen kleineren Anteil des Suchraums untersucht

wird (als bei P-Metaheuristiken)

2 Untersuchung des Suchraums

In diesem Kapitel sind zwei Begriffe kurz vorgestellt, die den Vorgang der Metaheuristik bezeichnen. Diese sind die Intensivierung und die Diversifizierung. In der Regel wird es eine Ausbalancierung zwischen den beiden angestrebt, um die besten Ergebnisse zu gewinnen.

2.1 Intensifizierung

Der Begriff *Intensivierung* (Verschärfung, Abbildung 1) zeigt, wie stark die Metaheuristik gute Lösungen angeht. D.h. es wird mit bekannten guten Lösungen weitergemacht, anstatt möglichst viel vom Suchraum zu entdecken. Bei der lokalen Suche wird immer die beste Lösung aus der Nachbarschaft der aktuellen Lösung gewählt, also handelt es sich bei dieser Metaheuristik um eine sehr starke Verschärfung. Es wird nur ein Pfad im Suchraum betrachtet, der nur aus besten Nachbarn besteht. Metaheuristiken mit Einzellösung sind in dieser Richtung (*Intensivierung*) orientiert.

Die Hoffnung bei diesem Vorgang ist, dass das globale Optimum in einer vielversprechenden Region liegt. Eine vielversprechende Region im Suchraum ist eine Region mit mehreren guten Lösungen.

2.2 Diversifizierung

Unter *Diversifizierung* (Streuung, Abbildung 2) versteht man, wie gut der Suchraum von der Metaheuristik untersucht wird. Genauer gesagt, wenn der Suchraum in gleich großen Regionen geteilt ist, werden diese Regionen gleichmäßig untersucht. D.h. die Metaheuristik verteilt sich im Suchraum und bleibt nicht nur bei einer Region. Bei populationsbasierten Metaheuristiken wird eine bessere Streuung angestrebt. Falls man immer eine zufällige Lösung aus dem Suchraum wählt, wird die Suche *randomisierte Suche* genannt. Diese hat eine sehr starke Streuung.

Bei diesem Vorgang werden nicht immer die beste Lösungen aus den Regionen gefunden, allerdings hat die Metaheuristik zumindest die Region besucht. So ist die Wahrscheinlichkeit geringer, dass das globale Optimum in einer nicht besuchten Region liegt.

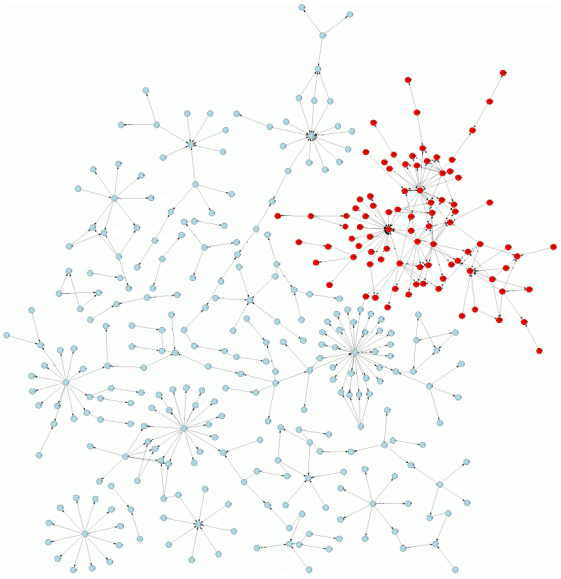

Abbildung 1: Besuchte Knoten (mit Rot) einer Metaheuristik, die in Richtung *Intensivierung* geht.

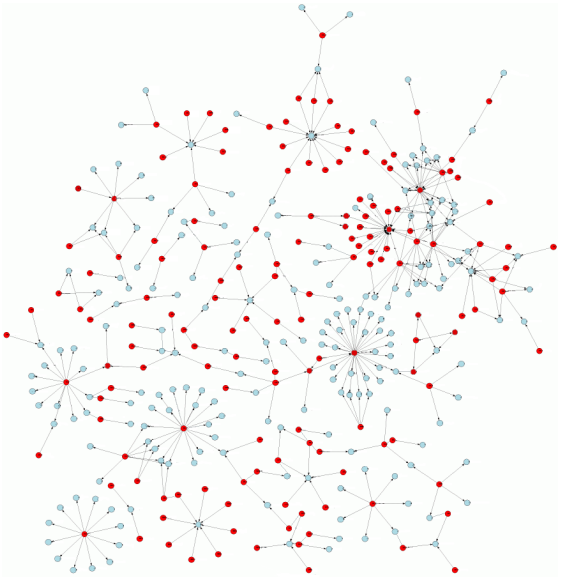

Abbildung 2: Besuchte Knoten (mit Rot) einer Metaheuristik, die in Richtung *Diversifizierung* geht.

3 Bausteine von S-Metaheuristiken

In diesem Kapitel sind einige Bausteine genauer betrachtet, die jede S-Metaheuristik hat. Diese sind die Zielfunktion, die Repräsentation, die Nachbarschaft und die Startlösung.

3.1 Auswahl der Zielfunktion

Wichtig beim Aufbau der Metaheuristik ist die Wahl der Zielfunktion. Man sollte die Zielfunktion so schnell wie möglich berechnen können, da sie ständig angewendet wird. Dabei sind mögliche Optimierungstechniken zu verwenden, oder wenn es keine starken Anforderungen an der Genauigkeit gibt, kann man sie auch approximieren.

Außerdem sollte die Zielfunktion im Suchraum so wenig wie möglich lokale Optima besitzen. Mehr dazu im Kapitel **5**.

Auch bei Problemen, wo die Antwort entweder *ja* oder *nein* ist, sollte man die Zielfunktion so modellieren, dass sie möglichst viel Informationen über die Güte der Lösung angibt. Beispielsweise kann man beim 3SAT Problem als Zielfunktionswert die Anzahl der erfüllten Klauseln benutzen und dann diese Zielfunktion maximieren. Beim 3SAT Problem wird es geprüft, ob eine aussagenlogische Formel erfüllbar ist. Diese Formel besteht aus Klauseln, die mit dem Operator $\wedge$ verknüpft sind. Die Klauseln haben drei mit dem Operator $\vee$ verknüpften Literale. Beim n-Damen-Problem kann die Anzahl von Damen, die sich bedrohen, verwendet und minimiert werden.
Ein ausführliches Beispiel für die Modellierung eines ja oder nein Problems ist im Kapitel **4.1** (Sudoku-Beispiel) zu finden.

