BACHELORARBEIT

zur Erlangung des akademischen Grades

„Bachelor of Science in Engineering“ im Studiengang Informatik

Genetische Algorithmen und ihre Anwendung zur Lösung des Rucksackproblems

Ausgeführt von: Florian Hilbinger
Personenkennzeichen: 1310257069

1. Begutachter: DI Dr. Gerd Hesina

Elsbach, 22.01.2016

FH University of Applied Sciences
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich, als Autor und Urheber der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. etwa §§ 21, 46 und 57 UrhG idgF sowie § 11 Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien).

Ich erkläre insbesondere korrekt fremde Inhalte, gleich welcher Form, übernommen zu haben und bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. § 11 Abs. 1 Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Ort, Datum	Unterschrift

Kurzfassung

NP-schwere Probleme bezeichnen in der Theoretischen Informatik die am schwierigsten zu lösenden Probleme und sind eine Übermenge der Klasse NP - all jenen Problemen, deren optimale Lösung nicht in vernünftiger, polynomieller Zeit berechnet werden kann. In den meisten Fällen reicht jedoch bereits eine annähernd optimale Lösung aus, deren Ermittlung durch Optimierungsverfahren in wesentlich geringerer Zeit möglich ist. Ein Beispiel für ein NP-schweres Problem ist das Rucksackproblem und den zurzeit wohl besten Kompromiss aus effizienten und effektiven Optimierungsverfahren für dieses stellen Genetische Algorithmen dar. Der Begriff stammt aus dem Bereich der Genetik, da die eingesetzten Operatoren Selektion, Crossover und Mutation den Vorgang der natürlichen Evolution und das Überleben des Stärksten widerspiegeln. Die vorliegende Arbeit erläutert die theoretischen Konzepte, die Vorteile und Unterschiede Genetischer Algorithmen gegenüber den herkömmlichen Verfahren Greedy Algorithmus sowie Dynamischer Programmierung, und enthält eine praktische Implementierung eines Genetischen Algorithmus zur annähernden Lösung des Rucksackproblems. Anhand mehrerer Messreihen mit unterschiedlichem Umfang erfolgen ein empirisch-experimenteller Vergleich der Algorithmen und ein praktischer Beleg der theoretischen Behauptungen. Schließlich wird die Komplexität des implementierten Genetischen Algorithmus erhoben und im Zuge dessen bewiesen, dass der Genetische Algorithmus in polynomieller Zeit abläuft.

Schlagwörter: Genetischer Algorithmus, Rucksackproblem, Optimierungsproblem

Abstract

In theoretical computer science, NP-hard problems are the most difficult ones to solve and represent a superset of the class NP that contains all problems for which an optimal solution cannot be calculated in a reasonable, polynomial time. In most cases, however, an approximately optimal solution is sufficient, which can be found in much less time by means of optimization algorithms. Among many others, the knapsack problem is one example of a NP-hard problem. At present, it is best computed by genetic algorithms which provide both effective and efficient approximations of optimization problems. The term is derived from the field of genetics, because its operators, selection, crossover, and mutation reflect natural evolution and the survival of the fittest. This paper explains the theoretical concepts, the advantages and differences of genetic algorithms in comparison to the more traditional greedy algorithm and dynamic programming. It also provides a practical implementation of a genetic algorithm approximately solving the knapsack problem. Based on several measurement series with differing scopes, the algorithms are compared using an empiric-experimental approach and the theoretical claims are proven in a practical manner. The paper concludes with a determination of the genetic algorithm's complexity, showing that the genetic algorithm does indeed run in polynomial time.

Keywords: Genetic Algorithm, Knapsack Problem, Optimization Problem

Inhaltsverzeichnis

1 Einführung in die Genetische Programmierung

1.1 Was ist Genetische Programmierung?

1.1.1 Definitionen und Abgrenzung

Wissenschaft entspringt dem Wunsch des Menschen, seine Umwelt zu verstehen und zu kontrollieren. Die digitale Revolution - der weitläufige Einsatz des Computers, der zu maßgeblichen Umbrüchen in Technik und Gesellschaft geführt hat - ermöglichte einen immer höheren Grad an Kontrolle und Verständnis der Natur. Viele Menschen sehen die Schaffung künstlicher Intelligenz in Form von Computerprogrammen als krönenden Abschluss dieser Entwicklung.
Aus dem Versuch, menschliches Lernen und biologische Evolution mit Computern zu simulieren, entwickelten sich die folgenden Teilgebiete der Informatik:
Neuronale Netzwerke, Maschinelles Lernen und Evolutionäre Algorithmen. ([1], S. 1)

Diese Arbeit behandelt Ausprägungsformen von Evolutionären Algorithmen. Bevor auf diese jedoch genauer eingegangen werden kann, ist ein Hinweis angebracht:
In einschlägiger Literatur werden die Begriffe Evolutionäre Algorithmen, Evolutionäre Programmierung, Evolutionäre Programme, Genetische Algorithmen, Genetische Programmierung und Genetische Programme oftmals synonym verwendet, was beim Lesen zu Verwirrung führen kann. Daher folgt für diese Arbeit nun eine Abgrenzung der obigen Begriffe, die auf deren Verwendung in den bedeutendsten zur Recherche verwendeten Literaturquellen fußt.

Allen obigen Ausdrücken gemein ist das Ziel, durch Imitation von Evolutionsprozessen aus der Natur Probleme mit Hilfe eines Computers zu lösen.
Als oberste Abstraktionsebene und Sammelbegriff für verschiedene Unterkategorien von Algorithmen verwenden Mitchell [1] Evolutionäre Algorithmen, Michalewicz [2] Evolutionäre Programme und Polli, Langdon und McPhee [3] Genetische Programmierung.
Da letztgenannter „Field Guide to Genetic Programming“ als neueste dieser Publikationen eine Momentaufnahme des aktuellen Forschungsstand auf dem Gebiet präsentiert, wird an dieser Stelle „Genetische Programmierung“ als Überbegriff für die gesamte Disziplin definiert:

„Genetic programming (GP) is a collection of evolutionary computation techniques that allow computers to solve problems automatically.” ([3], Preface)

Das Gebiet der Evolutionären Algorithmen ist also mit dem Begriff „Genetische Programmierung“ gleichzusetzen.

Im Feld der Genetischen Programmierung wiederum existieren unterschiedliche Vorgehensweisen, die zwar dasselbe Ziel verfolgen, sich jedoch in ihrer Ausführung unterscheiden. Drei wesentliche Ausprägungen Genetischer Programmierung sind Evolutionäre Programme, Genetische Algorithmen und Genetische Programme.

1966 entwickelten Fogel, Owens und Walsh erstmals sogenannte Evolutionäre Programme. Dabei wurden mögliche Lösungen zu gegebenen Problemen als Endliche Automaten dargestellt, und der biologische Evolutionsprozess durch die zufällige Mutation der Zustandsänderungsdiagramme und der darauf folgenden Auswahl des Stärksten simuliert.
([1], S. 2)

Genetische Algorithmen hingegen wurden von John Holland, seinen Kollegen und Studenten an der University of Michigan in den 1960er und 1970er Jahren entwickelt und führen Operationen auf binären Zeichenketten fixer Länge aus. Wie in der Natur werden diese „DNA-Strings“ durch Mutationen und Kreuzungen von Generation zu Generation verändert und die am besten angepassten Ausprägungen beibehalten. ([1], S.2; [2], Preface to the First Edition; [4], S.1)
Ein Beispiel einer binären 32 Bit Zeichenkette zeigt Abbildung 1:

00001111010110000111101010110110

Abbildung 1: Binäre Zeichenkette mit 32 Bit Länge

Genetische Programme wiederum werden üblicherweise als Syntax Bäume ausgedrückt. Dabei werden die Blätter durch Variablen und Konstanten - in der Fachliteratur als „terminals“ bezeichnet - und die Verzweigungen durch Operatoren - sogenannte „functions“ - beschrieben [3], S. 9, wie in Abbildung 2 zu sehen ist.

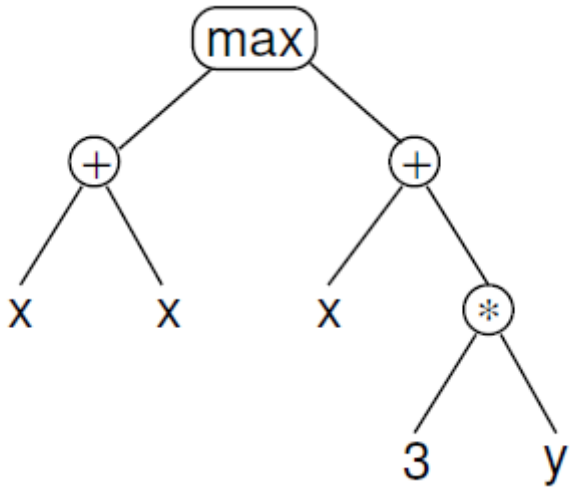

Abbildung 2: Syntax Baum der Funktion max(x+x, x+3*y) (Quelle: [3], S. 10)

Die Variation in den einzelnen Evolutionsstufen der Genetischen Programme erfolgt durch Crossover - dem Erstellen von Kinderprogrammen durch Kombination zufällig ausgewählter Teile zweier Elternprogramme - und Mutation - der Schaffung von Kinderprogrammen durch die Abänderung zufällig gewählter Teile eines Elternprogramms. ([3], S.1)

Im Gegensatz zu Genetischen Algorithmen ermöglichen Genetische Programme dem Computer, selbst Programme zu schreiben und sind somit flexibler als Genetische Algorithmen, die auf Zeichenketten fixer Länge operieren, und Evolutionäre Programme, die an kleine Endliche Automaten [2], S.1, gebunden sind.
Die Idee der selbstschreibenden Genetischen Programme wurde von John R. Koza in „Genetic Programming: On the Programming of Computers by Means of Natural Selection“ [5] im Jahr 1990 vorgeschlagen.

Diese Arbeit widmet sich der Anwendung eines Genetischen Algorithmus zur Lösung des Rucksackproblems, welches in Kapitel 1.2 erläutert wird, und wird sich daher mit Genetischen Algorithmen im Detail beschäftigen.
Genetische Programme werden nicht näher beleuchtet, für Interessierte bietet sich „Genetic Programming: On the Programming of Computers by Means of Natural Selection“ [5] von John R. Koza als Nachschlagewerk an.
Im Bereich der Evolutionären Programme empfiehlt sich die Lektüre von „Genetic Algorithms + Data Structures = Evolution Programs“ [2] von Zbigniew Michalewicz.

1.1.2 Die Natur als Vorbild

Bereits Alan Turing, John von Neumann, Norbert Wiener und viele weitere Pioniere des Computer Zeitalters wurden durch die Vision motiviert, Computer Programme mit künstlicher Intelligenz zu versehen und ihnen die Fähigkeiten zu verleihen, sich selbst zu replizieren und zu lernen, sich an ihre Umwelt anzupassen und diese zu kontrollieren. Dabei interessierten sich diese Wissenschaftler ebenso sehr für Biologie und Psychologie wie auch für Elektronik, und sahen natürliche Systeme als Metaphern, die sie in der Umsetzung ihrer Visionen leiten sollten. Daher sollte es nicht verwundern, dass bereits seit den Anfangszeiten der Informatik Computer zur Modellierung des Gehirns, zur Nachahmung menschlichen Lernens und zur Simulation der biologischen Evolution eingesetzt wurden. ([1], S. 1)

Seit den 1950er und 1960er Jahren wird im Bereich Evolutionärer Algorithmen im Allgemeinen geforscht. Als Begründer der Genetischen Algorithmen gilt John Holland, der diese in den 1960ern erfunden hat. Im Gegensatz zu Evolutionärer Programmierung war Hollands Ziel nicht, Algorithmen zum Lösen spezifischer Probleme zu entwerfen, sondern Wege zu finden, das Phänomen der Anpassung aus der Natur auf Computer Systeme zu übertragen. ([1], S.2 f.)

Als Vorlage diente dabei das Konzept vom Überleben des Stärksten und der natürlichen Selektion, welches Charles Darwin 1859 in „On the Origin of Species by Means of Natural Selection“ präsentierte. Darwin erkannte, welche Bedingungen als Voraussetzung für das Auftreten des evolutionären Prozesses in der Natur gelten:

- Ein Lebewesen (eine Spezies) besitzt die Fähigkeit, sich selbst zu reproduzieren
- Es gibt eine Population solcher selbst-reproduzierender Lebewesen
- Unter den selbst-reproduzierenden Lebewesen herrscht eine gewisse Verschiedenheit
- Diese Verschiedenheit entscheidet zu einem gewissen Teil über das Überleben eines Lebewesens in seiner Umgebung

Hollands 1975 erschienenes Buch „Adaptation in Natural and Artificial Systems“ stellte ein allgemeines Rahmenwerk zur Abbildung aller adaptiven Systeme, gleich ob natürlich oder künstlich, zur Verfügung und zeigte, wie der natürliche evolutionäre Prozess auf künstliche Systeme angewendet werden kann. ([5], S. 17)

Mitchell [1], S. 3, bezeichnet Hollands Konzept als bahnbrechende Innovation. Während in vorgehenden Modellen Evolutionärer Algorithmen von zwei Individuen ausgegangen wurde (einem Elternelement und seinem Nachwuchs), führte Holland die Idee einer ganzen Population (population) von Chromosomen (binären Strings) ein, die durch natürliche Selektion (natural selection) mit Hilfe der aus der Genetik inspirierten Operatoren Mutation (mutation), Kreuzung (crossover) und Umkehrung (inversion) in eine neue Generation überführt wurden. Bis vor kurzem galten Hollands Ideen als theoretische Grundlage für beinahe alle nachfolgenden Arbeiten auf dem Gebiet der Genetischen Algorithmen. In den letzten Jahren jedoch haben sich die unterschiedlichen Forschungsgebiete im Bereich Genetischer Programmierung zunehmend vermischt und der Begriff „Genetische Algorithmen“ wird von Forschern heutzutage oftmals anders verstanden als Hollands ursprüngliche Konzeption.
Diese Arbeit orientiert sich jedoch am klassischen Konzept Hollands, mit der Abweichung, dass auf die Anwendung des Umkehrungsoperators verzichtet wird. Dieser wird in heutigen Implementationen Genetischer Algorithmen kaum verwendet, da seine Vorteile, insofern sie denn existieren, sich nicht wirklich bewährt haben. ([1], S. 8)

Nach diesem kurzen Überblick über die Geschichte der Genetischen Algorithmen und deren wesentlicher Ideen, beschäftigen sich die nachfolgenden Unterkapitel mit dem strukturellen Ablauf von Genetischen Algorithmen und gehen näher auf die einzelnen Bestandteile ein.

Kapitel 1.3 zeigt das Potenzial Genetischer Algorithmen im Vergleich zu konventionellen Algorithmen auf und gibt einen Überblick über deren Anwendungsmöglichkeiten.

1.1.3 Elemente Genetischer Algorithmen

Die Mechanismen Genetischer Algorithmen sind erstaunlich einfach und beinhalten im Grunde keine komplexeren Vorgänge als das Kopieren und den teilweisen Austausch von binären Strings, sowie deren Manipulation an bestimmten Positionen.
Dadurch wirken Genetische Algorithmen sehr attraktiv, da trotz der Einfachheit ein erstaunlich großer Effekt erzielt wird. ([4], S. 10)

Genetische Algorithmen bestehen im Wesentlichen aus den folgenden Elementen:
([1], S. 8, ergänzt durch Teilnehmer)

- Teilnehmer
- Evaluierungsfunktion
- Operatoren
    - Selektion
    - Mutation
    - Crossover

Zur Beschreibung der Teilnehmer Genetischer Algorithmen wird ein Vokabular aus der natürlichen Genetik eingesetzt. Ein Genetischer Algorithmus geht von einer zumeist zufällig generierten Population aus, die sich aus mehreren Individuen (vielfach auch als Strings oder Chromosomen bezeichnet) zusammensetzt. Diese Chromosomen werden üblicherweise durch eine binäre Zeichenkette (String) abgebildet, wobei der String jedes Individuums einer Population dieselbe Länge besitzt. Ein einzelnes Chromosom wiederum besteht aus mehreren aneinandergereihten Genen (auch Features, Characters, Decoders genannt), die ebenfalls binär kodiert sind und jeweils einen Abschnitt der gesamten binären Zeichenkette eines Chromosoms abbilden. Jedes Gen kontrolliert bestimmte Eigenschaften (Features) des Chromosoms, und die Position eines Gens in der Zeichenkette des Chromosoms wird Locus (Latein für Ort, plural: loci) genannt. Jede Eigenschaft (zum Beispiel die Haarfarbe) kann in den Individuen einer Population unterschiedliche Ausprägungen (blond, braun, schwarz, …) annehmen. Diese unterschiedlichen Zustände eines Gens werden unter dem Begriff Allel (Eigenschaftswerte) zusammengefasst.
Jedes Individuum in Form eines Chromosoms kann als möglicher Lösungskandidat für ein Problem betrachtet werden. Ein Evolutionsprozess der auf einer Population von Chromosomen ausgeführt wird entspricht somit der Suche durch einen Raum unterschiedlicher potenzieller Lösungskandidaten. ([2], S. 15)

Zur Beurteilung der Fitness eines Chromosoms wird die Evaluierungsfunktion (auch Fitnessfunktion oder „objective function“) eingesetzt. Die Fitness ist dabei ein Maß dafür, wie gut ein Lösungskandidat das gegebene Problem bewältigen kann. ([1], S.8)

Der Selektionsoperator beschreibt einen gewichteten Reproduktionsvorgang:
Zur Bildung einer neuen Generation werden individuelle Strings gemäß ihrem Fitnesswert kopiert. Je größer der Fitnesswert, desto höher ist die Wahrscheinlichkeit, dass ein Individuum zur Bildung von Nachwuchs in der nächsten Generation beiträgt. Es bietet sich hierbei der bildliche Vergleich mit einem Roulette Rad an: je höher die Fitness eines Individuums, desto größer ist der „Slot" auf dem Roulette Rad, der dem Individuum zugeteilt wird. Zur Reproduktion einer Population wird das Roulette Rad n mal (n = Größe der Population) gedreht. Die „Gewinner" jeder Drehung dürfen ihr Erbmaterial zur Reproduktion bereitstellen. Daher sind fittere Strings durch eine größere Zahl an Nachwuchs in der Folgegeneration abgebildet. ([4], S. 10 f.)