Wie bei der mathematischen Optimierung, gibt es Fälle, wo man mehrere Kriterien hat, d.h. mehrere Zielfunktionen. Z.B. bei der Routenplanung sind kurze im Sinne von Länge und Reisezeit Wege gesucht. Und genau wie in der Mathematik hat man meherere Möglichkeiten damit umzugehen.

(1) Lexikographisch. Die Zielfunktionen werden nach Wichtigkeit sortiert. Dann wird mit der wichtigsten Zielfunktion optimiert. Falls ein eindeutiges Optimum gefunden ist, wird dieses zurückgegeben. Sonst wird die Menge von Optima mit der nächsten wichtigen Zielfunktion optimiert, usw.

(2) Zieldominanz. Es wird eine Zielfunktion als wichtigste gewählt, für alle andere nimmt man ein Anspruchsniveau. Dieses Niveau sollte von der entsprechenden Zielfunktion erfüllt werden, d.h. eine zu minimierende Zielfunktion sollte unter ihrem Niveau sein, und eine zu maximierende Zielfunktion sollte über diesem sein. Es werden also weitere Einschränkungen hinzugefügt. Hier hat man das Problem die Anspruchsniveaus geeignet

zu wählen.

(3) Skalarisierung. Für jede Zielfunktion $fi(.)$ wird ein Gewicht ci gewählt. Die Summe aller Gewichte sollte gleich 1 sein. Dann werden die Zielfunktionen in einer neuen Zielfunktion F zusammengefasst: $F(s) := \sum_i fi(s) * ci$. Das Gewicht sollte der Wichtigkeit der Zielfunktion entsprechen.

3.2 Repräsentation der Lösungen

Die Lösungen von den meisten Problemen sind auf mehrere Arten und Weisen in einem Rechner darstellbar. Ein berühmtes Beispiel dafür ist das n-Damen-Problem. Die direkte Repräsentation einer Lösung wäre die ganze Spielbrett-Situation zu speichern. Dabei gibt es $\binom{n^2}{n}n!$ mögliche Repräsentationen. Der Suchraum enthält alle Repräsentationen, also wird dieser riesig sein. Die Repräsentation die aber benutzt wird, um mit dem Problem umzugehen, ist eine Liste mit n Zahlen von *1* bis n. Diese Liste hat keine Wiederholungen. Die Zahl an der Stelle i entspricht der Zeile, wo es eine Dame in Spalte i steht. Es gibt $n!$ solche Repräsentationen, d.h. der Suchraum ist deutlich kleiner.

Solche Einschränkungen des Problems sollen wir, wenn möglich, in der Repräsentation einbauen. Ein kleinerer Suchraum bedeutet weniger Aufwand für die Metaheuristik. Trotzdem wird oft die direkte Repräsentation benutzt, da sie meistens durch das Problem schon gegeben ist.

Weiterhin müssen wir beachten, dass verschiedene Operationen auf der Repräsentation der Lösungen ausgeführt werden. Zuerst wird die Repräsentation mit der Zielfunktion ausgewertet und danach werden die Operatoren zur Erstellung der Nachbarschaft angewendet. Es ist wichtig die Repräsentation so zu wählen, dass diese möglichst effizient ist. Beispielsweise ist ein Bitflip auf einer Bitvektor-Repräsentation (Abbildung 4) fast mühelos.

Diese erstellenden Operatoren werden eine Repräsentation ändern. Es ist dabei wichtig zu beachten, dass kleine/große Änderungen an der Repräsentation auch kleine bzw. große Änderungen an der Lösung verursachen (mehr dazu in Kapitel **3.4**).

Eine weitere wichtige Eigenschaft ist die Vollständigkeit der Repräsentation. Die gewählte Repräsentation muss alle Lösungen des Problems abdecken. Wenn dies nicht erfüllt ist, wird die Metaheuristik nicht repräsentierbare Lösungen ignorieren, auch wenn diese möglicherweise optimal sind.

Neben der Bitvektor-Repräsentation sind Permutationen (Abbildung 3) auch eine übliche Repräsentation. Zwei weitere Repräsentationen sind in Kapitel **4.1** vorgestellt.

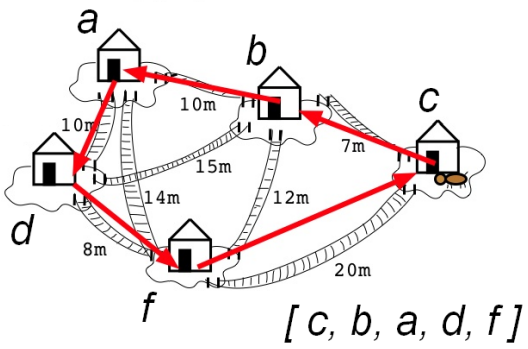

Abbildung 3: Eine Permutation der besuchten Orte als Repräsentation bei einem *Traveling Salesman Problem*

3.3 Konstruktion der Startlösung

Die lokale Suche geht von einer Startlösung aus. Probleme, bei denen eine Lösung schon gegeben ist und diese verbessert werden sollte, brauchen keine Konstruktion der Startlösung.

Solche Probleme treten beispielsweise bei der Entwicklung von algorithmischen Verfahren oder Beschleunigungstechniken auf, wo der Algorithmus eine grobe Lösung erstellt und dann diese verfeinert. Eine Metaheuristik kann mit kleinerem Programmieraufwand solche groben Lösungen noch mehr verbessern. Dann kann man schnell erfahren, wie gut der Ansatz ist, der diese groben Lösungen erstellt.

Leider kommen nicht alle Probleme mit einer Startlösung, also braucht man Methoden zum Aufbau dieser Startlösung. Zwei von den möglichen Ansätzen sind, sie zufällig zu wählen oder sie *greedy* zu konstruieren. Beide hängen von der Repräsentation ab.

(1) Randomisiert:

Eine zufällige Startlösung lässt sich schnell erstellen. Dabei können verschiedene Durchläufe der S-Metaheuristik zu verschiedenen Resultaten führen. Das ist nicht immer schlecht, allerdings wenn eine Metaheuristik mehrere Parameter hat und eine Startlösung braucht, kann die Feinsteuerung dieser Parameter mit dieser Methode schwierig sein. Ein weiterer Nachteil ist, dass so eine Startlösung sehr weit weg von einem lokalen Optimum sein kann, d.h. die Metaheuristik wird mehr Zeit brauchen. Also wird der Vorteil der schnellen Konstruktion vermindert.
Trotzdem gibt es auch Fälle, wo dieser Ansatz zu besseren Ergebnissen als der *greedy* Ansatz führt.

Beispielsweise kann man beim Rucksack-Problem zufällige Gegenstände wählen, solange die Einschränkung für die Kapazität erfüllt ist.

i	1	2	3	4	5	6	7	$\sum$
Gi	21	18	15	12	10	9	8	$42+12$
Ki	7	6	5	4	4	3	2	$14+4$
$\boldsymbol{Li}$	**0**	**1**	**1**	**1**	**0**	**1**	**0**	

Abbildung 4: **(1)** : Bitflip bei einer Bitvektor-Reprsentation ($i = 4$)

(2) Greedy:

Ausgehend von einer leeren Menge werden immer Elemente hinzugefügt, die einen maximalen Gewinn liefern. Das wird solange wie möglich gemacht, was für die verschiedenen Repräsentationen verschiedene Bedeutungen hat. Für eine Permutation (z.B. bei Hamilton-Kreisen) heißt es, dass alle Elemente hinzugefügt sind und für eine Bitvektor-Repräsentation, dass die Einschränkungen von weiteren Änderungen verletzt sind.

Die *greedy* Konstruktion für das Rucksack-Problem nimmt die wertvollsten Gegenstände bis keine Kapazität mehr vorhanden ist.

Für die meisten Anwendungen von Metaheuristiken wird eine zufällige Startlösung gewählt. Die Verwendung von Pseudozufall vermindert das Problem bei der Feinsteuerung von Parametern. Zusätzlich, bei P-Metaheuristiken werden mit Hilfe von Zufall sehr diverse Startpopulationen erstellt. Die Diversität der Populationen ist bei den P-Metaheuristiken geschätzt.

3.4 Erstellung der Nachbarschaft

Die Nachbarschaft einer Lösung kann mit Hilfe von Operatoren erstellt werden. Diese werden *move* Operatoren genannt und führen eine kleine Änderung an der Repräsentation aus. Sie "bewegen" die Lösung um eine Distanz im Suchraum. Wenn die kleine Änderung eine kleine Distanz verursacht, hat die Nachbarschaft starke Lokalität. Wenn nicht, gibt es einen kleineren Zusammenhang zwischen den Nachbarn und der Lösung, und die Metaheuristik wird sich nicht sinnvoll verhalten.

Die *move*-Operatoren sehen und ändern nur die Repräsentation. Sie haben kein Wissen, ob die kleinen Änderungen wirklich klein sind. Also sollte diese starke Lokalität in der Nachbarschaft von der Repräsentation der Lösungen gewährleistet sein.

Für verschiedene Probleme ist normalerweise die Struktur der Nachbarschaft unterschiedlich. Für zwei üblichen Repräsentationen kann ein *move*-Operator so aussehen:

(1) Bitvektoren. Beispielsweise ist für das Rucksack-Problem eine solche Repräsentation (Abbildung 4) denkbar, wobei das i-te Bit speichert, ob der i-te Gegenstand genommen wird oder nicht. Hier kann der *move*-Operator an jeder Position i einen Bitflip ausführen

und so werden n Nachbarn der aktuellen Lösung erstellt (wenn es n Gegenstände gibt). Ein Vorteil bei diesem Ansatz ist, dass die Hamming-Distanz zwischen zwei Repräsentationen von Lösungen auch der Distanz zwischen diesen Lösungen im Suchraum entspricht. Die Definition der Distanz ist im Kapitel **5.1** zu finden.

(2) Permutationen. Für Probleme wie Scheduling und TSP ist eine Permutation (Abbildung 3) gut geeignet. Die Reihenfolge der Elemente entscheidet, was die nächste Aufgabe ist oder welcher Knoten als nächstes besucht werden soll. Mögliche *move*-Operatoren sind z.B. Vertauschen von zwei Elementen der Permutation (*swap* Operator) oder ein Element entfernen und dieses an einer anderen Position einfügen (*insert* Operator).

Wenn das Problem Einschränkungen hat, wie z.B. das Rucksack-Problem (Kapazität überschritten), kann der *move*-Operator auch ungültige Lösungen erstellen. Man sollte diese erkennen und mit der S-Metaheuristik nicht betrachten. Sonst wird die S-Metaheuristik möglicherweise ungültige Lösungen benutzen.

Weiterhin sollte man beachten, dass die S-Metaheuristik in der Nachbarschaft eine neue Lösung finden muss. Um die Suche kurz zu halten, sollte (wenn möglich) die erstellte Nachbarschaft klein sein. Falls die Nachbarschaft sehr groß ist, brauchen wir weitere Ansätze, um sie zu untersuchen. Solche Ansätze werden in der nächsten Kapitel erwähnt. Eine weitere Voraussetzung für den *move*-Operator ist, dass der *move*-Operator schnell berechenbar sein sollte, da er ständig verwendet wird.

Im zweiten Beispiel der Kapitel **4.1** steht ein *move*-Operator, der eine riesigen Nachbarschaft erstellt und außerdem nicht einfach zu berechnen ist. D.h. ein *move*-Operator, der vermeidet werden soll.

3.5 Untersuchung der Nachbarschaft

In der erstellten Nachbarschaft müssen wir eine neue Lösung finden. In der Regel soll diese neue Lösung zumindest besser als die aktuelle Lösung sein. Am meisten wird nach der besten Lösung aus der Nachbarschaft gesucht und dann wird diese mit der aktuellen Lösung verglichen. Ein paar Möglichkeiten, so eine Lösung zu finden, sind folgende:

(1) Komplett. Alle Lösungen aus der Nachbarschaft werden betrachtet. So wird immer die beste Lösung gefunden. Bei großen Nachbarschaften kann das allerdings problematisch sein. Auch für kleine Nachbarschaften will man die Anzahl der angeschauten Lösungen aus der Nachbarscaft verkleinern, da z.B. die Zielfunktion eine teueren Simulation ist und sie für alle diesen Lösungen durchgeführt sein muss. Wenigere zu betrachteten

Nachbarn bedeuten natürlich auch eine kürzere Laufzeit.

(2) Erster Treffer. Es wird die erste gefundene Lösung benutzt, die besser ist. Dieser Ansatz ist selten benutzt, da er im schlimmsten Fall genau so schlecht ist, wie die exakte Durchsuchung. Hier wird außerdem die Intensivierung der Metaheuristik vermindert, da die besten besuchte Lösungen nicht am meisten ausgenutzt sind.