Crossover als zweiter Operator Genetischer Algorithmen läuft in zwei Stufen ab:
Für jedes Individuum der durch Selektion neu geschaffenen Population wird ein zufällig generierter Wert r im Bereich [0…1] mit der Crossoverwahrscheinlichkeit p_c verglichen.
Ist $r < p_c$, so wird das Individuum zum eigentlichen Crossover zugelassen, andernfalls bleibt das Individuum unverändert. Nun werden aus der Menge aller zum eigentlichen Crossover zugelassenen Individuen zufällige Paare gebildet. Für jedes Paar werden die Strings der beiden Individuen an einer zufälligen Position k im Bereich [1, (l - 1)] (l = Stringlänge) durchschnitten und die Teilstrings zwischen $k+1$ und l mit jenen des jeweils anderen Individuums vertauscht. Im Fall, dass die Anzahl der zum eigentlichen Crossover zugelassenen Individuen ungerade ist, wird entweder ein weiteres Individuum aus der neuen Population zum Crossover zugelassen, oder ein für das Crossover zugelassenes Individuum aus der selektierten Menge entfernt. ([2], S. 40; [4], S. 12)
Die Crossoverwahrscheinlichkeit p_c kann als Mischungsrate verstanden werden: je höher sie gewählt wird, desto eher vermischen sich Strings untereinander. Ein oft angewandter Wert für p_c bewegt sich im Bereich um 0,7. ([1], S. 11)

Schließlich existiert mit Mutation ein dritter Operator Genetischer Algorithmen.
Dieser wird auf jedes Bit eines Strings angewandt und kehrt mit einer Mutationswahrscheinlichkeit p_m den Wert des Bits um (vertauscht 0 und 1). Im Vergleich zu Selektion und Crossover spielt die Mutation eine sekundäre Rolle, da die Mutationswahrscheinlichkeit p_m üblicherweise sehr klein (zum Beispiel p_m = 0,001) gewählt wird, um gute Resultate zu erzielen.
Spärlich eingesetzt beugt der Mutationsoperator einem frühzeitigen, unwiederbringlichen Verlust von potenziell nützlichem, genetischem Material vor, der durch allzu übereifrige Selektion und Crossover hervorgerufen werden kann. (bestimmte 0,1 Folgen gehen verloren)
([1], S. 10; [2], S. 41; [4], S. 14)

1.1.4 Ablauf Genetischer Algorithmen

Nachdem im vorangehenden Unterkapitel die einzelnen Elemente Genetischer Algorithmen im Detail erläutert wurden, zeigt dieses Unterkapitel deren Einsatz im strukturellen Ablauf Genetischer Algorithmen.

Mitchell [1], S.8, und Michalewicz [2], S. 33 ff., beschreiben den Ablauf Genetischer Algorithmen in folgenden Schritten:

1. Erschaffung einer Anfangspopulation von n l-bit Chromosomen - erfolgt meistens zufällig; Wenn aber bereits Wissen über den Suchraum vorhanden ist, kann dieses zur Erschaffung miteinfließen.
2. Iterative Evolution der bestehenden Population zu einer neuen Generation durch:
    a. Berechnung der Fitness $f(x)$ jedes Chromosoms anhand der Evaluierungsfunktion
    b. Reproduktion durch auf Basis von Fitnesswerten gewichteter Selektion
    c. Modifikation von Chromosomen der neuen Population mittels:
        i. Austausch von Teilstrings im Zuge von Crossover
        ii. Bitumkehr infolge von Mutation

Für gewöhnlich werden 50 - 500 Iterationen durchlaufen und danach jener Lösungskandidat der finalen Population mit der höchsten Fitness als Lösung des Problems anerkannt. Infolge statistischer Abweichung kann es passieren, dass der Lösungskandidat mit der höchsten Fitness nicht Teil der finalen Population ist, sondern in einer früheren Generation auftritt und wieder verloren geht. Aus diesem Grund ist es gebräuchlich, den Lösungskandidaten mit der höchsten Fitness aus allen Generationen separat abzuspeichern. ([1], S. 10 f.; [2], S. 44)
Ein vollständiger Iterationszyklus wird auch als Durchlauf („run“) bezeichnet. Zufälligkeit spielt in jedem Durchlauf eine große Rolle, weshalb es üblich ist, mehrere Durchläufe mit unterschiedlichen Seeds (Startwerte für Generation von Zufallszahlen) durchzuführen, die sich in ihrem detaillierten Verhalten stark unterscheiden können. Aus den so erzielten Resultaten wird dann häufig der Durchschnitt gebildet. ([1], S. 11)

Ein Beispiel eines simplen Genetischen Algorithmus, das bei der Verinnerlichung der theoretischen Konzepte hilft, findet sich in Anhang A.

1.2 Das 0 - 1 Rucksack Problem

1.2.1 Definition des Rucksackproblems

Das Rucksack Problem ist ein Optimierungsproblem der Kombinatorik, welches unter vielen möglichen Lösungen die beste Lösung sucht [6], S.4.

Kellerer, Pferschy und Pisinger [7], S. 2 verwenden zur verständlicheren Beschreibung des Problems den Vergleich mit einem Bergsteiger, der einen Rucksack mit sich trägt. In diesen kann er *n* Gegenstände packen, die ihm den Aufstieg erleichtern sollen. Die Menge aller Gegenstände, die dafür zur Auswahl stehen, wird durch $N := \{1, \dots, n\}$ ausgedrückt. Der durch einen Gegenstand gewonnene Komfort wird als positive Nummer p_j angegeben. Jeder der Gegenstände besitzt zudem ein gewisses Gewicht w_j. Die Anzahl der Kopien eines Gegenstandes (zum Beispiel 2 gleiche Seile) im Rucksack des Bergsteigers ist durch x_j gegeben.
Der Bergsteiger kann nur ein bestimmtes Maximalgewicht in seinem Rucksack tragen, welches als Kapazität *c* bezeichnet wird. Er sucht nun nach einer Kombination von Gegenständen (1), deren Gewicht seine Kapazität nicht übersteigt und gleichzeitig einen möglichst hohen Level an Komfort ermöglicht (2).

Beim binären Rucksackproblem, auch als 0 - 1 Rucksackproblem bekannt, wird zusätzlich die Einschränkung getroffen, dass mindestens keine und maximale eine Kopie x_j eines Gegenstandes im Rucksack mitgeführt werden darf (3).

Das Rucksack Problem gehört in der Komplexitätstheorie zur Klasse der NP-schweren Optimierungsprobleme [7], S. 14.

Es würde den Rahmen dieser Arbeit sprengen, die unterschiedlichen Klassen der Komplexitätstheorie im Detail zu beschreiben. Ein kurzer Überblick über die wesentlichen Konzepte der Komplexitätstheorie, sowie der Grund für die Einordnung des Rucksackproblems als NP-schweres Problem, finden sich in Anhang B. Bei mangelhafter Kenntnis des Gebiets wird dessen Lektüre für das Verständnis der Motivation dieser Arbeit sowie zur Erklärung des Potenzials von Genetischen Algorithmen empfohlen. Zur Erlangung eines tieferen Verständnisses sei unter anderem auf das Buch „Introduction to Algorithms“ [8] von T. H. Cormen, C. E. Leiserson, R. L. Rivest und C. Stein verwiesen.

1.2.2 Darstellungsformen des Rucksackproblems

In seiner Form als Optimierungsproblem sucht das Rucksackproblem nach einer möglichst genauen Annäherung an die optimale Lösung und wird mathematisch beschrieben als:

$$\text{maximiere} \quad \sum_{j=1}^{n} p_j x_j \qquad (1) \quad \text{(Quelle [5], S. 3, Formel 1.1)}$$

$$\text{so, dass} \quad \sum_{j=1}^{n} w_j x_j \leq c \qquad (2) \quad \text{(Quelle [7], S.3, Formel 1.2)}$$

$$\text{mit den Bedingungen} \quad x_j \in \{0,1\}, \quad j = 1, \dots, n. \qquad (3) \quad \text{(Quelle [7], S.3, Formel 1.3)}$$

Die optimale Lösungsmenge des binären Rucksackproblems ist eine Untermenge X^* der Menge aller verfügbaren Gegenstände N und wird durch den optimalen Lösungsvektor $x^* = (x_1^*, \dots, x_n^*)$ und den Lösungswert z^* repräsentiert [7], S. 3.

Das Rucksackproblem kann zudem auch als Entscheidungsproblem abgebildet werden, welches „yes“ oder „no“ als Antwort liefert. Dafür wird ein Schwellwert t definiert und die Frage gestellt, ob die Summe aller positiven Nutzwerte einer bestimmte Kombination von Gegenständen unterhalb von t liegt, wobei $t \leq c$. Dies wird durch Anpassung der Formel (2)

$$\text{wenn} \quad \sum_{j=1}^{n} w_j x_j \leq t \leq c \quad \text{, dann liefere „yes“ zurück}$$

und unter Beachtung der in Einschränkungen in Formel (3) mathematisch ausgedrückt.

Das Optimierungsproblem kann in einen wiederholten Aufruf des Entscheidungsproblems umgewandelt werden. Dabei wird die optimale Lösung mit Hilfe von Binary Search ermittelt. Es ist bekannt, dass diese zwischen $a := 0$ und $b := \sum_{j=1}^{n} p_j$ (Summe aller Komfort - verbessernden positiven Werte) liegen muss. Durch $t := \frac{a+b}{2}$ wird der Schwellenwert als Mittelwert der unteren und oberen Schranken festgelegt. Nun wird das Ergebnis des Entscheidungsproblems ausgewertet. Liefert dieses „yes“ zurück, wird durch $a := t$ der jetzige Schwellenwert als neue untere Schranke festgelegt, ansonsten wird durch $b := t - 1$ der Wert nächstkleiner dem aktuellen Schwellenwert als neue obere Schranke definiert. Dieser Vorgang wird wiederholt bis $a = b$, was der optimalen Lösung entspricht. ([7], S. 484)

1.3 Genetische Programmierung - Wozu?

1.3.1 Potenzial

Goldberg [4], S. 1 f., zufolge stand die Robustheit, also die Balance zwischen Effizienz (Ergebnis im Verhältnis zum Aufwand) und Effektivität (Grad der Zielerreichung) zum Überleben in vielen unterschiedlichen Umgebungen, im Zentrum der Forschungen im Bereich Genetischer Algorithmen. Könnten künstliche Systeme mit Hilfe Genetischer Algorithmen durch eine verbesserte Anpassungsfähigkeit robuster gemacht werden, so würden teure Neugestaltungen reduziert oder gar komplett vermieden, da robuste Systeme länger und besser funktionieren. Entwickler künstlicher Systeme erkannten bald, dass natürliche Systeme den künstlichen in Sachen Robustheit weit voraus waren: Natürliche Systeme sind in der Regel mit Mechanismen zur Selbstreparatur, Selbststeuerung und Reproduktion ausgestattet, wohingegen künstliche Systeme diese Merkmale kaum besitzen.
Genetische Algorithmen sind bewiesenermaßen robuste Such- und Optimierungsalgorithmen für komplexe Problemfelder und kommen daher verstärkt in wirtschaftlichen, wissenschaftlichen und technischen Kreisen zum Einsatz. Dazu trägt unter anderem die einfache Berechenbarkeit bei gleichzeitig mächtigem Optimierungspotenzial bei.
Schließlich werden Genetische Algorithmen auch nicht durch einschränkende Annahmen bezüglich des Suchfeldes limitiert, wie dies bei anderen Algorithmen der Fall ist.
Als weiteren Vorteil Genetischer Algorithmen nennt Koza [5], S. 18, deren hohe Parallelisierbarkeit. Auch Mitchell [1], S. 4, betont die Signifikanz der hohen Parallelisierbarkeit Genetischer Algorithmen, welche vor allem in Suchalgorithmen bei einer riesigen Anzahl an Möglichkeiten von großer Bedeutung ist. Ebenso wie Goldberg weist sie weiters auf die herausragende Anpassungsfähigkeit Genetischer Algorithmen hin.

Michalewicz [2], S.13 ff., hebt besonders die Anwendungsmöglichkeiten Genetischer Algorithmen zum Lösen von Optimierungsproblemen hervor. Für diese äußerst komplexen Probleme existieren oft keine Algorithmen, die eine exakte Lösung in annehmbarer Zeit berechnen, weshalb auf Heuristiken zurückgegriffen wird, um eine ausreichend gute Lösung in vertretbarer Zeit zu erhalten. Da Optimierungsprobleme im Grunde durch Suchalgorithmen gelöst werden, die aus mehreren Suchergebnissen die beste Lösung auswerten, eignen sich Genetische Algorithmen hervorragend zu deren Lösung. Genetische Algorithmen gehören zwar zu den randomisierten Algorithmen, kombinieren jedoch im Gegensatz zu reinen Zufallsalgorithmen Elemente von gezielter/gerichteter und stochastischer Suchmethoden. Zudem unterhalten Genetische Algorithmen eine ganze Population an möglichen Lösungen, wohingegen alle anderen Lösungsmethoden pro Durchlauf lediglich einen einzelnen Punkt des Suchfeldes betrachten.

1.3.2 Vergleich traditioneller Optimierungsverfahren

Im Verlauf dieses Kapitels sollen die grundsätzlichen Funktionsweisen traditioneller Optimierungsverfahren, sowie die damit verbundenen jeweiligen Einschränkungen deutlich gemacht werden. Dabei wird besonderes Augenmerk auf die Robustheit (siehe Kapitel 1.3.1) der einzelnen Methoden gelegt. Im anschließenden Kapitel werden daraufhin die wesentlichen Unterschiede Genetischer Algorithmen zu den herkömmlichen Verfahren analysiert und deren Vor- und Nachteile aufgezeigt.

Goldberg [4], S. 2, zufolge wird in der aktuellen Literatur zwischen drei Arten von Suchmethoden unterschieden: analytisch (calculus-based), enumerativ und zufällig.
Analytische Suchmethoden unterteilen sich zudem in gerichtete und nicht-gerichtete Methoden.

Eine häufig gebräuchliche gerichtete Methode zum Auffinden von Optima ist die sogenannte „Hillclimbing“ Methode. Diese basiert auf iterativer Verbesserung, wobei pro Iterationsschritt ein neuer Punkt mit besserem Wert als der derzeitige Punkt ausgewählt wird. Wird kein Nachbarspunkt mit besserem Wert gefunden, ist das lokale Optimum erreicht. Die Hillclimbing Methode hängt stark vom gewählten Startpunkt aus und bietet nur Informationen über lokale Optima. Um die Wahrscheinlichkeit, ein globales Optimum zu finden zu erhöhen, wird eine möglichst große Anzahl an Startpunkten gewählt und die gefundenen lokalen Optima miteinander verglichen. ([2], S. 16)

Nicht-gerichtete Methoden hingegen suchen lokale Extremwerte durch Lösen der üblicherweise nicht-linearen Gleichungssysteme, die sich aus dem Gleichsetzen der Steigung der betrachteten Funktion mit 0 ergeben. ([4], S. 2)

Sowohl gerichtete als auch nicht-gerichtete analytische Suchmethoden beherrschen lediglich die Suche nach einem lokalen Optimum. Zudem sind sie nur in Funktionen mit glattem Verlauf sinnvoll einsetzbar. In der Realität sind Funktionen jedoch häufig von Unterbrechungen und „Rauschen“ geprägt, wie in Abbildung 3 ersichtlich. Daher, und aufgrund ihrer Beschränkung auf das Finden des lokalen Optimums, scheiden analytische Methoden zur Lösung von Optimierungsproblemen aus - sie sind in bestimmten Bereichen unzureichend robust. ([4], S. 3 f.)

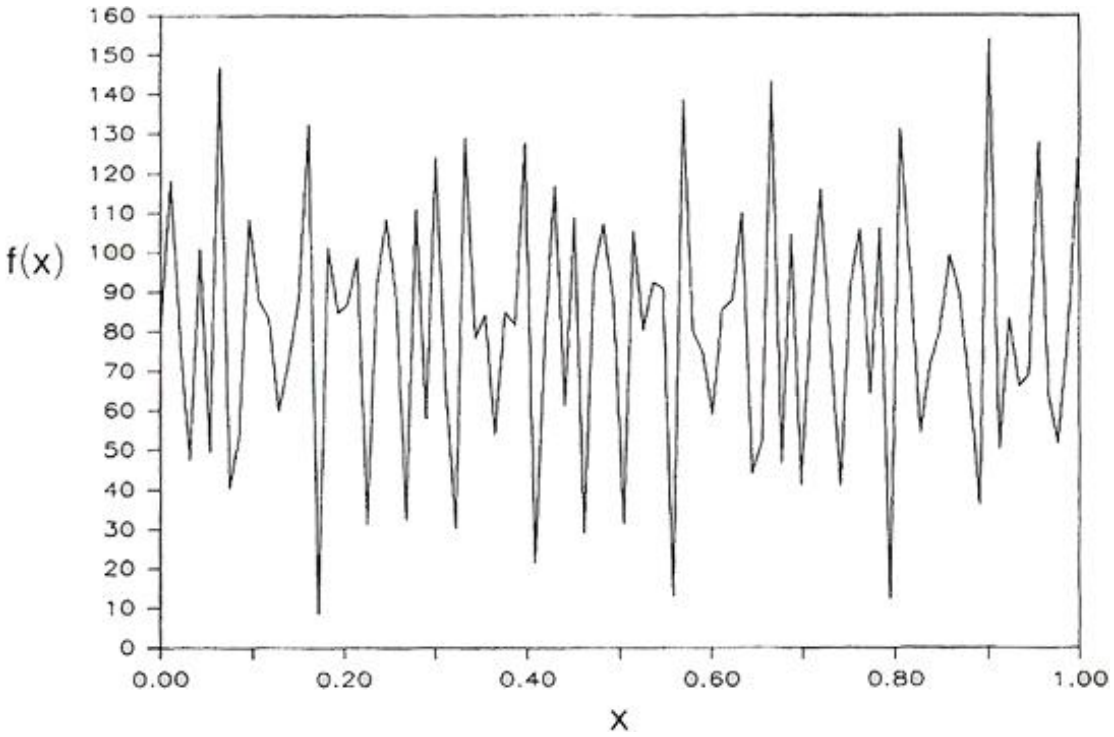

Abbildung 3: "Noisy Function" (Quelle [4], S. 4, Figure 1.3)

Enumerative Methoden basieren auf der Idee, aus einem begrenzten Suchfeld Funktionswerte für jeden Punkt im Suchfeld nacheinander zu betrachten und dadurch das globale Optimum zu finden. Trotz ihrer Einfachheit sind enumerative Methoden äußerst ineffizient und daher nicht besonders robust. Selbst das hochangepriesene Enumerierungsverfahren der Dynamischen Programmierung bricht bereits bei Problemen moderater Größe und Komplexität zusammen. ([4], S.4 f.)

Zufallsverfahren haben aufgrund der Nachteile von analytischen und Enumerationsverfahren zunehmend an Bedeutung gewonnen. Goldberg [4], S. 5, weist jedoch darauf hin, dass diese über längere Zeit gesehen an denselben Mängeln leiden wie enumerative Methoden. Allerdings muss hierbei zwischen strikten Zufallsverfahren und randomisierten Verfahren, zu denen auch die Genetischen Algorithmen gehören, unterschieden werden. Randomisierte Verfahren implizieren nicht zwangsläufig eine richtungslose Suche, sondern kombinieren viel mehr zufällige Auswahl mit gerichteter Suche. Michalewicz [2], S. 17, spricht sogar davon, dass Genetische Algorithmen eine multi-gerichtete Suche vollziehen, da sie eine ganze Population an potenziellen Lösungen bereithalten, wobei jedes Individuum eine eigene Richtung einschlagen kann und der Informationsaustausch zwischen den einzelnen Individuen (durch Crossover Kombinationen zwischen den fittesten Individuen [2], S. 26) bezüglich ihrer Richtung ermutigt wird.

Zusammenfassend betrachtet erzielen sowohl analytische als auch enumerative Methoden in spezialisierten Problemstellungen gute Ergebnisse, sind jedoch für den Einsatz über eine Vielzahl an Domänen hinweg nicht robust genug. Laut Goldberg [4], S.5 f., wäre es jedoch wünschenswert, eine Methode zu finden, die auf Kosten geringerer Spitzenperformance in ganz bestimmten Fragestellungen einen relativ hohen Performancelevel in einem breiten Spektrum an Problemen anbietet. Er betont weiters, dass in Optimierungsverfahren oft nur die Frage gestellt wird, ob ein Optimum gefunden wird, aber der Weg dorthin unbeachtet bleibt. Dieser Gedankenansatz entspringt der Analytik, während in der Natur weniger ein optimales Ergebnis, sondern viel mehr die ständige Verbesserung im Vordergrund steht, um ein ausreichend gutes Ergebnis zu erzielen. Ebenso ist in vielen komplexen Systemen das tatsächliche Erreichen eines Optimums viel weniger von Bedeutung als ein effizientes Vorgehen in der Annäherung an eben dieses. Und genau in diese Denkweise fallen die Genetischen Algorithmen.

1.3.3 Unterschiede Genetischer Algorithmen zu traditionellen Optimierungsverfahren

Gerade jene Eigenschaften, die Genetische Algorithmen von traditionellen Optimierungsmethoden unterscheiden, ermöglichen es ihnen, für ein breites Spektrum an komplexen Problemstellungen gute Näherungsergebnisse in akzeptabler Zeit zu liefern. Goldberg [4], S. 7, spricht von vier wesentlichen Unterschieden der Genetischen Algorithmen zu traditionellen Such- und Optimierungsmethoden:

1. Genetische Algorithmen arbeiten mit einer Kodierung des Parametersets und nicht mit den Parametern selbst.
2. Genetische Algorithmen setzen in ihrer Suche eine ganze Population von Ausgangspunkten, und nicht bloß einen einzigen Ausgangspunkt ein.
3. Genetische Algorithmen greifen nicht auf Ableitungen oder anderes zusätzliches Wissen zurück, sondern bedienen sich allein der Ergebnisse (payoff) einer Zielfunktion.
4. Genetische Algorithmen verwenden probabilistische anstelle deterministischer Übergangsregeln.

Auf die Vorteile, die sich aus diesen vier Unterscheidungsmerkmalen ergeben, geht Goldberg in [4], S. 8 ff., im Detail ein. Zusammenfassend sind folgende Punkte von entscheidender Bedeutung:

1. Durch die Kodierung des Parametersets sind Genetische Algorithmen in der Lage, Kodierungsähnlichkeiten auf eine sehr allgemeine Art und Weise auszunützen. Sie werden dadurch unabhängig von den Beschränkungen anderer Verfahren, wie etwa Stetigkeit, der Existenz von Ableitungen, oder Unimodalität (das Vorliegen von nur einem lokalen Extremwert [9]).
2. Der Einsatz einer gut angepassten, vielfältigen Population anstelle eines einzigen Ausgangspunktes in der Suche nach einem Optimum erhöht die Robustheit sowie in weiterer Folge die Wahrscheinlichkeit der Annäherung an einen optimalen globalen Wert, und ist die Basis der außergewöhnlich hohen Parallelisierbarkeit Genetischer Algorithmen.
3. Viele Suchmethoden setzen Zusatzinformation zur ordnungsgemäßen Funktionsweise voraus - so benötigen etwa die in Kapitel 1.3.2 erläuterten analytischen Verfahren die Ableitung der betrachteten Funktion zum Finden eines lokalen Optimums. Genetische Algorithmen hingegen sind selbst blind. Um eine effektive Suche nach immer besseren Strukturen auszuführen benötigen sie lediglich die Ergebnisse der Auswertung individueller Zeichenketten (Individuen) durch die Zielfunktion. Sie können daher in einem breiten Spektrum an Problemstellungen eingesetzt werden.
4. Die Verwendung probabilistischer Verfahren in Zustandsübergängen geht über eine simple zufällige Suche hinaus: Es wird zufällige Auswahl eingesetzt, um die Suche in Suchregionen mit einer wahrscheinlichen Verbesserung zu führen.

Zusammen verleihen diese vier Unterscheidungsmerkmale den Genetischen Algorithmen gegenüber den traditionellen Optimierungsverfahren eine höhere Robustheit.

Eine weitere Möglichkeit der Klassifikation bietet Michalewicz [2], S. 15, anhand der zwei, anscheinend widersprüchlichen, möglichen Ziele eines Suchalgorithmus: der Ausnutzung einer optimalen Lösung und der Erforschung des Suchraumes.
Während sich etwa die „Hillclimbing“ Methode voll auf die Ausnutzung einer optimalen Lösung für eine weitere Verbesserung konzentriert und dabei die Erforschung des Suchraums vernachlässigt, wenden Suchalgorithmen, die auf dem Zufallsprinzip basieren, die exakt gegensätzliche Strategie an: sie stellen die Erforschung des Suchraums auf Kosten der Ausnutzung einer optimalen Lösung für weitere Verbesserung in den Vordergrund.
Genetische Algorithmen wiederum finden eine bemerkenswerte Balance zwischen den beiden Suchstrategien, wodurch sie sich als „Allzwecksuchmethode“ auszeichnen können.

Jedoch hat die Anwendung Genetischer Algorithmen zur Problemoptimierung auch Kehrseiten, mit welchen sich Michalewicz [2] näher beschäftigt. Er untersucht dazu die Anwendbarkeit von Genetischen Algorithmen auf die Lösung vielfältiger Problemstellungen.

Michalewicz [2], S. 30, kommt zu dem Schluss, dass die Schwierigkeit der Anwendung Genetischer Algorithmen zu einem großen Teil davon abhängt, ob die Evaluierungsfunktion der Problemstellung klar definiert ist. Bei der Lösung der boolschen Form des Erfüllbarkeitsproblems der Aussagenlogik (SAT problem) etwa ist die Vorgehensweise zur Wahl einer Evaluierungsfunktion alles andere als offensichtlich.

Ein weiterer erschwerender Faktor ist die Frage, wie mit Einschränkungen von Problemen umgegangen werden soll. Zum Beispiel besitzt das Handelsreisenden Problem [2], S. 25 f., die Einschränkung, dass jede Stadt der bereisten Route nur einmal aufgesucht werden soll. Nun werden die unterschiedlichen Lösungsmöglichkeiten bei Genetischen Algorithmen (Individuen) als binäre Zeichenketten abgebildet, und eine spezifische Lösung des Handelsreisenden Problems ist eine Folge von einzigartigen Städten. Daraus folgt, dass die einzelnen Städte selbst durch binäre Zeichenketten kodiert werden müssen, und sich ein Individuum aus der Verkettung dieser kodierten Städterepräsentationen ergibt.
Zur Veranschaulichung wird von den Städten A (100), B (101), C (111) ausgegangen, die in der Reihenfolge ABC zu folgender binärer Repräsentation eines Individuums führen: 100.101.111. Durch Anwendung von Mutation und Crossover Operatoren kann es vorkommen, dass der binäre String eines Individuums in einen ungültigen Zustand versetzt wird. So kann in etwa die Einschränkung, dass jede Stadt nur einmal vorkommen soll, verletzt werden (111.101.111) oder die resultierende Binärdarstellung kann nicht in gültige Städte rücküberführt werden (010.111.110).
Zur Sicherstellung eines gültigen Zustands kann beispielsweise der Crossover Operator so modifiziert werden, dass seine Anwendung nur weitere gültige Zustände zulässt und somit die Bildung illegaler Individuen verhindert wird.

Resümierend kann gesagt werden, dass Genetische Algorithmen zwar unter Umständen einen höheren Adaptionsaufwand an das jeweilige Optimierungsproblem fordern als traditionelle Optimierungsverfahren, dafür jedoch in der Ausführung robuster und breiter einsetzbar sind.

1.3.4 Anwendungsmöglichkeiten

Nachdem die vorausgehenden Kapitel die Gründe und Vorteile dargestellt haben, warum sich Genetische Algorithmen hervorragend zur Lösung einer großen Vielfalt an Problemen eignen, sollen in diesem Kapitel Beispiele praktischer Anwendung von Genetischen Algorithmen aufgezeigt werden.

Eine kleine Auswahl an Anwendungen beschreibt Mitchell [1], S. 4 f., im Detail:

Beim sogenannten “Protein Engineering” beweisen Genetische Algorithmen ihre Stärke als Suchalgorithmen. Hierbei werden aus einer riesigen Menge an möglichen Aminosäuren Sequenzen Proteine mit ganz spezifischen Eigenschaften gesucht.
Die Robustheit und Anpassungsfähigkeit der Genetischen Algorithmen an ihre Umwelt machen sich Ingenieure in der Kontrolle von Robotern zugute. Diese Roboter müssen Aufgaben in einer sich ständig ändernden Umwelt erfüllen.
Ein weiterer Bereich, in dem die Anwendung Genetischer Algorithmen kaum mehr wegzudenken ist, stellt die Entwicklung künstlicher Intelligenz (KI) dar. In der heutigen Zeit sind KI Entwickler zu der Überzeugung gekommen, dass die der KI unterliegenden Regeln einen viel zu hohen Grad an Komplexität erreichen, um von den Entwicklern manuell in einer „top-down fashion“ fest eingebaut zu werden. Stattdessen wird heutzutage das „bottom up“ Paradigma angewandt. Dabei werden durch den Entwickler nur mehr ein paar sehr simple Regeln, bei den Genetischen Algorithmen in Form von natürlicher Selektion sowie Crossover und Mutation, definiert. Die KI entwickelt sich dann aus der massiv parallelen Anwendung und Verknüpfung dieser simplen Regeln.

Im Allgemeinen eignen sich Genetische Algorithmen besonders für Such- und Optimierungsprobleme, die aufgrund eines sehr großen Suchraums mit herkömmlichen Verfahren nicht in vernünftiger Zeit zu bewältigen wären. Dazu zählen unter anderem das Design von Leiterplatten und integrierten Schaltkreisen, Scheduling, Adaptive Control, Game Playing, kognitive Modellierung, Logistik Probleme (darunter das Rucksackproblem und das Handelsreisenden Problem) und die Optimierung von Datenbankabfragen. ([2], S. 13 ff.)

Aber auch abseits klassischer numerischer und kombinatorischer Optimierungsprobleme gibt es eine große Palette an Anwendungsmöglichkeiten für Genetische Algorithmen ([1], S. 16):

Im Bereich automatischer Programmierung werden Genetische Algorithmen eingesetzt, um Computerprogramme für spezifische Aufgaben zu entwickeln (Genetische Programmierung), oder um Rechenstrukturen wie zelluläre Automaten und Sortiernetzwerke zu entwerfen.
Für maschinelles Lernen gibt es zahlreiche Aufgaben, die durch Genetische Algorithmen wahrgenommen werden. Dazu zählen Klassifizierungs- und Voraussageaufgaben, oder die Entwicklung bestimmter Aspekte maschinellen Lernens wie die Gewichtung neuronaler Netzwerke, Regeln für das Lernen klassifizierter Systeme oder symbolischer Produktionssysteme und Sensoren für Roboter.
In der Wirtschaft dienen Genetische Algorithmen zum Beispiel zur Modellierung von Innovationsprozessen, der Entwicklung von Bieterstrategien, und der Entstehung neuer Märkte.
Schließlich zählen auch die Forschung in Biologie und Soziologie zu großen Einsatzgebieten Genetischer Algorithmen. Die evolutionären Aspekte sozialer Systeme, wie etwa das Sozialverhalten in Insektenkolonien, kann durch Genetische Algorithmen ebenso untersucht werden wie zahlreiche Fragestellungen der Ökologie, also der Interaktion zwischen

Organismen und ihrer Umwelt. Auch Immunsysteme werden häufig mit Hilfe Genetischer Algorithmen erforscht und die Genetik, nach deren Konzepten die Genetischen Algorithmen aufgebaut sind, gilt als Paradedisziplin für deren Anwendung.

Der erfolgreiche Einsatz von Genetischen Algorithmen in den erwähnten und anderen Disziplinen hat diesen zu einer rasch steigenden Beliebtheit verholfen, so dass das Feld der Genetischen Algorithmen heute eine eigene Teildisziplin der Informatik mit zahlreichen Konferenzen, Journalen und einer eigenen wissenschaftlichen Gesellschaft ist. ([1], S. 16)
Eine Publikationsreihe, die sich mit aktuellen Anwendungsmöglichkeiten von Genetischer Programmierung beschäftigt, ist im Jahr 2015 in zwölfter Ausgabe erschienen: „Genetic Programming Theory and Practice XII“ [10], herausgegeben von Rick Riolo, Mark Kotanchek und Jason H. Moore.

1.4 Vorarbeiten

Im Folgenden werden einige der Publikationen aufgeführt, die als Grundlage zur Recherche verwendet wurden und auf denen diese Arbeit aufbaut. Diese stellen einen Abriss der wichtigsten zum Thema verfügbaren Literatur da und sollen dem Leser/der Leserin als Nachschlagewerke zur genaueren Vertiefung in die Genetische Programmierung dienen.

Als Ausgangsbasis empfiehlt sich „A Field Guide to Programming“ [3] von Riccardo Polli, William B. Langdon und Nicholas F. McPhee. Dieser bietet eine Momentaufnahme des aktuellen Forschungsstandes, gewährt einen Überblick über die Grundlagen, eine Zusammenfassung von wichtigen Arbeiten und betrachtet neue Richtungen und Anwendungsmöglichkeiten auf dem Gebiet der Genetischen Programmierung.
Zudem wird eine umfangreiche Liste an Büchern, Videos, Journals, Konferenzen und Online Quellen zur weiteren Recherche zur Verfügung gestellt.

John R. Koza ist ein Pionier in der Optimierung komplexer Probleme mit Hilfe von Genetischer Programmierung [11]. Unter seinen zahlreichen Publikation gilt „On the Programming of Computers by Means of Natural Selection“ [5] als eines der ausführlichsten Referenzwerke der Genetischen Programmierung.

Ein weiteres Standardwerk im Bereich der Genetischen Programmierung ist „An Introduction to Genetic Algorithms“ [1] von Melanie Mitchell. Die Autorin gewährt eine sanfte Einführung in das Thema Genetische Algorithmen und bietet zahlreiche theoretische und praktische Übungen um das Verständnis des Lesers / der Leserin zu stärken.

Auf die Anwendung von Genetischen Algorithmen zur Numerischen Optimierung spezialisiert sich Zbigniew Michalewicz mit „Genetic Algorithms + Data Structures = Evolution Programs“ [2]. Auch ohne spezielles Vorwissen gibt das Buch eine Einführung in Genetische Algorithmen und widmet sich in einem weiteren Abschnitt Evolutionärer Programmierung.

Eine der meistverkauften [12] Einführungen in Genetische Algorithmen ist „Genetic Algorithms in Search, Optimization and Machine Learning“ [4] von David Goldberg, einem Schüler John Hollands, dem Vater der Genetischen Algorithmen. Dieses Werk deckt alle wichtigen Bereiche des Gebiets ab und bietet einem Neuling mit Hintergrundwissen in Informatik genug Informationen, um selbst einen genetischen Algorithmus zu implementieren.

Die Recherchen zum Rucksack Problem basieren zum größten Teil auf „Knapsack Problems“ [7] von Hans Kellerer, Ulrich Pferschy und David Pilsinger. In diesem Buch werden die unterschiedlichsten Herangehensweisen zur Lösung des Rucksack Problems, sowie die Komplexitäten der jeweils angewandten Algorithmen, beschrieben.

Zur Erklärung der wesentlichen Konzepte der Komplexitätstheorie erwies sich „Introduction to Algorithms“ [8] von Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest sowie Clifford Stein als wahre Fundgrube. Dabei ist das Kapitel „NP-Completeness“ nur ein kleiner Teil dieses äußerst umfangreichen Werkes, das sich mit Algorithmen und Datenstrukturen jeglicher Art auseinandersetzt.

Abschließend sei an dieser Stelle noch die Forschungsarbeit „Solving the 0-1 Knapsack Problem with Genetic Algorithms“ [6] von Maya Hristakeva und Dipti Shrestha am Computer Science Department des Simpson College als konkrete Grundlage für diese Arbeit genannt.

Weitere Literaturquellen sind dem Literaturverzeichnis zu entnehmen.

1.5 Ziel dieser Arbeit

Durch die Anwendung Genetischer Algorithmen zur annähernden Lösung des 0-1 Rucksackproblems sollen die in Kapitel 1.3.3 besprochenen theoretischen Vorteile Genetischer Algorithmen zur Lösung komplexer Optimierungsprobleme in der Praxis belegt werden. Gleichzeitig wird durch die praktische Umsetzung der theoretischen Konzepte Genetischer Algorithmen deren Implementierung anschaulich dargestellt.
Darüber hinaus erfolgt ein Vergleich mit den Resultaten ausgewählter traditioneller Methoden, die zur Lösung des 0-1 Rucksackproblems eingesetzt werden. Dabei werden die eingesetzten Genetischen Algorithmen auf ihre Komplexität untersucht und es erfolgt der Beweis, dass Genetische Algorithmen das binäre Rucksackproblem tatsächlich in polynomieller Zeit annähernd lösen können.