(3) Randomisiert. Wie bei der erwähnten randomisierten Suche werden neue Lösungen mit Hilfe von Zufall gewählt. Ein zufälliger Nachbar wird aus der Nachbarschaft gewählt. Eine weitere Möglichkeit ist nur zufällige Nachbarn zu wählen, die bestimmte Kriterien erfüllen, z.B. solche mit einem besseren Zielfunktionswert. Hier wird die Streuung der Metaheuristik vergrößert.

Eine Möglichkeit, bessere Diversifizierung ohne Zufall zu bekommen, ist größere Nachbarschaften zu betrachten. Um eine gute Lösung in so einer sehr großen Nachbarschaft schnell zu finden, braucht man Ansätze, die nicht die ganze Nachbarschaft untersuchen. Wie man in diesen Fällen vorgeht, hängt vom Aufbau der Nachbarschaft ab, wobei die Ansätze auf Punkte **(4)** und **(5)** basiert sind:

(4) Heuristisch. Mit Hilfe von einer weiteren Heuristik kann die nächste Lösung angenähert werden. Man soll beachten, dass dadurch ein besserer Nachbar nicht immer gewährleistet ist. Einige Möglichkeiten hier sind:

(4.1) Wenn die Nachbarschaft mit einem *move*-Operator N erstellt ist, der der n-fachen Anwendung eines anderen *move*-Operators N' entspricht, kann man z.B. mit Hilfe von N' eine lokale Suche bauen. Die Repräsentation bleibt gleich und die Startlösung ist die aktuelle Lösung. Die Suche ist durch n Iterationen begrenzt und die Ausgabe ist die verbesserte Lösung.

(4.2) Wenn die große Nachbarschaft der *k-hop* Nachbarschaft entspricht (alle Lösungen mit Distanz $\leq k$ zur aktuellen Lösung), kann man wieder eine durch k Schritte begrenzte lokale Suche verwenden. Die Nachbarschaft dieser Suche ist dann die *1-hop* Nachbarschaft, die Startlösung ist die aktuelle Lösung und die Repräsentation ist unverändert.

(4.3) Allgemein kann man eine selbständige (Meta)Heuristik entwerfen [6], d.h. eine (Meta)Heuristik, die nur eine neue Lösung aus der Nachbarschaft findet und die wir noch entwerfen müssen. Hier ist es wichtig, dass der ganze Suchraum der neuen Heuristik die Nachbarschaft der ursprünglichen Metaheuristik ist. Es ist mindestens ein neuer *move*-Operator erforderlich.

(5) Exakt. In manchen Fällen ist es immer noch möglich die beste Lösung in der Nachbarschaft zu finden ohne sie komplett zu betrachten. Die Nachbarschaft wird normalerweise speziell konstruiert, damit die Suche nach der neuen Lösung ein leicht lösbares Problem ist. Zwei konkrete Beispiele sind **(5.1)** und **(5.2**):

(5.1) Dynamische Programmierung. Im so genannten Vierbesserungsgraph (*improvement graph*) ist es möglich, mit Hilfe von einem kürzesten Weg, den besten Nachbar zu bestimmen. Dieser Graph ist für ein Partitionierungsproblem folgendermaßen definiert: die zu partitionierenden Elemente sind mit Zahlen von *1* bis n kennengezeichnet. Die Partition des Elementes i wird mit $p[i]$ bezeichnet. Dann sind die Knoten des Graphs die Elemente und eine Kante zwischen Knoten i und j ist mit einer *move*-Operation verbunden, die i von seiner Partition $p[i]$ in $p[j]$ bewegt, und j aus $p[j]$ entfernt. Dabei ist das Gewicht dieser Kante (i, j) gleich den neuen Kosten (nach der *move*-Operation) der alten Partition (vor der *move*-Operation) von j.
In einer Nachbarschaft, die durch die m-fachen Anwendung der *move*-Operator erstellt ist, ist der beste Nachbar auf einem kürzesten Weg der Länge m. So eine Suche nach dem kürzesten Weg im Vierbesserungsgraph wird *Dynasearch* genannt.

(5.2) Matchings. Eine Nachbarschaft kann mit Hilfe von Matchings erstellt und untersucht werden. Beispielsweise werden für das TSP aus der aktuellen Lösung k Knoten entfernt. Dann wird ein bipartiter Graph mit linker Seite die entfernten Knoten $\{v_1, ..., v_k\}$ und rechter Seite die restlichen Knoten $\{u_1, ...u_{n-k}\}$ konstruiert. Die Gewichte der Kanten zwischen der linken und rechten Seiten (v_i, u_j) sind die Kosten für die Tour $[u_1, ..., u_j, v_i, u_{j+1}, ..., u_{n-k}, u_1]$. In diesem Graph wird ein Matching mit minimalen Kosten berechnet. Knoten aus der linken Seite werden dann in der rechten Seite anhand von den Kanten des Matchings hinzugefügt.

4 Anwendung der lokalen Suche

Diese Kapitel enthält zwei ausführliche Beispiele. Die Beispiele sind aus echten Verfahren genommen und zeigen, wie die Bausteine aus der Kapitel **3** in der Realität aussehen können. Ferner geht es in der Kapitel an, wie das Hauptproblem von der lokalen Suche gelöst wird, nämlich, wie die lokale Suche aus lokalen Optima entkommen kann.

4.1 Beispiele

Das Sudoku-Puzzle kann mit Hilfe von *Simulated Annealing* gelöst werden [3], wobei *Simulated Annealing* eine *S*-Metaheuristik ist.

Für die Repräsentation wird die bekannte 9x9 Tabelle von Sudoku verwendet. Genau wie im Puzzle wird diese Tabelle mit Zahlen von 1 bis 9 gefüllt. D.h. die direkte Repräsentation wird genommen.

Die Zielfunktion zählt die Anzahl von Wiederholungen, die in einer Spalte oder in einer Zeile vorkommen. Da bei Sudoku keine Zahl in einem 3x3 Quadrat, einer Spalte oder einer Zeile sich wiederholen darf, will man diese Zielfunktion minimieren. Warum die Zielfunktion keine Wiederholungen in den 3x3 Quadraten zählt, lässt sich mit der Konstruktion der Startlösung und mit dem *move*-Operator erklären.

Als Startlösung werden in den 3x3 Quadraten zufällige Zahlen von 1 bis 9 gesetzt, wobei es in jedem 3x3 Quadrat keine Wiederholungen gibt (Abbildung 5 (b)). Die Startlösung ist *randomisiert* gewählt.

Man merkt, dass die Startlösung schon eine der drei Regeln von Sudoku erfüllt. Das ist natürlich kein Zufall und der *move*-Operator erhält diese Eigenschaft.

Der *move*-Operator vertauscht zwei zufällige Zahlen in einem 3x3 Quadrat. Die Tatsache, dass die mit der Startlösung gewährleistete Regel erfüllt bleibt, ist nicht der einzige Vorteil von diesem *move*-Operator. Um die Zielfunktion mit so wenig Aufwand wie möglich zu berechnen, werden die Wiederholungen in jeder Zeile und in jeder Spalte einer Lösung zusätzlich gespeichert (Abbildung 5 (c)). Wenn der *move*-Operator angewendet ist, treten Änderungen nur bei maximal zwei Zeilen und Spalten ein. Also müssen nur für diese Zeilen und Spalten die gespeicherten Werten neu berechnet werden und der Zielfunktionswert lässt sich aus dem alten Wert ableiten (Abbildung 5 (d)).

Das nächste Beispiel kommt aus dem Dll-Verfahren [1], das den Rand eines Sensornetzes berechnen soll. Allgemein besteht ein Sensornetz aus Knoten, die mit allen anderen Knoten in einer kleinen Umgebung kommunizieren können. Wenn zwei Knoten miteinander kommunizieren können, steht eine Kante im Graph des Sensornetzes zwischen diesen. Größe Regionen im Sensornetz, wo es keine Bedeckung gibt, werden interne

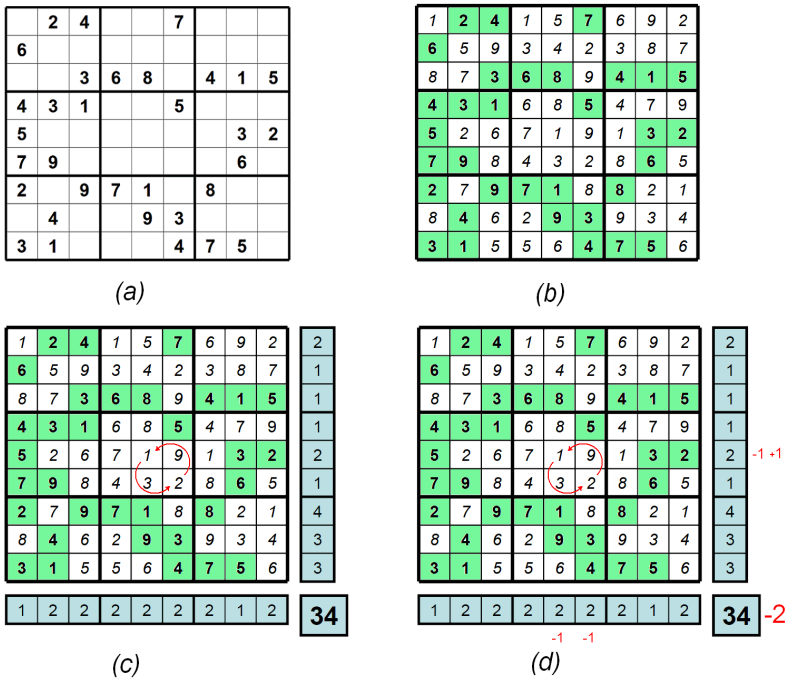

Abbildung 5: Eine Instanz des Puzzles (a), die Startlösung (b), eine *move*-Operation (c) und die Berechnung des neuen Zielfunktionswertes (d)

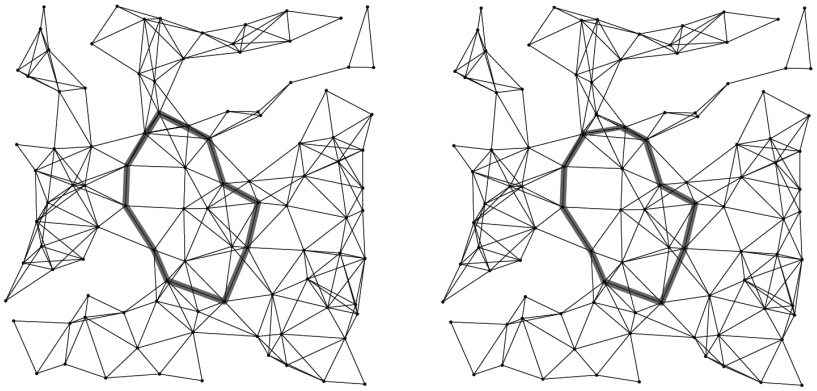

Abbildung 6: Die aktuelle Lösung (links) und die Ausgabe des *move*-Operators (rechts)

Löchern genannt. Knoten, die am Rand von Löcher sind, formen den internen Rand des Sensornetzes.

Im Ablauf des Algorithmus werden Zyklen ermittelt, die grob interne Löcher im Sensornetz einschließen (grob, da sie keine genügende Beschreibung des Lochs angeben). Diese Zyklen will man verfeinern. Eine einfache Möglichkeit das zu schaffen, ist eine Heuristik zu benutzen.

Der Zyklus wird verkleinert, indem man zwei Kanten des Zyklus durch eine dritte Kante ersetzt, falls diese drei Kanten ein Dreieck formen (Abbildung 6). Das ist eine lokale Suche, wobei ihre Bausteine nicht explizit genannt sind:

- Die Repräsentation ist eine Liste von Knoten, die einen Zyklus beschreibt.

- Die Zielfunktionswert ist die Länge dieser Liste.

- Der *move*-Operator ersetzt [..., a, b, c, ...] in der Liste mit [..., a, c, ...], falls zwischen a und c eine Kante existiert.

Dieser Ansatz wird *1-hop shrinking* genannt. Die ursprüngliche Methode heißt *k-hop shrinking* und funktioniert wie folgt: zwischen zwei Knoten, die mit einen Weg mit k Kanten verbunden sind (*k-hop* Distanz), wird nach einem kürzeren Pfad (mit weniger als k Kanten) gesucht. Falls ein solcher Pfad existiert, werden die k Kanten durch diesen Pfad ersetzt.