2 Lösung des Rucksackproblems

2.1 Lösungsstrategie

Nachdem in der Einführung dieser Arbeit die grundlegenden theoretischen Konzepte erläutert und das Ziel dieser Arbeit definiert wurden, erfolgt in diesem Kapitel zur Erfüllung der Aufgabenstellung ein experimenteller Vergleich Genetischer Algorithmen und traditioneller Verfahren zur Lösung des binären Rucksackproblems. Als traditionelle Verfahren kommen der Greedy Algorithmus und Dynamische Programmierung zum Einsatz.
Neben einer detaillierten Beschreibung der Umsetzung der einzelnen Verfahren werden unter anderem die „Benefits“ (Nutzwerte) und Gewichte, Laufzeiten, Effektivität und Effizienz der angewandten Algorithmen erhoben. Zusätzlich erfolgt eine Ermittlung der Komplexität der unterschiedlichen Algorithmen.
Die ermittelten Messdaten dienen dazu, die wesentlichen Unterschiede zwischen Genetischen Algorithmen und traditionellen Verfahren anschaulich zu belegen und die Einsatzmöglichkeiten der angewandten Methoden abzugrenzen. Jedoch erfolgt in diesem Kapitel eine rein sachliche Präsentation der Daten. Eine Analyse und Deutung der Ergebnisse findet in Kapitel 3 statt.

Alle Algorithmen wurden eigenständig in C++ im Rahmen einer Kommandozeilenanwendung implementiert. Als Eingabedaten dienten die Datensätze P06, P07, sowie zwei modifizierte Varianten von P08 (beide jeweils mit um den Faktor 100 reduzierter Kapazität und Fitness, der zweite Datensatz darüber hinaus mit einer fünffachen Anzahl an Elementen) der Website „KNAPSACK_01 - Data for the 01 Knapsack Problem“ [13].
Eine Parametrisierung der verwendeten Algorithmen ist über Textdateien möglich.
Die Anwendung in Form der ausführbaren Datei „KnapsackProblem.exe“ liegt der Arbeit bei, ebenso wie der Quellcode, sowie Eingabedaten und Resultate im .csv/.xlsx Format.

Zur Repräsentation aller im Inputfile enthaltenen Datensätze wird im Programm ein Array von Objekten der Klasse „Item“ angelegt, die den Namen, das Gewicht und den „Benefit“ eines Datensatzes als Entität abspeichert. Dieses „mItems“ genannte Array dient als gemeinsame Basis aller eingesetzten Algorithmen, auch wenn sich diese ansonsten grundlegend voneinander unterscheiden. Die spezifischen Lösungsansätze der einzelnen Algorithmen werden in den folgenden Unterkapiteln erläutert.

2.2 Implementierung des Genetischen Algorithmus

Bei der Implementierung des Genetischen Algorithmus wurde besonderer Wert auf Parallelisierbarkeit gelegt. Es kommt ein Threadpool zur Anwendung, dessen Größe über Eingabeparameter festgelegt werden kann. Die Threads im Threadpool werden einmalig erzeugt und laufen dann permanent. Sie führen Operationen auf klar abgegrenzten Teilbereichen (Arrays von Chromosomen) einer Population aus. Ein Hauptthread übernimmt nicht parallelisierbare Aufgaben, steuert den strukturellen Ablauf und die Synchronisation der einzelnen Threads.

Ein Problem bei der Parallelisierung stellt die Generierung von Pseudozufallszahlen dar. Die in C++ standardmäßige Methode rand(), um Pseudozufallszahlen zu generieren, ist nicht thread-safe und liefert daher bei „gleichzeitigem“ Aufruf von mehreren Threads dieselben Werte zurück. Um dieses Problem zu umgehen, wird jedem Thread ein eigener Pseudozufallszahlengenerator, jeweils mit einem unterschiedlichen Seed, zugewiesen. Wenn ein Thread einen pseudo zufällig generierten Wert benötigt, greift er auf eigens implementierte Funktionen (für bool, unsigned int, double) zu, denen er seine ID innerhalb des Threadpools übergibt. Unter dieser ID wird dann aus einem Array der jeweilige Pseudozufallszahlengenerator mit der Erzeugung einer Zufallszahl beauftragt. Auf diese Weise konnte auch eine besonders hohe Streuung der Zufallszahlen erreicht werden.

Alle Daten, auf denen der Genetische Algorithmus operiert, sind in einem „Population“ struct abgespeichert. Diese lassen sich grob in zwei Bereiche unterteilen: Auf der einen Seite stehen die als Array von binären Zeichenketten hinterlegten Chromosomen und über deren Indizes verknüpfte Informationen - beispielsweise werden die Fitnesswerte der Chromosomen in einem eigenen Array abgespeichert, und sowohl auf die Fitnesswerte als auch auf die zugehörigen Chromosomen selbst kann über idente Indizes zugegriffen werden. Im Schritt der Selektion hingegen werden die für die Selektion ausgewählten Chromosomen durch ein Array von Indizes der ursprünglichen Chromosomen repräsentiert.
Auf der anderen Seite sind in dem struct auch Metainformationen über die Population hinterlegt. Dazu zählen etwa die durchschnittliche Fitness, die höchsten und niedrigsten Fitnesswerte der aktuellen Generation, sowie das fitteste Chromosom aller bisherigen Generationen.

Vor Ablauf der eigentlichen Logik liest der implementierte Genetische Algorithmus Parameter wie die Populationsgröße, die Wahrscheinlichkeiten für Crossover und Mutation, die Anzahl an gewünschten Iterationen und eingesetzten Threads, sowie eine Abbruchbedingung (Anteil identer Chromosomen in einer Generation), ein und initialisiert den Threadpool.

Die Struktur des Genetischen Algorithmus lautet:

- Generiere anfängliche Population (zufallsbasiert)
- Je Iteration für Anzahl an gewünschten Iterationen:
  - Berechne Fitness aller Chromosomen
  - Beende, falls definierter Anteil aller Chromosomen dieselbe Fitness besitzt
  - Selektion
  - Crossover
  - Mutation

Für die Generierung der anfänglichen Population von Chromosomen wird pro Bit in der binären Zeichenkette die „Münzwurfmethode“ angewandt. Ein Wert von true bedeutet, dass das Element am jeweiligen Index aus dem Array aller verfügbaren Items „mItems“ in den Rucksack gepackt wird, bei false wird es dagegen nicht mitgeführt. Da jedes Bit in der binären Zeichenkette eines Chromosoms ein Item repräsentiert, ergibt sich die Länge eines Chromosoms aus der Anzahl aller verfügbaren Items.

Bei der Berechnung der Fitness eines Chromosoms kann es gerade in den ersten Generationen vorkommen, dass ein großer Teil der Chromosomen die maximale Kapazität des Rucksacks übersteigt und daher ungültige Lösungen produziert. Um selbst in dem Fall, dass in einer Generation kein einziges Chromosom unterhalb der zulässigen Kapazität bleibt, einen gewichtbaren Vergleichswert zu erhalten, werden daher auch negative Fitnesswerte eingesetzt. Die Fitness eines Chromosoms berechnet sich folgendermaßen:
Für alle Gegenstände, die durch ein true-Bit im Chromosom als im Rucksack befindlich markiert wurden, wird die Summe aller einzelnen Nutzwerte als gesamte Fitness des Chromosoms definiert. Sobald jedoch durch Hinzufügen des Gewichts eines Gegenstandes zum Gesamtgewicht die maximale Kapazität des Rucksacks überschritten würde, wird die Fitnesssumme auf 0 zurückgesetzt und für jeden „überpackten“ Gegenstand um den jeweiligen Nutzwert vermindert. Somit erhalten Gegenstandskombinationen, die die Rucksackkapazität deutlich übersteigen, eine höhere negative Fitness als jene, deren Gesamtgewicht nur knapp über der zulässigen Obergrenze liegt. Dadurch können letztere in der gewichteten Selektion bevorzugt werden in der Hoffnung, dass auf diese Weise schneller Chromosomen gefunden werden, die unterhalb der zulässigen Kapazität bleiben und somit gültige Lösungen darstellen.

Zur Überprüfung der Abbruchbedingung werden die Fitnesswerte aller Chromosomen in einer Map als Key abgespeichert. Der Value des Entry Paares in der Map ist anfänglich 1 und wird für alle weiteren identen Fitnesswerte um 1 erhöht. Aus einem in den Parametern definierten Prozentsatz identer Chromosomen und der Anzahl aller Chromosomen einer Population berechnet sich ein Schwellwert für den Abbruch des Algorithmus. Sobald das Inkrementieren eines Map Entries diesen überschreitet, beendet der Algorithmus vorzeitig.

Die Selektion der zur Reproduktion zugelassenen Chromosomen erfolgt mittels eines gewichteten Roulette Rades. Die Größe eines Slots ist dabei die Differenz aus der Fitness eines Chromosoms und der niedrigsten Fitness (einschließlich negativer Fitnesswerte) unter allen Chromosomen einer Generation. Das Roulette Rad wird sooft gedreht, wie es Chromosomen in einer Population gibt. Die Größe der Population über zwei Generationen hinweg bleibt also immer konstant. Pro Drehung wird eine Zufallszahl im Bereich [0, Summe aller Slotgrößen] ermittelt und jenes Chromosom, das den „Gewinnerslot" besetzt, für die Reproduktion eines neuen Chromosoms herangezogen. Dadurch sind solche Chromosomen bei der Reproduktion bevorzugt, welche eine höhere Fitness aufweisen, da diese auch einen größeren Slot am Roulette Rad einnehmen.

Im Zuge des Crossover werden für alle Chromosomen der in der Selektion neu erstellten Generation Zufallswahrscheinlichkeiten berechnet. Jene Chromosomen, deren Zufallswert im Bereich [0,1] kleiner ist als die Crossoverwahrscheinlichkeit p_c, werden einem Pool hinzugefügt. Aus diesem Pool werden zufällige Chromosomenpaare gebildet, wobei jedes Chromosom genau einmal, also ohne Zurücklegen, gezogen wird. Sollte die Anzahl der Chromosomen im Pool nicht gerade sein, wird ein weiteres zufälliges Chromosom aus der Population als Paarungspartner ausgewählt. Für jedes Chromosomenpaar werden die binären Zeichenketten an einem zufällig berechneten Index im Bereich [1, l - 1] (l = Chromosomenlänge) durchtrennt und die hinteren, abgetrennten Teile der beiden beteiligten Chromosomen vertauscht. Dahinter steht die Idee, dass mit ein wenig Glück der neu eingesetzte Teil eines Chromosoms eine höhere Teilfitness produziert als der bisherige. Dadurch kann selbst eine Population, in der jedes Chromosom ungültige, die zulässige Kapazität übersteigende Lösungen produziert, durch Zusammensetzung zweier „leichtgewichtigerer" Teilketten schließlich Chromosomen mit gültigen Lösungen hervorbringen.

Dem eventuellen Verlust bestimmter Geninformationen (binärer Teilstrings) aufgrund eines allzu übereifrigen Einsatzes von Selektion und Crossover soll der Mutationsoperator entgegenwirken. Für jedes Chromosom in der Population wird über alle Bit der binären Zeichenkette iteriert, und ein zufälliger Wert im Bereich [0,1] erzeugt. Liegt dieser Wert unterhalb der Mutationswahrscheinlichkeit p_m, wird der binäre Wert des Bits umgekehrt.

In ihrer Gesamtheit führen die drei Operatoren Selektion, Crossover und Mutation, über mehrere Generationen hinweg zu einem Anstieg der durchschnittlichen Fitness der Population. Als Endergebnis wird jenes Chromosom betrachtet, das über alle Generationen hinweg die höchste Fitness aufweisen konnte.

2.2.1 Komplexität des Genetischen Algorithmus

Abschließend soll noch die Komplexität des implementierten Genetischen Algorithmus erhoben werden. Da für die Berechnung der Komplexität eines Programmes in Big O Notation einzig der komplexeste Teil eines Programmes ($O(n^2) + O(n) = O(n^2)$) ausschlaggebend ist, ergibt sich die Komplexität des Genetischen Algorithmus aus seinem komplexesten Teilschritt.

Für die Erstellung der anfänglichen Population, ebenso wie für die Berechnung der Fitness und die Mutation, werden alle Bits/Items n aller Chromosomen der Population (Populationsgröße s) einmal durchlaufen. Die Komplexität hierfür wäre also $O(n * s)$. In der Big O Notation soll die Laufzeit jedoch einzig in Abhängigkeit von der Eingabe eines Programmes untersucht werden. Die Anzahl an Chromosomen einer Population stellt einen konstanten Faktor dar und ist nicht von der Eingabe abhängig. Daher wird die Komplexität $O(n * s)$ letztendlich als $O(n)$ angeschrieben.
Bei der Selektion werden alle Chromosomen s zur Konstruktion des Roulette Rades betrachtet und zum Abgleich der generierten Zufallszahl mit einem Slot jeweils maximal alle Slots s des Rades einmal abgeglichen. Daraus ergibt sich eine Komplexität $O(s^2) \rightarrow O(1)$.
Die Abbruchbedingungsfunktion durchläuft alle Chromosomen der Population einmal, also in $O(s) \rightarrow O(1)$.
Das Crossover findet in mehreren Stufen statt: die Auswahl des Crossover Pools erfolgt durch Iteration über alle Chromosome der Population in $O(s)$. Im Zuge der Rekombination der Teilstrings müssen maximal $n - 1$ Bits für jedes Chromosomenpaar im Crossover Pool (s_p = Poolsize) überschrieben werden. Hierfür wird als Komplexität $O(n * s_p) \rightarrow O(n)$ angegeben. Gesamt ergibt sich $O(s) + O(n) = O\ (s + n) = O(n)$.

Der komplexeste Teilschritt, und damit auch der Genetische Algorithmus selbst, besitzen also eine Komplexität von $O(n)$.

2.3 Traditionelle Lösungsverfahren

2.3.1 Greedy Algorithmus

Der Greedy Algorithmus sortiert das Gegenstände-Array nach der Effizienz als Ergebnis von Nutzwert geteilt durch Gewicht. Nun werden solange Gegenstände aus dem Array in den Rucksack gepackt, bis das Hinzufügen eines weiteren Gegenstandes die Kapazität des Rucksacks übersteigen würde.

Die Komplexität des Packvorgangs beträgt, da maximal jeder Gegenstand einmal betrachtet wird, $O(n)$. Insofern das angewandte Sortierverfahren die Komplexität $O(n)$ übersteigt, hängt die Gesamtkomplexität daher vom Sortieralgorithmus ab. Für diese Arbeit wurde Quicksort als Sortieralgorithmus ausgewählt. Dessen Komplexität, und somit die Gesamtkomplexität des Greedy Algorithmus, ist $O(n \log n)$.

2.3.2 Dynamische Programmierung

Kellerer, Pferschy und Pisinger, [7], S. 20, zufolge wird Dynamische Programmierung häufig eingesetzt, um optimale Lösungen zu berechnen, die sich in eine Kombination von optimalen Teillösungen zerlegen lassen. Der Algorithmus basiert auf folgendem Grundgedanken:
Entfernt man aus der optimalen Lösungsmenge N des Rucksackproblems einen Gegenstand r, so muss die verbleibende Lösungsmenge eine optimale Lösung des Teilproblems, definiert durch die Kapazität $c' = c - w_r$ und die Gegenstandsmenge $N \setminus \{r\}$, sein.
Bei der Dynamischen Programmierung wird dieses Prinzip umgekehrt angewandt. Es wird davon ausgegangen, dass bereits optimale Lösungen für eine Teilmenge sowie alle Kapazitäten $\sum c^* < c'$ kleiner der der Teilmenge berechnet wurden. Bei Einfügen eines weiteren Gegenstandes wird überprüft, ob sich die optimale Lösung für die neue, erweiterte Teilmenge verändert hat. Diese Überprüfung wird durch den Umstand vereinfacht, dass auf die bereits gelösten Teilprobleme mit geringeren Kapazitäten zugegriffen werden kann. Falls die Überprüfung eine Verbesserung der Lösung ergibt, müssen die optimalen Teillösungen aller kleineren Kapazitäten erneut berechnet werden, damit die Verbesserung auch allen weiteren hinzuzufügenden Gegenständen zur Verfügung steht. Dieser Vorgang wird wiederholt, bis alle Gegenstände berücksichtigt wurden und somit die optimale Lösung des Gesamtproblems ermittelt wurde.

Als Grundlage der in Abbildung 4 dargestellten konkreten Implementierung dient das struct „BenefitSet“. Dieses setzt sich zusammen aus einem Nutzwert „.benefit“, und einem Bitstring für die verwendeten Items „.itemFlags“, welcher einem Chromosom beim Genetischen Algorithmus entspricht. Um die Lösungen aller Teilprobleme zu speichern, wird ein zweidimensionales Array von BenefitSets, „mOptimalSolutions“, erzeugt. Dabei ist die Größe der ersten Dimension durch die Größe des Arrays aller verfügbaren Items, „mItems“, und die der zweiten Dimension durch die Kapazität des Rucksacks, „mCapacity“, definiert.

Das iterative Hinzufügen neuer Items, sowie die Berechnung aller möglichen Teilkapazitäten, wird durch zwei ineinander geschachtelte Schleifen realisiert. Für einen Rucksack ohne Items oder ohne Kapazität werden alle Nutzwerte anfänglich mit 0 initialisiert. Beim Versuch ein neues Item in den Rucksack zu packen, können zwei Fälle eintreten:
Bei Überschreiten der Teilkapazität „d“, oder falls durch Einbeziehung des neuen Items keine Verbesserung der optimalen Teillösung möglich ist, wird die bisherige Teillösung „mOptimalSolutions[i-1][d]“ unverändert übernommen.
Ansonsten wird die aktuelle optimale Teillösung auf den neu berechneten Wert gesetzt, und das neue Item in den Rucksack gepackt, also der Flag am entsprechenden Index des boolschen Arrays auf true gesetzt. Wie bereits erwähnt kann bei der Überprüfung auf bereits berechnete Teillösungen geringerer Teilkapazitäten zurückgegriffen werden. Die als „partialIdx“ bezeichnete Variable gibt den Index jener Teillösung an, zu der das neue Item hinzugefügt werden soll. Diesen erhält man aufgrund der Überlegung, dass durch Hinzufügen des neuen Items r nur mehr eine limitierte Teilkapazität $c' = c - w_r$ für andere Items zur Verfügung steht.

```
for (i = 0; i < mItems.size(); ++i) {
  for (d = 0; d <= mCapacity; ++d) {
    if (i == 0 || d == 0) {
      mOptimalSolutions[i][d].benefit = 0;
    }
    else if (d >= mItems[i - 1]->getWeight()) {
      unsigned int prevTotalBenefit = mOptimalSolutions[i-1][d].benefit;
      unsigned int partialIdx = d - mItems[i - 1]->getWeight();
      unsigned int newTotalBenefit = mItems[i - 1]->getBenefit()
        + mOptimalSolutions[i - 1][partialIdx].benefit;

      if (newTotalBenefit > previousTotalBenefit) {
        mOptimalSolutions[i][d].benefit = newTotalBenefit;
        mOptimalSolutions[i][d].itemFlags =
          mOptimalSolutions[i - 1][partialIdx].itemFlags;
        mOptimalSolutions[i][d].itemFlags[i - 1] = true;
      }
      else {
        mOptimalSolutions[i][d] = mOptimalSolutions[i - 1][d];
      }
    else {
      mOptimalSolutions[i][d] = mOptimalSolutions[i - 1][d];
    }
  }
}
```

Abbildung 4: Implementierung der Dynamischen Programmierung

Abbildung 4 zeigt, dass die Komplexität der Dynamischen Programmierung durch die zwei ineinander geschachtelten Schleifen bestimmt wird. Als Komplexität ergibt sich somit $O(n * c)$. Da die Kapazität c, ebenso wie die Anzahl der verfügbaren Items n, von der Eingabe abhängt, darf diese nicht als konstanter Faktor aus der Berechnung gekürzt werden.