Hier wird nur der *move*-Operator geändert, indem alle Knoten in der Liste mit einem zyklischen Abstand gleich k betrachtet werden. Zwischen zwei solchen Knoten wird dann der kürzester Pfad im Graph berechnet und die Liste wird geändert, falls der Pfad weniger als k Kanten hat. So ein *move*-Operator ist leider viel aufwändiger als der Operator bei *1-hop shrinking*, auch wenn er bessere Resultate liefert.

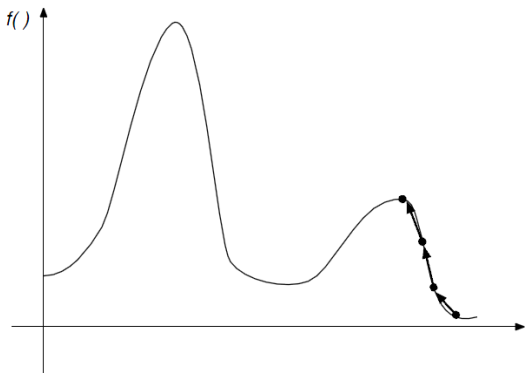

Abbildung 7: Das Problem mit lokalen Optima bei lokaler Suche [4].

4.2 Umgang mit lokalen Optima

Ein wichtiges Problem bei der lokalen Suche sind lokale Optima (Abbildung 7). Schon im Pseudokode (Kapitel **1.3**) von der lokalen Suche sehen wir, dass ein lokales Optimum für die Abbruchbedingung genügend ist. Es existieren weitere *S*-Metaheuristiken, die auf der lokalen Suche basieren und verschiedene Mechanismen verwenden, um lokale Optima zu behandeln.

Es gibt auch Fälle, wo man mit den lokalen Optima zufrieden ist, z.B. wenn die Zielfunktion im Suchraum nur ein lokales Optimum besitzt. Oder wenn die lokale Suche benutzt wird, um eine relativ gute Lösung zu finden (beispielsweise bei der Suche nach besserer Lösung in einer Nachbarschaft).

Da für die meisten Probleme das nicht der Fall ist, existieren ein paar Möglichkeiten, mit lokalen Optima umzugehen.

(1) Aus der Nachbarschaft werden auch Lösungen akzeptiert, die schlechter als die aktuelle sind (Abbildung 8).
Dabei muss man beachten, dass in diesem Fall Zyklen entstehen können, d.h. Elemente können mehrmals besucht werden.
Simulated Annealing und *Tabu Search* sind zwei *S*-Metaheuristiken, die mit diesem Ansatz arbeiten.

(2) Änderungen an der Problemdarstellung. Das kann mehrere Änderungen bedeuten:
- Transformation der Zielfunktion, um wenigere oder weniger variierende lokale Optima im Suchraum zu bekommen (Abbildung 9).
- Ändern der Nachbarschaften. Die Idee dahinter ist, dass ein globales Optimum

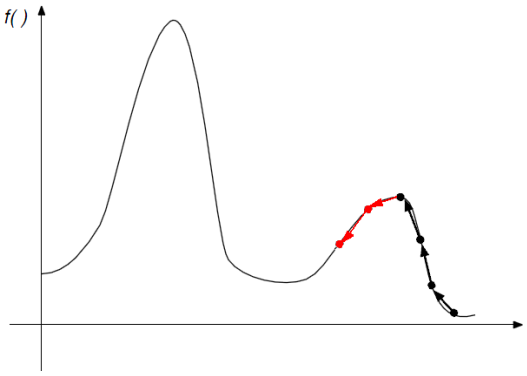

Abbildung 8: **(1)** : Akzeptanz von schlechteren Lösungen.

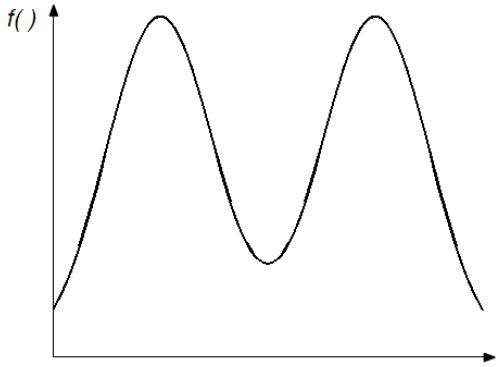

Abbildung 9: **(2)** : Ähnliche lokale Maxima, ein lokales Minimum.

auch ein lokales Optimum in allen Nachbarschaften ist, wobei ein lokales Optimum in einer Nachbarschaft nicht unbedingt lokales Optimum in einer anderen Nachbarschaft ist. Also wird es z.B. eine Menge von Nachbarschaften betrachtet und die beste Lösung von allen diesen Nachbarschaften wird gefunden. Ein großer Vorteil ist, dass man viel bei der Auswahl dieser Menge variieren kann.
Variable Neighborhood Search ist eine *S*-Metaheuristik, die diesen Ansatz verwendet.

- Änderungen an der Eingabe. Man kann die Ausgabe einer lokalen Suche für weitere *S*-Metaheuristiken als Startlösung verwenden. Man kann sogar diese Ausgabe wieder für eine lokale Suche benutzen. Ohne Änderungen wird natürlich die lokale Suche gleich am Anfang fertig sein, deshalb führt man eine Störung in der Ausgabe und wendet die lokale Suche auf dieser "deformierten" Lösung an.
Iterated Local Search ist eine *S*-Metaheuristik, die in dieser Richtung geht.

5 Fitness-Landschaften

Der Zielfunktionswert einer Lösung wird auch Fitness dieser Lösung genannt. Je besser eine Lösung, desto fitter ist sie. Wenn man über die Fitness des Suchraums spricht, bedeutet das, wie die Zielfunktion sich im Suchraum verhält - ob es mehrere lokale Optima gibt oder nur eins, ob viele Stellen existieren, wo ganze Nachbarschaften die gleichen Zielfunktionswerte haben, usw.

Die Fitness des Suchraums wird *Fitness-Landschaft* genannt. Sie ist offensichtlich vom Suchraum, also von der Nachbarschaft-Funktion, abhängig. Außerdem spielt die Zielfunktion eine Rolle. Eine Zielfunktion kann mehrere lokalen Optima als eine andere (beispielsweise konvexe) Zielfunktion verursachen.

Wenn die Ziel- und Nachbarschaft-Funktion gewählt sind, kann man über die Fitness-Landschaft der Metaheuristik reden. In einem zusammenhängenden Suchraum, wo die Zielfunktion nur ein lokales Optimum besitzt, wird die lokale Suche immer die beste Lösung finden. Der Suchraum ist zusammenhängend, falls wir ausgehend von einer beliebigen Lösung mit Hilfe des *move*-Operators alle andere Lösungen erreichen können. Das bedeutet folgendes: mit Hilfe der Fitness-Landschaft sind Aussagen über das Verhältnis der Metaheuristik möglich.