2.4 Gegenüberstellung der Ergebnisse

Alle Messungen wurden auf einem Intel Core i7-4710MQ (4 physikalische Kerne, 8 virtuelle Kerne) unter Windows 7 64bit in „Höchstleistungsmodus“ mit einer Taktung von 2,5 GHz ausgeführt. Es standen 16GB DDR3 RAM Speicher zur Verfügung.

Zum Vergleich der verschiedenen Algorithmen wurden Messreihen auf 4 sich in der Anzahl an Elementen und der Kapazität unterscheidenden Eingabedatensätzen durchgeführt. Es wurden der Nutzwert, das Gewicht, die Laufzeit, die Effektivität und die Effizienz der Lösungen erhoben. Als kleinste zulässige Messeinheit für die Laufzeit wurde 1ms festgelegt. Die Effektivität berechnet sich aus der „Genauigkeit“ des vom jeweiligen Algorithmus erzielten Nutzwerts im Verhältnis zum optimalen Nutzwert in Prozent. Die als Effizienz bezeichnete Messgröße hat hier ausschließlich relativen Vergleichscharakter und wird durch die Formel $\frac{Effektivität}{Laufzeit}$ angegeben. Pro Messreihe wurde das Ergebnis aus dem Mittelwert mehrerer Messungen gebildet. Dabei wurden für den Genetischen Algorithmus jeweils 5 Messungen, und für den Greedy Algorithmus und die Dynamische Programmierung jeweils 3 Messungen ausgeführt, da sich letztere ausschließlich in ihrer Laufzeit geringfügig unterscheiden, während beim Genetischen Algorithmus auch der Fitnesswert und das Gewicht variieren. Die Ergebnisse der Messreihen 1 bis 4, sowie die Anzahl der zugehörigen Eingabeelemente n, die jeweilige Kapazität c und die optimale Lösung o, sind den Tabellen 3 bis 6 zu entnehmen.

Beim Genetischen Algorithmus wurde in der umfangreichsten Messreihe 4 zusätzlich das Verhalten bei Variation der Parameter Populationsgröße p und Mutationswahrscheinlichkeit p_m untersucht. Insofern diese nicht explizit anders angegeben sind, gelten für die Parameter des Genetischen Algorithmus die Standardwerte aus Tabelle 1:

Anzahl an Iterationen (Generationen):	75
Populationsgröße p (Anzahl an Chromosomen):	50
Crossoverwahrscheinlichkeit p_c:	0.7
Mutationswahrscheinlichkeit p_m:	0.001
Prozentsatz identer Chromosomen für Abbruch:	0.9
Anzahl verwendeter Threads:	4

Tabelle 1: Standardparameter für Genetischen Algorithmus

Algorithmus	Nutzwert	Gewicht	Laufzeit (ms)	Effektivität	Effizienz
Greedy Algorithmus	1621	161	1	93.429	93.429
Dynamische Programmierung	1735	169	1	100.000	100.000
Genetischer Algorithmus	1729	169	4	99.666	23.730

Tabelle 2: Messreihe 1 → n = 7, c = 170, o = 1735

Algorithmus	Nutzwert	Gewicht	Laufzeit (ms)	Effektivität	Effizienz
Greedy Algorithmus	1406	7410	1	96.433	96.433
Dynamische Programmierung	1458	7490	21	100.000	4.762
Genetischer Algorithmus	1441	7466	6	98.807	17.036

Tabelle 3: Messreihe 2 → n = 15, c = 7500, o = 1458

Algorithmus	Nutzwert	Gewicht	Laufzeit (ms)	Effektivität	Effizienz
Greedy Algorithmus	123063	63825	1	90.827	90.827
Dynamische Programmierung	135491	64026	310	100.000	0.323
Genetischer Algorithmus	133019	63737	8	98.176	12.587

Tabelle 4: Messreihe 3 → n = 24, c = 64041, o = 135491

Algorithmus	Nutzwert	Gewicht	Laufzeit (ms)	Effektivität	Effizienz
Greedy Algorithmus	474308	249655	1	88.852	88.852
Dynamische Programmierung	533817	249997	13125	100.000	0.008
Genetischer Alg. (p = 50, pm = 0.001)	510006	249588	15	95.539	6.369
Genetischer Alg. (p = 100, pm = 0.001)	511132	249363	24	95.750	4.057
Genetischer Alg. (p = 200, pm = 0.001)	513731	249723	43	96.237	2.259
Genetischer Alg. (p = 200, pm = 0.02)	515348	249331	45	96.540	2.136
Genetischer Alg. (p = 1000, pm = 0.001)	517953	249705	183	97.028	0.531
Genetischer Alg. (p = 1000, pm = 0.02)	516707	249565	187	96.795	0.517

Tabelle 5: Messreihe 4 → n = 120, c = 250000, o = 533817

3 Schlussbetrachtung

3.1 Reflexion der Erkenntnisse

Beim Vergleich der Effektivität der einzelnen Algorithmen in Abbildung 5 garantiert ausschließlich die Dynamische Programmierung, immer die optimale Lösung zu finden. Beim Greedy Algorithmus hängt die Effektivität stark von den eingegebenen Daten ab und lässt sich daher nur schwierig abschätzen. Der Genetische Algorithmus erreicht nur in sehr einfachen Problemstellungen die optimale Lösung, jedoch bleibt die Effektivität auch bei komplexen Problemen in einem vergleichsweise sehr hohen Bereich und weicht nur geringfügig von der optimalen Lösung ab.

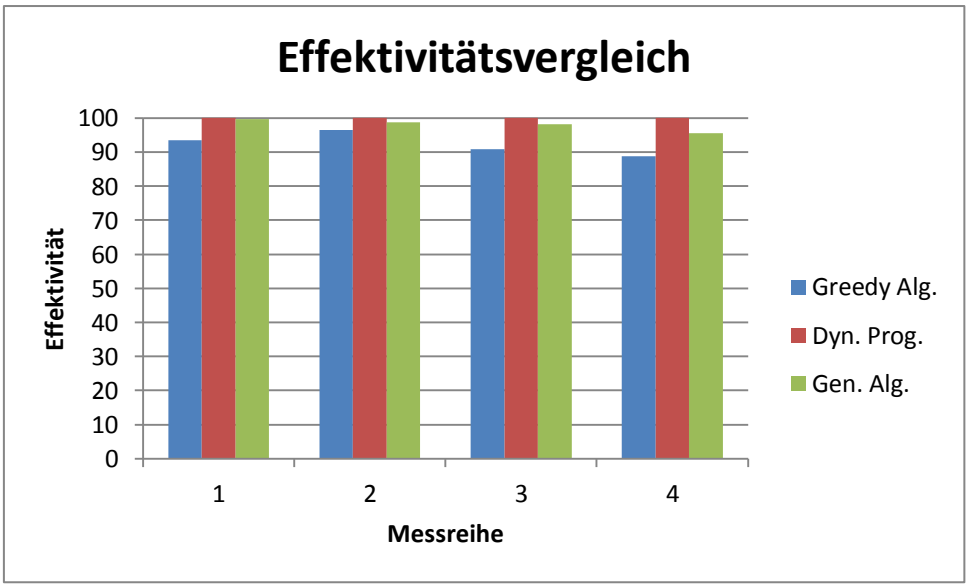

Abbildung 5: Effektivitätsvergleich der eingesetzten Algorithmen

Der Effizienzvergleich in Abbildung 6 fällt für die Dynamische Programmierung ernüchternd aus. Die Komplexität von $O(n * c)$ führt bei umfangreicheren Problemen dazu, dass die Laufzeiten im Vergleich zu Greedy Algorithmus und Genetischem Algorithmus explosionsartig ansteigen. Dadurch sinkt auch die Effizienz auf so kleine Werte, dass diese im Vergleich nicht mehr vernünftig dargestellt werden können. Die Effizienz des Greedy Algorithmus fällt aufgrund der sehr geringen Laufzeit vergleichsweise hoch aus. Der Genetische Algorithmus schließlich liegt mit seiner Effizienz zwischen Greedy Algorithmus und Dynamischer Programmierung, ist jedoch, auch wenn dies in der graphischen Darstellung nicht so gut ersichtlich wird, deutlich näher an jener des Greedy Algorithmus angesiedelt und fällt auch bei weitem nicht so drastisch ab wie jene der Dynamischen Programmierung.

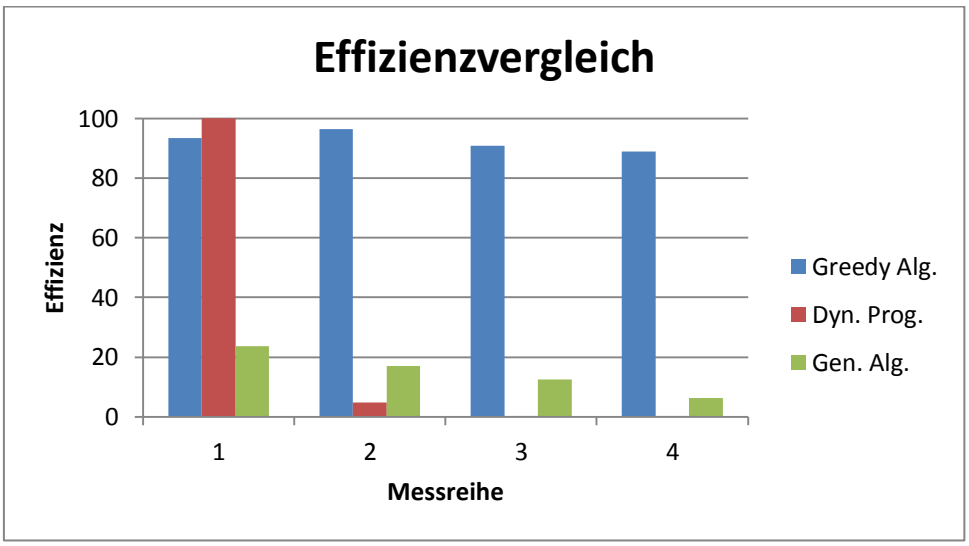

Abbildung 6: Effizienzvergleich der eingesetzten Algorithmen

Da bei der Implementierung des Genetischen Algorithmus besonderer Wert auf Parallelisierung gelegt wurde, findet sich in Abbildung 7 auch ein Vergleich der Laufzeiten von Messreihe 4 für eine Populationsgröße p = 100 bei unterschiedlicher Anzahl an eingesetzten Threads. Es zeigt sich deutlich, dass durch die parallele Abarbeitung auf mehreren Kernen die Laufzeit des Algorithmus gesenkt werden kann, jedoch nur solange die Anzahl verwendeter Threads nicht die Anzahl an für das Programm zur Verfügung stehenden Threads übersteigt. Das Testsystem verfügt über 8 virtuelle Kerne, wovon aber nicht alle Kerne für das Programm freigestellt werden können, da auch Aufgaben des Betriebssystems und anderer Prozesse abgearbeitet werden müssen. Sobald mehr Threads vom Programm angefordert werden, als virtuelle Kerne zur Verfügung stehen, muss der Scheduler des Betriebssystems die Aufgaben der vom Algorithmus verwendeten Threads auf die vorhandenen Kerne verteilen. Diese Verteilung ist mit einem Mehraufwand verbunden, der sich im Diagramm ab einer Anzahl von 8 Threads deutlich bemerkbar macht.

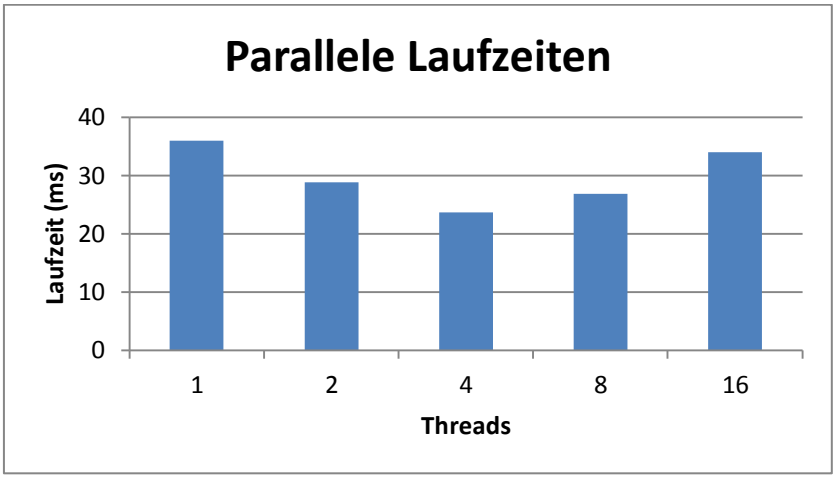

Abbildung 7: Laufzeitenvergleich unterschiedlicher Threadanzahl bei Genetischem Algorithmus

Um das evolutionäre Grundprinzip eines Genetischen Algorithmus, das „Überleben des Stärksten“, auch bildlich zu veranschaulichen, zeigt Abbildung 8 den typischen Verlauf eines Genetischen Algorithmus anhand von Messreihe 3. Über alle Generationen hinweg werden die durchschnittliche Fitness der Population, das fitteste Chromosom einer Generation sowie das fitteste Chromosom aller Generationen, dargestellt. Dabei fällt auf, dass die durchschnittliche Fitness der Population über lange Zeit gesehen ansteigt, da die fittesten Chromosomen bei der Reproduktion bevorzugt werden. Dieser Anstieg ist jedoch nicht kontinuierlich, denn wie in der Natur kommt es zu zwischenzeitlichen Rückschlägen und Fehlbildungen, vergleichbar etwa mit Krankheitsepidemien oder Gesundheitseinbrüchen in Folge von Hungersnöten. Gleiches gilt auch für das jeweils fitteste Chromosom einer Generation. Schließlich wird deutlich, dass das fitteste Chromosom aller Generation oft bereits sehr früh entdeckt wird. Erfahrene AnwenderInnen Genetischer Algorithmen können die Anzahl der Iterationen so anpassen, dass „unnötige“ Generationen eingespart werden und trotzdem eine sehr hohe beste Fitness erzielt wird.

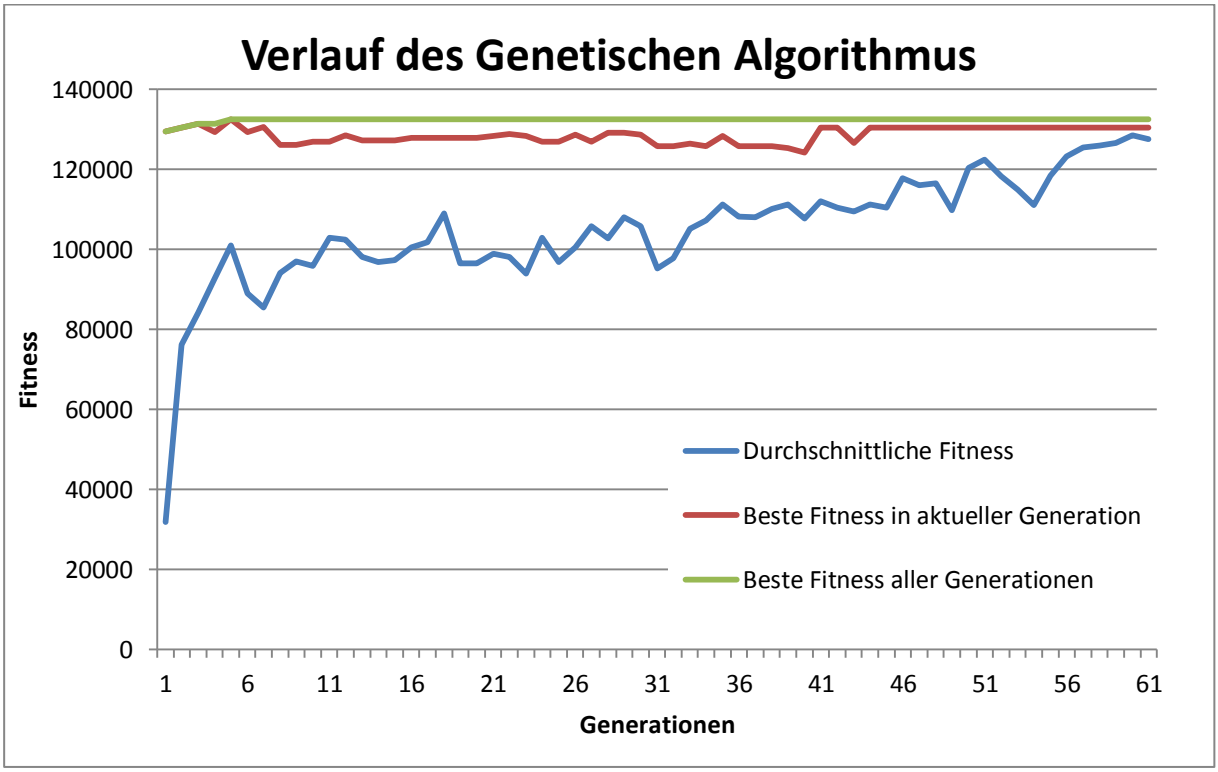

Abbildung 8: Verlauf des Genetischen Algorithmus

Schließlich wurde auch die Auswirkung der Variation der Mutationswahrscheinlichkeit p_m des Genetischen Algorithmus untersucht und für die bildliche Repräsentation der Resultate in Abbildung 9 der Verlauf der durchschnittlichen Fitness in Messreihe 4 für den Standardwert p_m = 0.001 und den erhöhten Wert p_m = 0.02 gemessen. Es wird ersichtlich, dass, während Mutation in geringem Ausmaß noch eine Reparaturfunktion darstellt, bei einer zu hohen Zahl an Mutationen die durchschnittliche Fitness der Population dauerhaft beeinträchtigt wird, da die Selektion der fittesten Chromosomen die hohe „Mutationsschädigung“ nicht ausgleichen kann. Ebenso fällt die höhere Schwankungsbreite der durchschnittlichen Fitness bei einer größeren Mutationsrate ins Auge.