Aus diesem Grund wird die Fitness-Landschaft einer Metaheuristik betrachtet. Falls unsere Metaheuristik sich besonders gut oder besonders schlecht verhält, wollen wir den Grund dafür kennen. So können wir die Metaheuristik neu entwerfen, falls sie schlechte Ausgaben liefert. Oder wir wissen, dass die sehr gute Ausgaben kein Zufall sind. Wie gesagt ist die Fitness-Landschaft von der Nachbarschaft und von der Zielfunktion abhängig, also können wir die Fitness-Landschaftsanalyse durchführen, ohne eine bestimmte Stra-

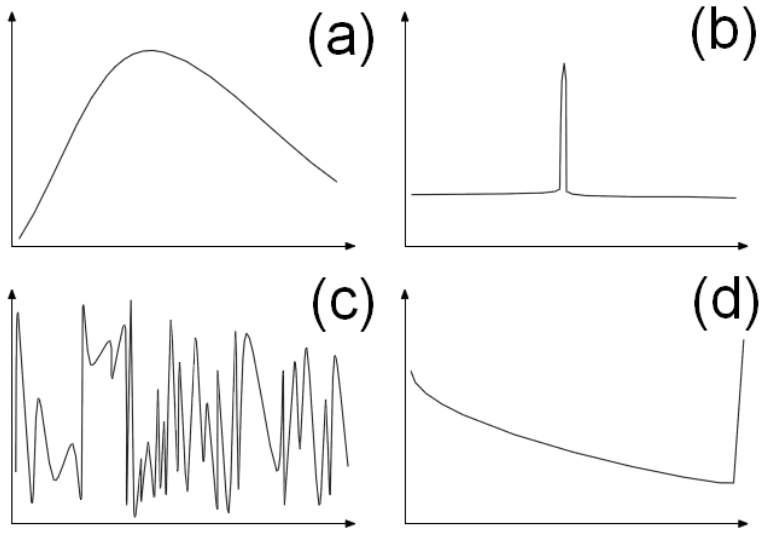

Abbildung 10: Beispielhafte Fitness-Landschaften:
unimodal (a), Nadel im Heuhaufen (b), schroff (c), irreführend (d)

tegie zu wählen. D.h. diese Analyse hilft uns auch, eine passende Metaheuristik für unsere schon definierten Bausteine zu wählen.

Als Nächstes wird die Fitness-Landschaftsanalyse genauer betrachtet.

5.1 Eigenschaften der Fitness-Landschaft

Bevor wir über die Eigenschaften der Fitness-Landschaft reden, müssen wir zuerst die Distanz im Suchraum definieren. Die Distanz zwischen zwei Lösungen s und s' ist die kleinste Anzahl von Anwendungen des *move*-Operators, um die Lösung s in s' zu transformieren. Für die meisten Maßnahmen wird diese Distanz in Kombination mit der Fitness von Lösungen benutzt, um eine Aussage für das Verhalten einer Metaheuristik zu treffen.

Den Suchraum können wir als einen Graphen darstellen, wobei jede Lösung ein Knoten ist. Eine Kante existiert zwischen zwei Lösungen s und s', falls sie benachbart sind. Das hängt natürlich von der Nachbarschaft ab, d.h. vom verwendeten *move*-Operator. Alle Kanten haben dann das gleiche Gewicht (z.B. 1) und die Distanz zwischen s und s' ist der kürzeste Weg von s nach s' in diesem Graph. Also berechnen wir die Distanz im Suchraum mit Hilfe von kürzesten Wegen. Das ist eine Prozedur, die relativ aufwendig ist. Bei bestimmten Kombinationen von einem Operator und einer Repräsentation können wir die Distanz wesentlich leichter und schneller berechnen. So ein Beispiel war schon erwähnt, nämlich wenn wir einen Bitflip bei einer Bitvektor-Repräsentation benut-

zen. Wenn wir die Fitness-Landschaftsanalyse durchführen, sollten wir überprüfen, ob unsere Bausteine so eine Kombination sind. Das wird die Analyse deutlich beschleunigen und das wollen wir immer.

Die Fitness-Landschaftsanalyse hat einige Voraussetzungen:

(1) Für verschiedene *move*-Operatoren sind die Distanzen und die Fitness-Landschaften möglicherweise unterschiedlich und wir sollten eine Fitness-Landschaft mit Hilfe einer anderen Distanz, die durch anderen *move*-Operator definiert ist, nicht betrachten. Sonst ist die Ausgabe der Analyse nicht korrekt.

(2) Der Suchraum muss verbunden sein, d.h. im Graph existiert ein Pfad zwischen allen Paare von Lösungen (s_i, s_j). Das ist eine Voraussetzung für die Verfahren, die Fitness-Landschaften analysieren. Sonst kann es sein, dass die Analyse nur für einen getrennten Teil des Suchraums gemacht ist. Anhand von so einer Analyse können wir nichts über die anderen Teile des Suchraums sagen.

5.2 Untersuchung der Fitness-Landschaft

Wenn wir bestimmte Eigenschaften der Landschaft bestimmen wollen, wie z.B. Anzahl von lokalen Optima, deren Verteilung, Rauheit, usw., können wir i.A. zwei Typen von Methoden verwenden:

(1) Bestimmung von Verteilungen.
Hier wird die Verteilung der lokalen Optima untersucht:

- Die Verteilung im Suchraum können wir mit Hilfe der Distanz von Lösungen erkennen. Wir betrachten die durchschnittliche Distanz einer Population von Lösungen P:

$$\frac{\textbf{durchschnittliche Distanz in P}}{\textbf{maximale Distanz in P}}, \text{ mit:}$$

durchschnittliche Distanz in P $:= \frac{1}{|M|} \sum_{(s,\ s')\in M} d(s, s')$,
maximale Distanz in P $:= \max_{(s,s')\in M} d(s, s')$,
$M := \{(s,\ s') \in P \times P | s \neq s'\}$,

$d(s, s') :=$ die Distanz zwischen zwei Lösungen s und s' (bzgl. des *move*-Operators)

Falls dieses Verhältnis klein ist, d.h. gegen 0 geht, dann sind die Lösungen in P im Suchraum gebündelt. Wenn P genug Informationen enthält, d.h. genug lokale Optima, können wir eine Aussage über das Verhalten einer Metaheuristik machen. In diesem Fall, wo das Verhältnis klein ist, wird eine S-Metaheuristik sehr wahrscheinlich gute Lösungen liefern, da die guten Lösungen in einer Region im Suchraum sind.