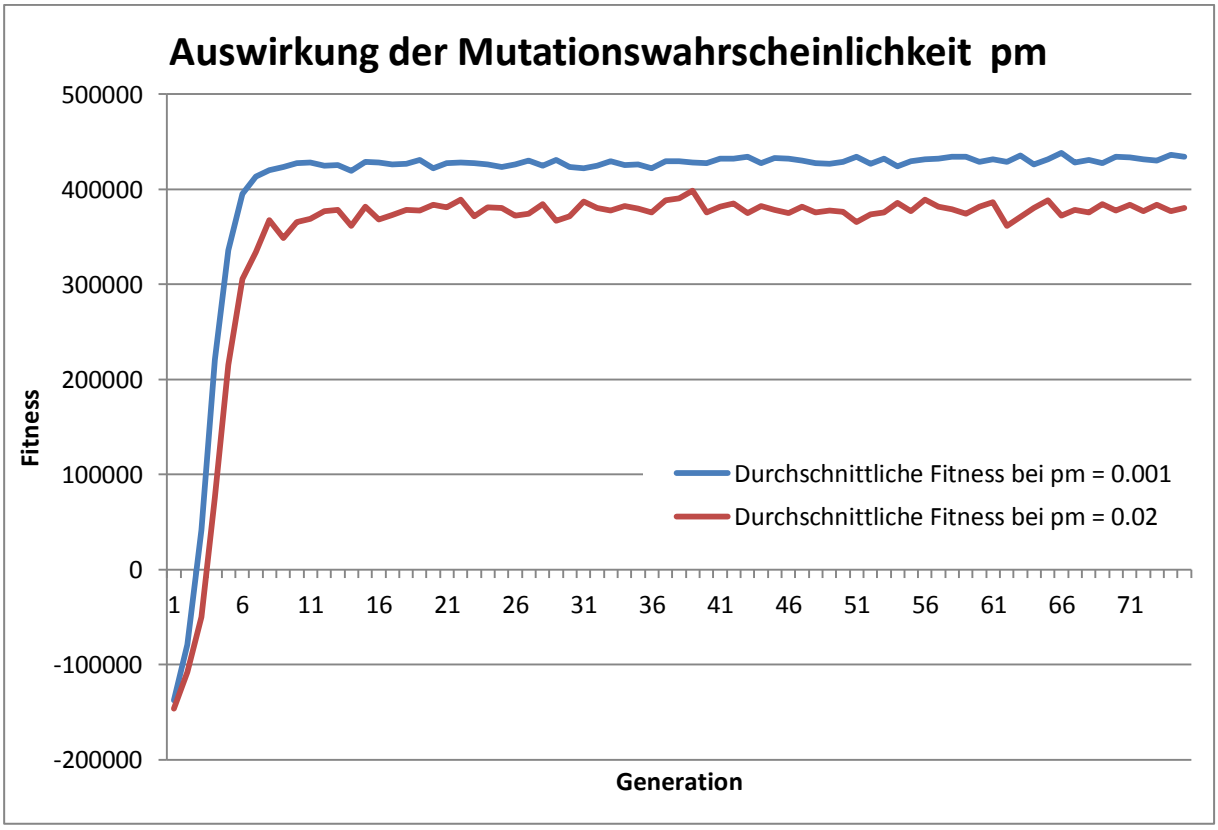

Abbildung 9: Auswirkung der Mutationswahrscheinlichkeit p_m

Anhand von Messreihe 4 in Tabelle 5 konnte durch Erhöhung der Populationsgröße p die Effektivität des Genetischen Algorithmus gesteigert werden. Dies erscheint logisch, da die Wahrscheinlichkeit, in einer riesigen Menge an Chromosomen ein besonders fittes Chromosom zu finden wesentlich höher sein muss als in einer kleinen Population. Auch wenn der erreichte Mehrwert deutlich niedriger ausfällt als die zusätzlich benötigte Laufzeit, ist allein schon der Umstand, dass sich für ein Optimierungsproblem die Laufzeit über einen konstanten Faktor, in diesem Fall die Größe der Population, steuern lässt, eine bedeutende Erleichterung.

3.2 Schlussfolgerung und Erfüllung der Aufgabenstellung

Im Rahmen dieser Arbeit konnte gezeigt werden, dass Genetische Algorithmen mit einer Komplexität von $O(n)$ tatsächlich in der Lage sind, komplexe Optimierungsprobleme in polynomieller Zeit, unter Einsatz eines vernünftigen Aufwandes, annähernd optimal, und damit ausreichend genau, zu lösen. Darüber hinaus können Genetische Algorithmen über die Parameter Populationsgröße und Iterationsanzahl sehr genau an die eigenen Bedürfnisse angepasst werden: je nachdem, wie viele Ressourcen zur Verfügung stehen, kann die Genauigkeit der Optimierung geregelt werden.
Auch die bereits in der Theorie (siehe Kapitel 1.3.3) beschriebenen Vorteile Genetischer Algorithmen gegenüber traditioneller Verfahren konnten in der Praxis belegt werden. So ermöglicht etwa die binärkodierte Abbildung der Parameter auch Gewichte, die als double oder float Werte dargestellt sind. Im Gegensatz dazu ist Dynamische Programmierung auf unsigned integer für das Gewicht eines Gegenstandes und die Rucksackkapazität beschränkt, da diese direkt als Indizes in einem Array verwendet werden. Ebenso wurde durch den Einsatz einer gut angepassten, vielfältigen Population anstelle eines einzigen Ausgangspunktes die Robustheit des Algorithmus beträchtlich gesteigert und über Zeit eine immer fittere Population herangezüchtet, sowie die hohe Parallelisierbarkeit gleicher Operationen auf einer Vielzahl unterschiedlicher Individuen ausgenutzt. Anders als bei einer Vielzahl anderer Optimierungsverfahren waren keine Zusatzinformationen oder Einschränkungen (etwa unsigned int Gewichte bei Dynamischer Programmierung) vonnöten. Schließlich wurde gezeigt, dass sich Genetische Algorithmen von rein zufallsbasierten Verfahren absetzen können, da die Selektion der fittesten Chromosomen die Suche in Suchregionen mit einer wahrscheinlichen Verbesserung führte und dadurch über längere Zeit höhere Fitnesswerte erreicht wurden.
Zusammen verleihen diese Kriterien den Genetischen Algorithmen eine beachtliche Flexibilität und ermöglichen deren Einsatz in einem breiten Spektrum von Problemstellungen.

Die eigenständige praktische Umsetzung des Genetischen Algorithmus als C++ Implementierung hilft dabei, die theoretischen Konzepte verständlicher zu machen und anhand eines konkreten Beispiels anzuwenden. Auch die Implementierung der Dynamischen Programmierung sowie des Greedy Algorithmus, welche in bestimmten Szenarien ebenso nützliche Lösungsverfahren darstellen, gewähren wichtige Einblicke in deren Funktionsweise und stellen für die Leserschaft eine Erleichterung bei eigener Anwendung besagter Verfahren dar. Tatsächlich können alle drei Algorithmen mit relativ wenig Aufwand an andere Problemstellungen angepasst werden. Die Struktur der Algorithmen selbst bleibt in allen Fällen erhalten, lediglich die Datenstrukturen, und im Fall des Genetischen Algorithmus auch die Evaluierungsfunktion zur Berechnung der Fitness, müssen adaptiert werden.

Im Gegensatz zu anderen gebräuchlichen Implementierungen Genetischer Algorithmen (zum Beispiel [6], S. 8, Fitness Function) wurden bei der Berechnung der Fitness von Chromosomen auch negative Fitnesswerte, welche ungültige Lösungen repräsentieren, zugelassen. Dies widerspricht dem Prinzip, dass ein Chromosom einer gültigen Lösung entsprechen muss, ermöglicht jedoch eine Aussage darüber, wie weit ein ungültiges (zu schweres) Chromosom von einer gültigen Lösung innerhalb der zulässigen Kapazität entfernt ist. Dadurch ist der Algorithmus im Stande, gewichtete Selektion auch bei Populationen durchzuführen, die aufgrund einer sehr knapp bemessenen Kapazität anfänglich nur ungültige Lösungen hervorbringen. Es kann somit schneller eine gültige Lösung gefunden werden und selbst wenn der Algorithmus beendet, ohne eine gültige Lösung berechnet zu haben, kann eine Aussage darüber getroffen werden, wie viel mehr Aufwand (in Form größerer Populationen oder einer höheren Zahl an Iterationen) noch ungefähr investiert werden muss, um zu einer gültigen Lösung zu kommen.

Im Vergleich mit Greedy Algorithmus und Dynamischer Programmierung stellen Genetische Algorithmen einen guten Kompromiss dar. Sie sind in der Lage, bessere (näher an der optimalen Lösung) Ergebnisse zu erzielen als der Greedy Algorithmus und benötigen für komplexe Probleme nur einen Bruchteil jener Laufzeit von Dynamischer Programmierung. Zudem sind Genetische Algorithmen von allen drei Kandidaten die flexibelsten, am vielfältigsten Einsetzbaren, und können darüber hinaus auch noch sehr gut parametrisiert werden. Die Gründe dafür wurden bereits im ersten Absatz dieses Kapitels erläutert.
Jedoch können auch die beiden anderen Verfahren in bestimmten Szenarien sehr nützlich eingesetzt werden: Der Greedy Algorithmus hat aufgrund seiner Komplexität von $O(n \log n)$ eine äußerst geringe Laufzeit. Seine Effektivität hängt zur Gänze von den eingegebenen Daten ab, er kann also durchaus auch sehr gute Ergebnisse erzielen. Die Anwendung des Greedy Algorithmus empfiehlt sich somit immer dann, wenn mittelgroße bis große Probleme vorliegen, und die Laufzeit anstelle der Genauigkeit der Lösung oberste Priorität besitzt.
Beim Greedy Algorithmus wurde bewusst nicht die Anwendung auf kleine Probleme empfohlen, da in diesen Fällen die Dynamische Programmierung beinahe immer vorzuziehen ist. Diese rechnet nämlich für niedrige Kapazitäten nur unmerklich langsamer als der Greedy Algorithmus, liefert jedoch immer die optimale Lösung und löst zudem auch alle Subprobleme mit geringerer Kapazität und einer kleineren Anzahl an Eingabeelementen. Dynamische Programmierung stellt, falls die optimale Lösung eine zwingende Anforderung ist, von den drei betrachteten Algorithmen die einzige Alternative dar und ist wie bereits erwähnt bei kleinen Problemen den anderen Verfahren vorzuziehen. Bei größeren Problemen muss abgewogen werden, ob der wesentlich höhere Aufwand die resultierende perfekte Genauigkeit wert ist. Aber auch für variable Problemstellungen, bei denen nach Belieben Eingabeelement hinzugefügt und entfernt werden können sollen, bietet sich die Dynamische Programmierung hervorragend an, da die Weiterverwendung bereits berechneter Lösungen von Teilproblemen möglich ist. Dies führt jedoch auch zu einem enormen Speicherbedarf, der bei großen Problemen die Kapazitäten des Systems sprengt.

Weiters fällt auf, dass Genetische Algorithmen der Theorie zufolge aufgrund einer Komplexität von $O(n)$ eigentlich schneller rechnen müssten als der Greedy Algorithmus mit einer Komplexität von $O(n \log n)$. Im praktischen Experiment konnte dieser Effekt jedoch nicht beobachtet werden. Die Erklärung liegt darin begründet, dass für die Betrachtung der Komplexität in Big O Notation der konstante Faktor der Populationsgröße s aus der Komplexitätsformel gestrichen wurde, dieser sich in der Realität aber natürlich dennoch auswirkt. Bei einer Populationsgröße von 100 Individuen müsste die Anzahl der Eingabeelemente n so groß sein, dass der binäre Logarithmus von n den Wert 100 übersteigt. Dafür müsste n eine Größe von 2^{100} erreichen, was wohl kaum je in einer praktischen Anwendung der Fall sein wird. Allerdings soll die Komplexitätsbetrachtung gemäß Big O Notation auch für nahezu unendlich viele Eingabeelemente Gültigkeit besitzen. Zudem kann durch Parallelisierung des Genetischen Algorithmus der Schwellwert an eingegebenen Elementen, der für eine schnellere Berechnung mittels Genetischem Algorithmus gegenüber dem Greedy Algorithmus erreicht werden muss, erheblich gesenkt werden. Im konkret vorliegenden Fall von 4 Threads können 25 Individuen gleichzeitig abgearbeitet werden, wodurch der Schwellwert theoretisch auf 2^{25} Elemente sinkt.

Abschließend könnte Genetischen Algorithmen beinahe schon das Prädikat „eierlegende Wollmilchsau der Informatik“ verliehen werden: Sie bieten Approximationen sehr nahe an der optimalen Lösung, glänzen durch polynomielle Laufzeit, sind in einem breiten Gebiet an Problemstellungen einsetzbar, äußerst flexibel und fein granular parametrisierbar. Einzig die durch Einschränkungen bedingte Sicherstellung gültiger Lösungskandidaten in den Operatoren, sowie das Finden einer geeigneten Evaluierungsfunktion, können sehr komplex ausfallen. ([2], S. 25 ff.)
Es bewahrheitet sich jedenfalls einmal mehr ein Prinzip, das dem Menschen schon zu zahlreichenden technologischen Höchstleistungen verholfen hat: die Natur als richtungsweisendes Vorbild der Technik.

Anhang A: Beispiel eines Genetischen Algorithmus

Das folgende Beispiel entstammt dem Referenzwerk „Genetic Algorithms in Search, Optimization and Machine Learning“ [4], S. 8 ff., und soll die Umsetzung der in Kapitel 1.1 beschriebenen Konzepte Genetischer Algorithmen veranschaulichen.

Als Problem wird die Optimierung des Parameters/Lösungskandidaten x der Funktion $f(x) = x^2$ betrachtet, welche in Abbildung 10 dargestellt ist. Dem geübten mathematischen Auge erscheint die Lösung des Problems geradezu intuitiv: je größer x gewählt wird, desto höher steigt der Funktionswert $f(x)$ und damit die Fitness des Lösungskandidaten. Jedoch ist ein Genetischer Algorithmus gewissermaßen blind, und kann die Qualität eines Lösungskandidaten x nur durch Evaluierung anhand der Fitnessfunktion (in diesem Fall $f(x) = x^2$) eruieren und Muster, die zur Lösung führen (wie etwa die Maximierung des Wertes von x), nicht wie ein Mensch mit freiem Auge und ein wenig Erfahrung erkennen. Die besagte Erfahrung schlägt sich in Genetischen Algorithmen allerdings durch die iterative Selektion der fittesten Individuen nieder. Schließlich sollte auch nicht darauf vergessen werden, dass das behandelte Beispiel extrem simpel ist und nur der Veranschaulichung dient. Der Vorteil der Genetischen Algorithmen liegt vielmehr darin, dass sie trotz ihrer Blindheit auch äußerst komplexe Probleme, bei denen die Wahl der Evaluierungsfunktion bei weitem nicht so offensichtlich ausfällt, bewältigen können, da sie sich hierbei auf dieselben Verfahren stützen wie zur Lösung von simplen Aufgabenstellungen.

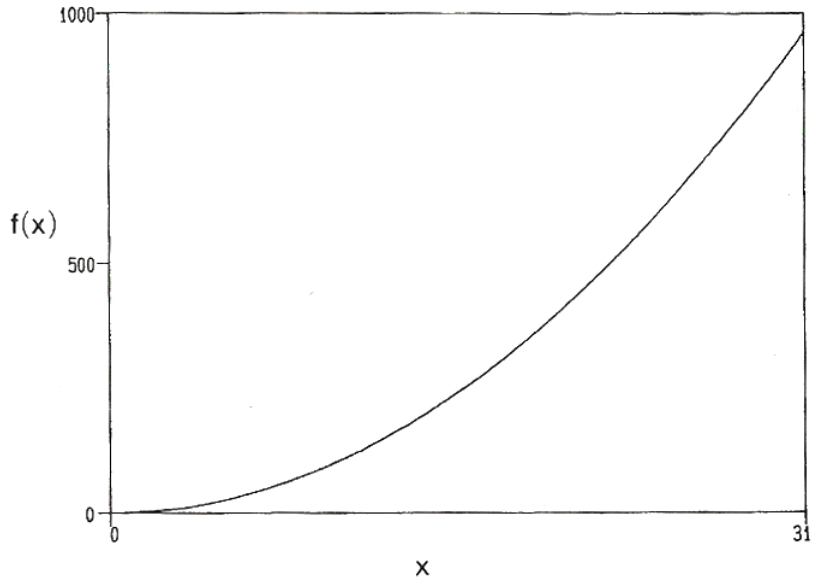

Abbildung 10: Die Funktion $f(x) = x^2$(Quelle [4], S. 8, Figure 1.5)

Der Wertebereich von x ist hier auf [0, 31] beschränkt. Die optimale Lösung wäre also intuitiv ein Wert für x von 31 und beträgt $31^2 = 961$.

Wie bereits erwähnt arbeiten Genetische Algorithmen auf binären Kodierungen von Parametern/Lösungskandidaten. Zur Abbildung des Wertebereichs werden 5 Bit benötigt ($2^5 = 32$ mögliche Werte). Die optimale Lösung in Binärkodierung dargestellt ist somit 11111.

Die Blindheit Genetischer Algorithmen sowie die Binärkodierung der möglichen Lösungskandidaten verbildlicht Goldberg anhand einer in Blackbox mit 5 on/off Schaltern, die zu jedem eingespeisten Signal s (einem binär kodierten Lösungskandidaten) mit Hilfe der Evaluierungsfunktion $f(s)$ das Ausgabesignal (den Fitnesswert, auch „Payoff“) zurückgibt. Die Positionen der einzelnen Schalter stellen den binären Wert an der jeweiligen Stelle im binären String eines Lösungskandidaten dar (1 für on, 0 für off).

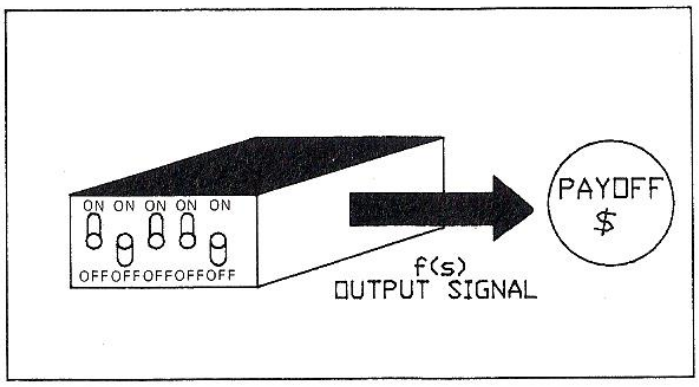

Abbildung 11: Blackbox, die zu einem binär kodiertem Eingangssignal s einen Fitnesswert als Ausgabesignal $f(s)$ zurückgibt (Quelle [4], S. 8, Figure 1.6)

In dem Beispiel wird von einer Anfangspopulation mit der Größe $n = 4$ und den zufällig generierten Individuen 01101, 11000, 01000 und 10011 ausgegangen.

Die Evaluierung der Fitnesswerte jedes Individuums durch die Fitnessfunktion $f(x) = x^2$ sowie deren prozentuelle Gewichtung ist in Tabelle 6 dargestellt:

No.	String	Fitness	% of Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.0

Tabelle 6: Evaluierung durch die Fitnessfunktion $f(x) = x^2$ (Quelle [4], S. 11, Table 1.1)

Als nächstes werden an der Reproduktion beteiligte Individuen durch gewichtete Selektion ausgewählt. Die in Kapitel 1.1.3 angesprochene Versinnbildlichung anhand eines Roulette Rades ist in Abbildung 12 zu sehen. Jedes Individuum erhält auf dem Roulette Rad einen Slot in jener Größe zugewiesen, die der durch den Fitnesswert ermittelten Proportion entspricht. Das Roulette Rad wird vier Mal gedreht ($n = 4$) und der jeweilige „Gewinner“ in die nächste Population übernommen. Die Chance eines fitten Individuums, in der Folgegeneration vertreten zu sein, ist somit wesentlich höher als die Chance eines nicht so fitten Individuums.

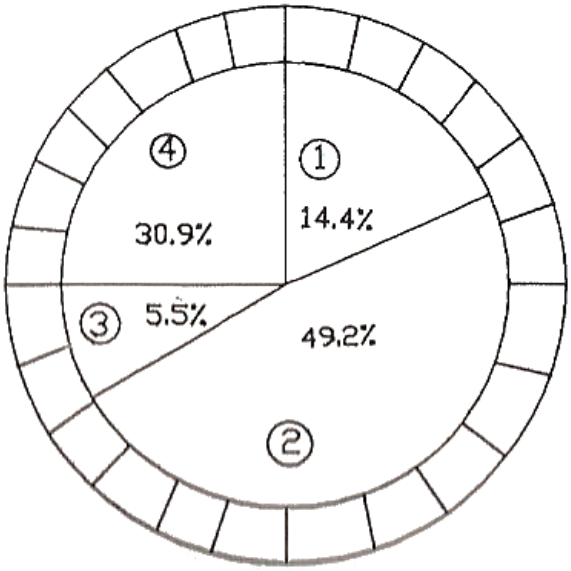

Abbildung 12: Roulette Rad zur gewichteten Selektion (Quelle [4], S. 11, Figure 1.7)

Nach Abschluss der Reproduktion werden die neuen Individuen mit einer Wahrscheinlichkeit p_c zum Crossover zugelassen. Für dieses Beispiel wird angenommen, dass die Strings 11000 und 10011 Eingang in die neue Population gefunden haben und für das Crossover selektiert wurden. Der zufällig ermittelte Locus k ist die Position 2 (die 1. Stelle hat hier einen Index von 1). Die beiden Strings werden nun also nach der 2. Stelle abgeschnitten und deren hintere Teilstrings vertauscht:

11|000	→	11|011
10|011		10|000

Der Crossover Vorgang wird in Abbildung 13 graphisch abgebildet. Es werden hierbei deutlich die unterschiedlichen Ausprägungen des ausgetauschten Genmaterials dargestellt.

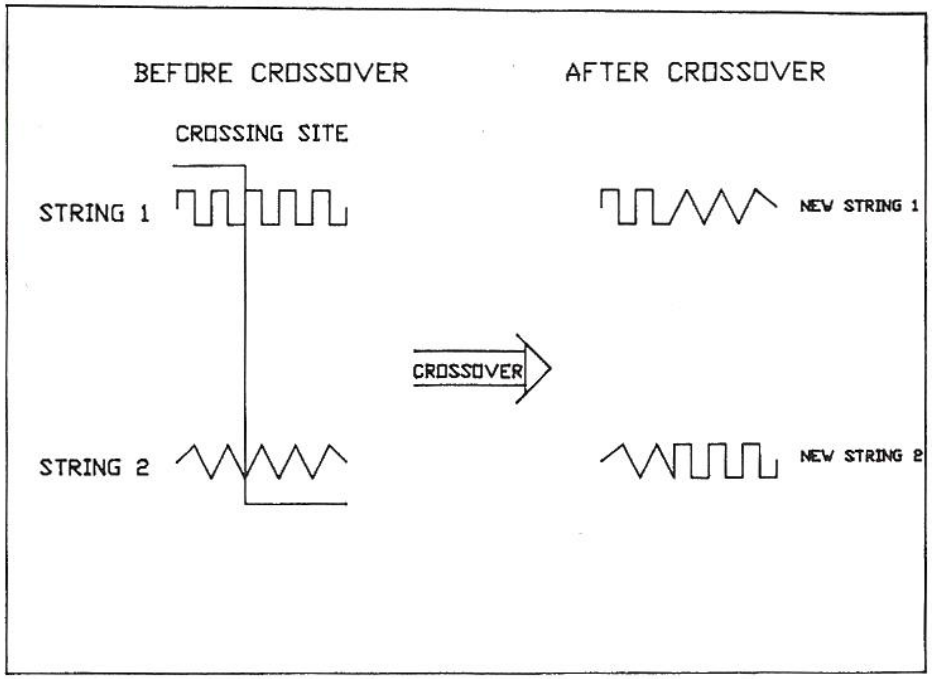

Abbildung 13: Crossover zweier Strings (Quelle [4], S. 12, Figure 1.8)

Schließlich werden durch Anwendung des Mutationsoperators zufällige Bits in allen Individuen manipuliert. Dazu muss für jedes Bit in jedem Individuum eine Zufallszahl berechnet werden. Ist diese kleiner als die Mutationswahrscheinlichkeit p_c, wird der Wert des Bits umgekehrt.

Die obigen Schritte werden nun eine vorgegebene Anzahl von Iterationen wiederholt.
Ohne jede Iteration im Detail zu beschreiben ist absehbar, dass die Population am Ende aus einer möglichst hohen Anzahl an 11111 Strings bestehen wird, da diese die höchste Fitness aufweisen. Somit wird in diesem äußerst simplen Beispiel auch sehr wahrscheinlich die optimale Lösung gefunden. In komplexeren Aufgabenstellungen dagegen ist es eher unwahrscheinlich, dass die optimale Lösung tatsächlich gefunden wird, jedoch ist in vernünftiger Zeit (50 bis 500 Iterationen) eine gute Annäherung an diese möglich.

Anhang B: Grundzüge der Komplexitätstheorie

Ziel dieses Anhangs

Wie bereits in Kapitel 1.2.1 erwähnt werden die wesentlichen Konzepte der Komplexitätstheorie an dieser Stelle nur grundlegend erklärt. Auf eine vollständige Beweisführung wird bewusst verzichtet. Am Ende soll der Leser/die Leserin im Stande sein, den Ausdruck „NP-schwer“ zu deuten und die Signifikanz der Genetischen Algorithmen im Zusammenhang mit der aktuellen Lösung von NP-schweren Problemen zu erkennen.

In den folgenden Teilabschnitten wird zunächst auf das Konzept der Komplexität eingegangen. Es folgt eine genauere Betrachtung der Bezeichnung „NP“: diese steht für nichtdeterministisch polynomielle Zeit ([14], S.60). Außerdem werden die Begriffe „Entscheidungsproblem“, „Optimierungsproblem“ und „Reduzierbarkeit“ beschrieben, die zur Definition einzelner Komplexitätsklassen benötigt werden. Danach erfolgt eine Erklärung der Klassen P, NP, NP-schwer sowie NP-vollständig. Abschließend wird das Rucksackproblem im Kontext der Komplexitätstheorie eingeordnet und anhand einer graphischen Darstellung nochmals ein Überblick über die besprochenen Komplexitätsklassen gegeben.

Komplexität

Die Komplexitätstheorie, ein Teilbereich der Theoretischen Informatik, versucht den Aufwand der Lösung eines Problems abzuschätzen und bedient sich zum Vergleich zweier Algorithmen einer als Komplexitätsmaß bezeichneten Messgröße [15], S. 372.

Für die Betrachtungen im Laufe dieser Arbeit kommen als Messgrößen die Laufzeit eines Algorithmus sowie der benötigte Speicherplatz in Frage. Dabei werden die maximale Laufzeit und der maximal benötigte Speicherplatz mittels O-Notation angegeben.
Im weiteren Verlauf wird als Komplexitätsmaß standardmäßig die Laufzeit eines Algorithmus festgelegt und explizit darauf hingewiesen, falls dies nicht der Fall ist.

Determinismus versus Nicht-Determinismus

Gemäß [16], S. 580, lassen nicht-deterministische Programme bei der Ausführung mehrere mögliche Abläufe zu, während deterministische Programme bei gleichen Eingabeparametern genau einen möglichen Ablauf haben. Als Beispiel für eine nicht-deterministische Konstruktion wird folgende bewachte Anweisung aufgeführt:

if $x \geq 0 \rightarrow$ print („nichtnegativ“)
□ $x \leq 0 \rightarrow$ print („nichtpositiv“)
fi.

Diese Konstruktion ist nicht-deterministisch, da im Falle von x = 0 nicht klar definiert ist, welche der beiden Anweisungen ausgeführt werden soll. Würde hier das ≤ der zweiten Prüfbedingung durch ein < ersetzt, wären alle möglichen Fälle für x klar definiert und die Konstruktion eine deterministische.

Polynomialzeit-, Pseudopolynomielle und Nicht-Polynomielle Algorithmen

Laut Kellerer, Pferschy und Pisinger [7], S. 13, sind die effizientesten Algorithmen jene, deren Laufzeit durch ein Polynom von n (Anzahl der betrachteten Elemente) begrenzt wird, wie zum Beispiel $O(n), O(n \log n), O(n^3)\ oder\ O(n^k)$. Diese Algorithmen werden auch als Polynomialzeit Algorithmen bezeichnet und wenn eine Aufgabe in annehmbarer/vertretbarer/akzeptabler Zeit gelöst werden soll, sind damit polynomielle Algorithmen gemeint.
Darüber hinaus gibt es die sogenannten Pseudopolynomiellen Algorithmen, bei denen die Laufzeit asymptotisch durch ein Polynom sowohl in n als auch in einem oder mehreren Eingabewerten begrenzt wird. Dazu zählen etwa $O(nc)$ und $O(n^2 p_{max})$. Diese werden als weniger „attraktiv" angesehen, da trotz einer geringen Zahl an betrachteten Elementen n die Laufzeit durch große Eingabewerte sehr in die Länge gezogen werden kann.
Als unangenehmste Klasse von Algorithmen gelten schließlich die Nicht-Polynomiellen Algorithmen. Dazu gehören in etwa Algorithmen, deren Laufzeit durch eine Exponentialfunktion wie $O(2^n)$ begrenzt wird.
Vergleicht man die Laufzeit eines Polynomialzeit Algorithmus $O(n^3)$ mit jener eines Nicht-Polynomiellen Algorithmus $O(2^n)$, wird klar, warum ersterer bevorzugt wird:
Bei einer Verdopplung von n erhöht sich diese bei $O(n^3)$ um den Faktor 8, während sie bei $O(2^n)$ quadriert wird.

Entscheidungsprobleme und Optimierungsprobleme

Als Entscheidungsprobleme werden jene Probleme bezeichnet, die „ja" oder „nein" als Antwort liefern. Es wird zwischen Algorithmen unterschieden, die ein Entscheidungsproblem akzeptieren, und jenen, die ein Entscheidungsproblem entscheiden.
Algorithmen akzeptieren ein Entscheidungsproblem, wenn sie für jede „yes"-Instanz eines Entscheidungsproblems in polynomieller Zeit „yes" ausgeben, ihr Verhalten bei „no"-Instanzen jedoch nicht klar definiert ist. Das heißt, dass in zweitem Falle diese Algorithmen unterschiedliches Verhalten aufweisen können, indem sie etwa in polynomieller Zeit „no" ausgeben, nicht in polynomieller Zeit terminieren oder gar nicht terminieren.
Falls ein Algorithmus für jede „yes"-Instanz eines Entscheidungsproblems „yes" und für jede „no"-Instanz „no" in polynomieller Zeit ausgibt, wird das Problem vom Algorithmus entschieden S. 484 f.

Neben den Entscheidungsproblemen existieren sogenannte Optimierungsprobleme, bei denen jede Lösung einen zugordneten Wert hat und man aus allen plausiblen Lösungen diejenige mit dem besten Wert sucht [8], S. 1050.

Cormen et al. [8], S. 1051, zufolge können Optimierungsprobleme gewöhnlicher Weise in Entscheidungsprobleme umgewandelt werden, indem der zu optimierende Wert begrenzt wird. So kann in etwa die Frage nach dem kürzesten Weg zwischen zwei Punkten in die Frage, ob zwischen diesen zwei Punkten überhaupt ein gültiger Weg existiert, oder ob dieser kürzer/länger als ein bestimmter Schwellwert ist, umgewandelt werden.

Instanz eines (Rucksack) Problems

Eine Instanz I ist ein konkreter Datensatz, der einem Problem unterworfen wird.
Für das Rucksackproblem in Form eines Entscheidungsproblems wird eine Instanz definiert als $I = \{p_1, \dots, p_n, w_1, \dots, w_n, c\}$.
Ein Beispiel hierfür ist der Datensatz in Abbildung 14:

j	1	2	3	4
p_j	8	5	2	3
w_j	5	2	4	8

$c = 10, t = 9$

Abbildung 14: Instanz des Rucksackproblems (Quelle [7], S. 484 A.3)

Da bei Auswahl der Gegenstände j_1 und j_3 eine Komfortsumme $p_1 + p_3$ von 10 zustande kommt, welche den Schwellenwert $t = 9$ übersteigt und zugleich die Summe der Gewichte $w_1 + w_3 = 9$ die maximale Kapazität von $c = 10$ nicht überschreitet, handelt es sich um eine „yes“ - Instanz. ([7], S.484).

Größe der Eingabe

Um das Rucksackproblem im Kontext der Komplexitätstheorie einordnen zu können, müssen auch Aussagen über die Speicherplatz Komplexität des Rucksackproblems getroffen werden. Diese wird als Funktion der Eingabegröße einer beliebigen Instanz $L(I)$ angegeben.
Die Eingabegröße einer Instanz ist definiert als Anzahl von Bits, die benötigt werden, um eine Instanz in binärer Form abzubilden. ([7], S.484)

Diese Anzahl an Bits berechnet sich wiederum aus der Summe der Größe aller für die Instanz benötigten, als Zahlen dargestellten, Eingabeparameter. Es stellt sich somit für jeden Eingabeparameter die Frage, wie viele Bits notwendig sind, um diesen zu repräsentieren.
Im Binärsystem können durch n Bits 2^n unterschiedliche Werte gespeichert werden. 4 Bits erlauben somit die Speicherung von $2^4 = 16$ verschiedenen Werten.

Im Umkehrschluss kann die zur Speicherung eines Wertes benötigte Anzahl an Bits durch die Umkehrfunktion der binären Exponentialfunktion 2^n, den binären Logarithmus $\log_2 n$, berechnet werden.

Bei genauer Betrachtung ist die oben angegebene Formel für den binären Logarithmus nicht korrekt, da sie für den Wert 1 eine Bitanzahl von 0 ergibt, obwohl mindestens ein Bit benötigt wird, um überhaupt Informationen zu speichern. Zudem ist der binäre Logarithmus des Wertes 0, also $\log_2 0$, als unendlich definiert. Eine Korrektur der Berechnungsformel für die zur Speicherung eines Wertes benötigte Anzahl an Bits auf $\lfloor \log_2 n \rfloor + 1$ behebt die beiden genannten Probleme und liefert durch Abrundung ($\lfloor\rfloor$) des Ergebnisses des binären Logarithmus eine Ganzzahl in Bit zurück [17].
Für die Betrachtung der Komplexität spielen die additive Konstante +1 sowie die Abrundung ($\lfloor\rfloor$) jedoch keine Rolle, da die Größenordnung $\log_2 n$ unverändert bleibt. Daher wird zur Berechnung der Speicherplatz Komplexität schlicht von $\log_2 n$ ausgegangen.

Gemäß [7], S. 484, ist es üblich, für das Rucksackproblem in Entscheidungsform anzunehmen, dass alle Gewichte kleiner sind als c und alle Komfortwerte kleiner als t.
Für Abbildung 14 kann die gesamte Eingabegröße als maximal $((1c + 4w) + (1t + 4p)) * 4\,Bit = 40$ Bit angegeben werden, da sowohl $c = 10$ als auch $t = 9$ eine Bitanzahl von 4 zur Speicherung benötigen.
Als Funktion ausgedrückt ergibt dies $L(I) = (n+1)\log_2 c + (n+1)\log_2 t$ und die Eingabegröße für I wird somit durch $O(n \log c + n \log t)$ in der Speicherplatz Komplexität beschränkt.

Reduzierbarkeit - intuitiv

Die Reduzierbarkeit von Problemen ist ein wesentliches Konzept der Theoretischen Informatik und wird zum leichteren Verständnis in zwei Stufen erläutert. Dieser Abschnitt widmet sich der intuitiven Beschreibung der Reduzierbarkeit von Problemen. Eine formal korrekte Definition folgt im nächsten Abschnitt.