Falls das Verhältnis groß ist, sind die lokale Optima gleichmäßig im Suchraum verteilt. Also eine P-Metaheuristik wird bessere Lösungen liefern.

- Die Verteilung im Raum der Zielfunktion wird anhand der Zielfunktionswerte verschiedener Lösungen bestimmt, beispielsweise der Differenz zwischen den Zielfunktionswerten von der Lösungen einer Population P und dem globalen Optimum (oder der besten bekannten) s^*:

$$\frac{\textbf{Distanz von } \boldsymbol{P} \textbf{ zu } \boldsymbol{s^*}}{\textbf{Normalisierung}}, \text{ mit:}$$

Distanz von P zu $s^* := \sum_{s \in P} f(s) - f(s^*)$,
Normalisierung $:= |P| f(s^*)$

Falls dieses Verhältnis klein ist, ist die Population nahe zu s^*, d.h. wir werden mit jeder Lösung aus dieser Population zufrieden sein. Wenn zusätzlich die Population sehr viele lokalen Optima enthält, können wir einfach die lokale Suche verwenden. Sie liefert immer lokale Optima, also Lösungen die sehr wahrscheinlich in P liegen, da P viele lokalen Optima hat.

(2) Bestimmung von Korrelationen.
Wir können die Korrelation zwischen Fitness und Distanz betrachten. Dabei sind mehrere Maßnahmen möglich:

- Länge der Wege, die von der S-Metaheuristik gelaufen sind. Bei einer robusten Landschaft sind diese kurz, da es viele Optima gibt. Bei glatten Landschaften sind diese länger, da die S-Metaheuristik mehrere Schritte braucht, um ein Optimum zu finden. Hier können wir die lokale Suche mehrmals mit zufälligen Startlösungen ausführen und dann die gelaufenen Distanzen betrachten.

- Autokorrelation. Es wird die Fitness von Lösungen beobachtet, die eine Entfernung d haben. Diese Methode ist auch für $d = 1$ anwendbar. Dann bedeuten große positive Werte, dass die benachbarten Lösungen ähnliche Werte haben (glatte Landschaft). Kleine Werte dagegen weisen auf eine robuste Landschaft hin. Je mehreren Lösungen mit Distanz d wir betrachten, desto genauer ist diese Maßnahme.

- Korrelation zwischen Fitness und Distanz zum globalen Optimum. Eine große positive Korrelation bedeutet, dass die S-Metaheuristik das Problem relativ leicht lösen

kann, da die Fitness sich näher zur besten Lösung verbessert. Eine große negative Korrelation bedeutet, dass es entfernte Lösungen gibt, die eine gute Fitness haben und der *move*-Operator nach diesen Lösungen streben wird. Da das Wissen über globale Optima nicht immer vorhanden ist, können wir bei dieser Methode die beste bekannte Lösung auch verwenden. Das kann natürlich zu ungenauen Schlussfolgerungen führen. Wenn mehrere globalen Optima bekannt sind, nehmen wir die nächste zur aktuellen untersuchten Lösung.

6 Zusammenfassung

Neben den Exakt- und Approximationsalgorithmen sind die Metaheuristiken ein weiteres Werkzeug, mit schwierigen Optimierungsproblemen umzugehen. Sie sind besonders wertvoll, wenn die exakte und Approximationsalgorithmen zu lange bei der Lösung des Problems brauchen, oder wenn die Anforderungen an der Genauigkeit der gefundenen Lösung nicht so hoch sind.

Zwei Kategorien von Metaheuristiken sind die populationsbasierten Metaheuristiken und die Metaheuristiken mit Einzellösung. Wir sollen für unseres Problem eine Metaheuristik wählen, die von der geeigneten Kategorie ist. Um diese Wahl treffen zu können, sollen wir die Vor- und Nachteile der Kategorien kennen.

Viele Metaheuristiken haben gemeinsamen Bausteine, wie z.B. die Repräsentation von Lösungen und die Nachbarschaft. Falls wir diversen Metaheuristiken verwenden wollen, ist es wertvoll, solche Bausteine kennenzulernen.

Die lokale Suche ist der Ursprung von mehreren Metaheuristiken und spielt eine Rolle bei der Fitness-Landschaftsanalyse. Diese Suche liefert immer lokale Optima und ist besonders einfach als Idee. Für spezielle Problemen ist die lokale Suche sehr gut geeignet, beispielsweise falls die Zielfunktion im Suchraum konvex ist oder uns ein lokales Optimum genügt.

Die Fitness-Landschaften sind eine Möglichkeit die entworfenen Metaheuristiken zu analysieren und somit sie zu verbessern. Die Fitness-Landschaften sind außerdem ein Versuch, die praxisorientierten Metaheuristiken in der Theorie der Informatik einzubringen. Für die wichtigen Eigenschaften der Fitness-Landschaf existieren mehrere Maßnahmen, die uns helfen diese Eigenschaften nachzuprüfen. Ein Beispiel für so eine Eigenschaft ist die Robustheit, wobei in einer robusten Fitness-Landschaft die Metaheuristiken sich schlecht verhalten.

7 Literatur

Literatur

[1] Dezun Dong, Yunhao Liu, and Xiangke Liao. Fine-grained boundary recognition in wireless ad hoc and sensor networks by topological methods. pages 1–10, 2009.

[2] Cyril Fonlupt, Denis Robilliard, Philippe Preux, and El-Ghazali Talbi. Fitness landscapes and performance of meta-heuristics. pages 255–265.

[3] Rhyd Lewis. Metaheuristics can solve sudoku puzzles. pages 1–11.

[4] Sean Luke. Essentials of metaheuristics. pages 15–26, 2009. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[5] S. Olafsson. Metaheuristics. pages 633–654, 2006.

[6] David Pisinger and Stefan Ropke. Large neighborhood search. pages 4–18.

[7] Cesar Rego and Fred Glover. The traveling salesman problem and it's variations. pages 309–368.

[8] El-Ghazali Talbi. Metaheuristics - from design to implementation. pages 88–126, 2009.

[9] Yue Wang and Jie Gao. Boundary recognition in sensor networks by topological methods. pages 1–12, 2006.