Intuitiv ausgedrückt wird als Reduzierbarkeit die Möglichkeit bezeichnet, ein Problem R in ein anderes Problem Q zurückzuführen beziehungsweise umzuformulieren. Eine Reduktion in polynomieller Zeit wird durch Formel (4) angegeben:

$$R \leq_p Q$$ (4) Reduktion von R auf Q in polynomieller Zeit ([14], S. 14)

Ein Polynomialzeit - Algorithmus der Q löst kann folglich als Unterprogramm zur Lösung von R, ebenfalls in polynomieller Zeit, eingesetzt werden. ([14], S. 14)

Ein Beispiel für eine Reduktion in polynomieller Zeit ist die Potenzfunktion $pow(x,n)$, die auf eine Reihe von Multiplikationen (siehe Abbildung 15) und diese wiederum auf eine Reihe von Additionen (siehe Abbildung 16) reduziert werden können. Hierbei ist allerdings zu beachten, dass es sich bei der Potenzfunktion um kein Entscheidungsproblem handelt. Dieses Beispiel dient lediglich zur Verdeutlichung des Prinzips einer Reduktion und ist formal nicht korrekt.

```
int pow(x, n)
{
  int retVal = 1;
  for (int i = 0; i < n; ++i)
  {
    retVal = multiply(retVal, x);
  }
  return retVal;
}
```

Abbildung 15: Reduktion von Potenzfunktion auf Reihe von Multiplikationen

Die Reduktion von $pow(x,n)$ nutzt zur Lösung die Funktion $multiply(x,n)$ als Unterprogramm. Die Reduktion erfolgt in polynomieller Zeit, da das Unterprogramm n mal durchlaufen werden muss.

```
int multiply(x, n)
{
  int retVal = 0;
  for (int i = 0; i < n; ++i)
  {
    retVal = add(retVal, x);
  }
  return retVal;
}
```

Abbildung 16: Reduktion von Multiplikation auf Additionen

Gleiches gilt bei der Reduktion von $multiply(x,n)$ auf $add(x_1, x_2)$. Die Addition zweier Zahlen (Abbildung 17) besteht aus nur einer Operation und besitzt somit eine Komplexität von $O(1)$. Sie kann daher nicht weiter reduziert werden.

```
int add(x1, x2)
{
  return x1 + x2;
}
```

Abbildung 17: Addition zweier Zahlen mit O(1)

Insgesamt konnte die Funktion $pow(x,n)$ durch zweimalige Reduktion in jeweils $O(n)$ auf die Funktion $add(x_1, x_2)$ mit der Komplexität $O(1)$ reduziert werden. Die gesamte Reduktion wurde in polynomieller Zeit $O(n*n) = O(n^2)$ durchgeführt. Die gesamte Komplexität von $pow(x,n)$ setzt sich zusammen aus der zur Reduktion benötigten Zeit $O(n^2)$ sowie der Zeit zur Berechnung des reduzierten Problems $O(1)$. Daraus ergibt sich für $pow(x,n)$ eine Komplexität von $O(n^2 * 1) = O(n^2)$.

Reduzierbarkeit - formal

Cormen et al. zufolge [8], S.1051, werden Reduktionen in beinahe jedem Beweis von NP-Vollständigkeit eingesetzt.
Als Ausgangspunkt dient ein Entscheidungsproblem A, das in polynomieller Zeit gelöst werden soll. Die Eingabe für dieses Problem wird als Instanz α von A bezeichnet.
Gegeben ist zudem ein zweites Entscheidungsproblem B, von dem eine Lösung in Polynomialzeit bekannt ist.
Schließlich existiert ein Verfahren, dass jeder Instanz α von A einer Instanz β von B zuordnet. Die Zuordnung erfolgt dabei in polynomieller Zeit und die Antworten von α müssen denen von β entsprechen. Die Antwort von α darf also nur dann „ja“ sein, wenn die Antwort von β ebenfalls „ja“ ist.

Abbildung 18 zeigt ein Verfahren zur Reduktion eines Entscheidungsproblems A auf ein Entscheidungsproblem B:

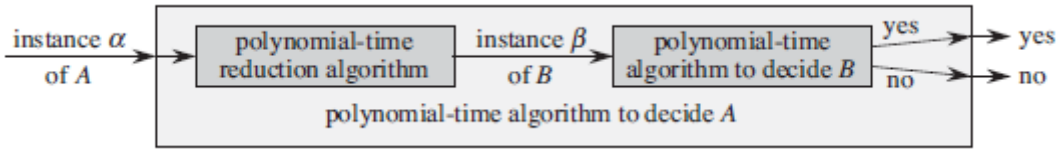

Abbildung 18: Reduktion von A auf B in polynomieller Zeit (Quelle [8], S.1052)

Dieses Verfahren wird als Polynomialzeit Reduktionsalgorithmus bezeichnet und zur Lösung des Entscheidungsproblems A in polynomieller Zeit eingesetzt:

1. Eine Instanz α von A wird in eine Instanz β von B mithilfe eines Polynomialzeit Reduktionsalgorithmus umgewandelt.
2. Der Polynomialzeit Entscheidungsalgorithmus von B wird für die Instanz β ausgeführt.
3. Das Ergebnis von β wird als Ergebnis von α verwendet.

Solange jeder dieser drei Schritte in polynomieller Zeit ausgeführt wird, ist auch die Lösung von A in Polynomialzeit möglich. Wenn hingegen bereits bekannt ist, dass A nicht in polynomieller Zeit lösbar ist und B auf A reduziert werden kann, so ist auch B nicht in polynomieller Zeit lösbar ([8], S. 1052).

Komplexitätsklasse P

Cormen et al. [8], S. 1049, beschreiben die Komplexitätsklasse P als Menge jener Probleme, die in polynomieller Zeit gelöst werden können. Genauer gesagt können diese Probleme in $O(n^k)$ gelöst werden, wobei n die Anzahl an Eingabeparametern und k eine Konstante ist.

P kann auch als Menge aller Entscheidungsprobleme bezeichnet werden, die von Algorithmen in polynomieller Zeit entschieden werden können [7], S.485.

Komplexitätsklasse NP

Die Komplexitätsklasse NP beinhaltet über P hinaus jene Probleme, deren Lösung nicht mit Polynomialzeit - Algorithmen berechnet werden kann, bei denen jedoch sowohl das „Erraten“ als auch das Verifizieren einer Lösung in polynomieller Zeit möglich ist. ([6], S. 6)

Cormen et al. [8], S. 1049, sprechen von einem auf nicht genau festgelegte Weise erhaltenen „Zertifikat“ (eine vorgeschlagene Lösung [7], S. 485), das einem Verifizierungsalgorithmus zur Überprüfung vorgelegt wird. Die Korrektheit dieses Zertifikats kann für NP Probleme in polynomieller Zeit verifiziert werden.

Es ist jedoch nicht für alle Probleme in NP möglich, die Lösung direkt in polynomieller Zeit zu berechnen, und das Zertifikat (welches zum Beispiel bloß erraten sein könnte) kann sich auch als inkorrekt herausstellen.

Etwas formeller definieren Kellerer, Pferschy und Pisinger [7], S. 486, die Klasse NP:
Ein Problem ist genau dann Teil der Menge NP ($Q \in NP$), wenn ein Polynomialzeit - Verifizierungsalgorithmus $A(I, C)$ und eine Konstante k existieren, sodass gilt:

Für alle „yes“ Instanzen $I \in Q$ existiert ein Zertifikat C mit $L(C) = O\left(L(I)^k\right)$,
wobei $A(L, C)$ „yes“ ausgibt.

(5) Formelle Definition der Klasse NP ([7], S. 486)

Dies bedeutet einerseits, dass es nie dazu kommen darf, dass ein ungültiger Lösungsvorschlag als gültige Lösung verifiziert wird, und andererseits, dass die Größe jedes Zertifikats polynomiell, also einzig aufgrund der Anzahl an Eingabeparametern der Instanz, durch die Größe der Instanz beschränkt sein muss.

Weiters ist jedes Problem aus P ebenso in NP enthalten, da ein Problem aus P sogar ohne Vorlage eines Zertifikats direkt in polynomieller Zeit gelöst werden kann [8], S. 1049.
Die Verifikation wird durch die direkte Berechnung einer gültigen Lösung bereits impliziert.

Die Frage, ob die Menge aller Probleme, deren Lösungen direkt in polynomieller Zeit berechnet werden können, mit der Menge aller Probleme, die in polynomieller Zeit verifiziert werden können, übereinstimmt, also ob P = NP?, ist eines der sieben „Millenium Problems“ [18] des Clay Math Institutes. Für die Lösung dieses Problems hat das Institut ein Preisgeld von 1 Million US-Dollar ausgeschrieben.

Zur besseren Illustration des P vs NP Problems und zur Verdeutlichung, wie sich diese Komplexitätsklassen unterscheiden, ist auf der Website des Clay Math Institutes ein Beispiel [19] beschrieben.

NP-schwer

NP-schwere Probleme besitzen die Eigenschaft, dass alle Probleme aus NP in polynomieller Zeit auf diese reduziert werden können. Allerdings muss ein NP-schweres Problem nicht notwendigerweise in polynomieller Zeit verifizierbar, also Teil von NP, sein. NP-schwere Entscheidungsprobleme, die in Polynomialzeit verifiziert werden können, bilden die Untermenge der NP-vollständigen Probleme. ([8], S. 1069)

Allen NP-schweren Problemen gemein ist die Tatsache, dass, ließe sich für ein NP-schweres Problem eine optimale Lösung in polynomieller Zeit finden, zugleich alle NP-schweren Probleme in polynomieller Zeit gelöst werden könnten. Es wird daher davon ausgegangen, dass kein Algorithmus existiert, der ein NP-schweres Problem in polynomieller Zeit optimal lösen kann. ([7], S. 14)

NP-vollständig

Kellerer, Pferschy und Pilsinger [7], S. 486, definieren NP-vollständig als Menge aller Entscheidungsprobleme Q, die die folgenden zwei Eigenschaften (6) besitzen:

1. $Q \in NP$
2. $\forall R \in NP : R \leq_p Q$

(6) Definition von NP-vollständig (Quelle [7], S. 486)

Anders ausgedrückt ist NP-vollständig die Menge aller Entscheidungsprobleme, die in polynomieller Zeit verifizierbar sind und auf die alle Probleme aus NP in polynomieller Zeit reduziert werden können ([14], S. 14). NP-vollständig ist also die Schnittmenge von NP und NP-schweren Entscheidungsproblemen.

NP-vollständige Probleme können weiters gemäß der Schwere ihrer Lösung in schwach NP-schwere und stark NP-schwere Probleme unterschieden werden.
Kann die Lösung eines NP-vollständigen Entscheidungsproblems in pseudopolynomieller Zeit berechnet werden, handelt es sich um ein schwach NP-schweres Problem.
Als stark NP-schwer werden jene NP-vollständigen Entscheidungsprobleme bezeichnet, die, selbst wenn alle in der Eingabe verwendeten Zahlen polynomiell beschränkt sind zur Größe der Eingabe, immer noch NP-schwer sind. ([7], S. 487)

Obwohl NP-vollständig auf Entscheidungsprobleme beschränkt ist, kann man sich dennoch die Beziehung zwischen Entscheidungs- und Optimierungsproblemen zu Nutze machen:
Es gilt, dass ein Optimierungsproblem schwerer zu lösen ist als das verbundene Entscheidungsproblem. Gelingt also der Beweis, dass ein Entscheidungsproblem schwer zu lösen ist, so ist auch ein damit verbundenes Optimierungsproblem schwer zu lösen. ([8], S. 1050 f.)

Die meisten theoretischen Informatiker glauben, dass NP-vollständige Probleme in polynomieller Zeit unlösbar sind. Dies konnte aber bislang nicht bewiesen werden.
Aus Bedingung 2 der Definition von NP-vollständig (6) folgt jedenfalls:
Findet sich ein Algorithmus, der nur ein NP-vollständiges Problem in polynomieller Zeit löst, so können alle Probleme in NP in Polynomialzeit gelöst werden und P wäre gleich NP.
([8], S. 1050)

Einordnung des Rucksackproblems

Es gibt starke theoretische Hinweise darauf, dass das Rucksack Problem nicht in polynomieller Zeit gelöst werden kann. Daher ist das Rucksack Problem üblicherweise nicht der Klasse P zugeordnet, obwohl es aus dieser nicht mit Sicherheit ausgeschlossen werden kann. ([7], S.14)

Als nächste mögliche Klasse kommt NP in Frage, welche die Verifizierung eines Problems in polynomieller Zeit voraussetzt. Für das Rucksackproblem als Entscheidungsproblem kann als Zertifikat die Liste von Indizes der ausgewählten Gegenstände betrachtet werden.
Am Beispiel der Instanz von Kapitel 0, Abbildung 14, wäre ein solches Zertifikat $C = \{1, 3\}$.
Der zur Verifizierung eingesetzte Algorithmus $A(I, C)$ wird in Formel (7) dargestellt:

$$\sum_{j \in C} p_j \geq t \text{ und } \sum_{j \in C} w_j \leq c \qquad (7) \text{ ([7], S. 485, A.4)}$$

Falls alle durch das Zertifikat C referenzierten Gegenstände in Summe ein Gewicht kleiner oder gleich der maximalen Kapazität c aufweisen und die Summe der Komfortwerte mindestens so groß ist wie der Schwellenwert t, gibt der Algorithmus „yes“ zurück.
Die Laufzeit ist mit $O(n + n) = O(2n) = O(n)$ polynomiell. ([7], S.485)

Das Rucksackproblem als Entscheidungsproblem kann also in polynomieller Zeit verifiziert werden und gehört somit der Klasse NP an.
Dies wird auch durch Betrachtung der formellen Definition von NP in Formel (5), bestätigt: Der Verifizierungsalgorithmus liefert für jede „yes“ Instanz des Rucksackproblems in Entscheidungsform ein „yes“ zurück und die Größe eines Zertifikats ist polynomiell durch die Größe der Instanz, also allein aufgrund der Anzahl an Eingabeparametern der Instanz, beschränkt.

Ein vollständiger Beweis für die Einordnung des Rucksackproblems in Entscheidungsform in NP-vollständig würde die Grenzen dieses Anhangs sprengen und kann in [7], S. 487 ff, nachgelesen werden. Es sei nur in Kürze erwähnt, dass das Subset Sum Entscheidungsproblem in polynomieller Zeit auf das Rucksack Entscheidungsproblem reduziert werden kann, wodurch die Zugehörigkeit des Rucksackproblems in Entscheidungsform zu NP-vollständig bewiesen wird. ([7], S. 491)

Das Rucksack Entscheidungsproblem kann zudem mit Hilfe von dynamischer Programmierung (siehe [7], S. 39 ff) in $O(nc)$ entschieden werden, wobei die Laufzeit sowohl in der Anzahl an Eingabe Koeffizienten n als auch in deren Größe c polynomiell ist. Das Rucksack Entscheidungsproblem kann somit in pseudopolynomieller Zeit gelöst werden und wird daher als schwach NP-schwer bezeichnet. ([7], S. 487)

Aus der Zuordnung des Rucksackproblems in Entscheidungsform kann zudem eine Einteilung des Problems in Optimierungsform erfolgen:
Kellerer, Pferschy und Pisinger [7], S.487, bezeichnen ein Optimierungsproblem als NP-schwer, wenn das zugehörige Entscheidungsproblem NP-vollständig ist.
Da das Rucksack Entscheidungsproblem zu den NP-vollständigen Problemen gehört, wird das Rucksack Optimierungsproblem daher als NP-schwer klassifiziert.

Überblick

Abbildung 19 bietet nochmals einen Überblick über die einzelnen Komplexitätsklassen, sowohl unter der Annahme dass P ≠ NP als auch unter der Annahme, dass P = NP:

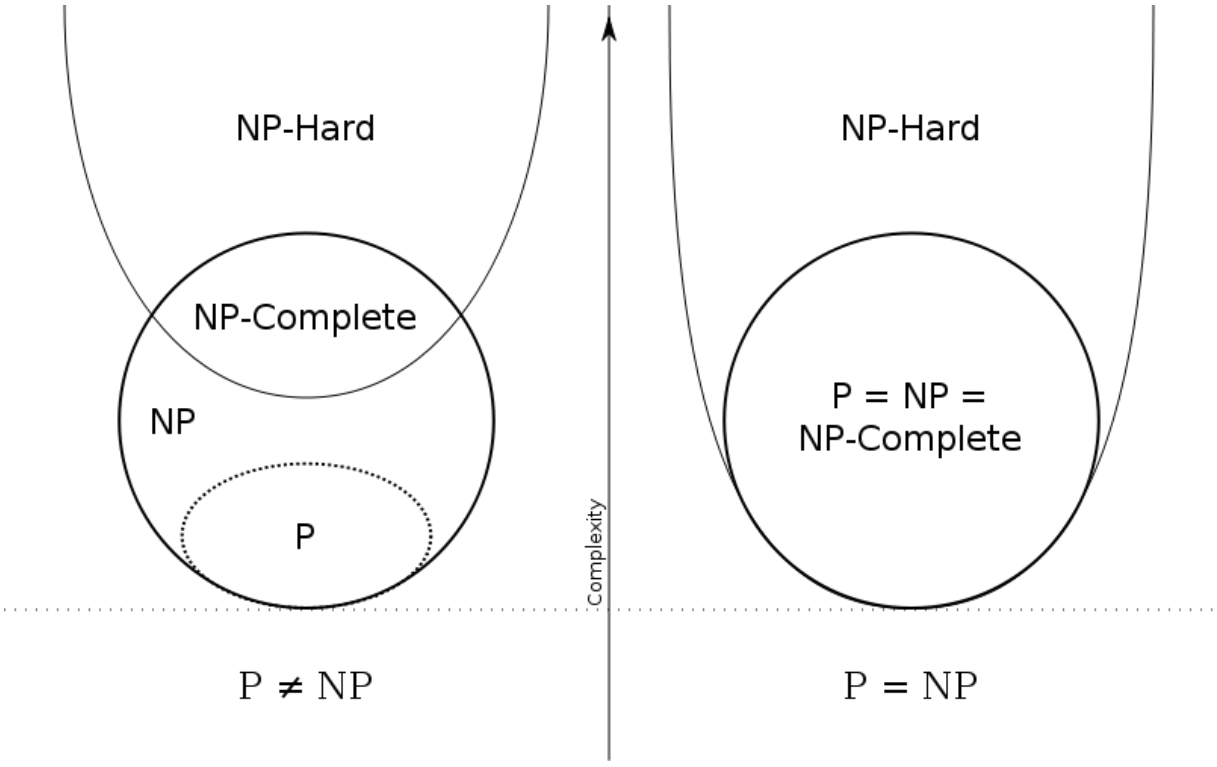

Abbildung 19: Euler Diagramm für P, NP, NP-vollständig, NP-schwer (Quelle [20])

Anhang C: Source Code

Der Source Code für die durchgeführten Messungen befindet sich auf der beigelegten CD im Verzeichnis /Code.

Es wurden folgende externe Codequellen verwendet:

- SettingsParser.cpp & SettingsParser.h: adaptiert von https://github.com/SFML/SFML/wiki/Source:-Settings-Parser
- csv.h: Library zum Lesen und Schreiben von CSV Dateien von https://github.com/ben-strasser/fast-cpp-csv-parser/blob/master/csv.h
- ctpl_stl.h: Threadpool library von https://github.com/vit-vit/ctpl

Anhang D: Daten und ausführbares Programm

Die Eingabedaten und Resultate liegen im .csv Format unter /Data in den Unterordnern /input und /output vor. Im /input Ordner befinden sich ebenfalls die Textdateien zur Parametrisierung der Algorithmen. Das Excelsheet „results.xlsx“ in /Data/output enthält die Zusammenfassung der Messdaten und deren grafische Auswertungen. Schließlich kann das Programm mit Hilfe der ausführbaren Datei „KnapsackProblem.exe“ mit den Parametern „<input.csv> <output.csv> <GA|DP|GR> <parameters.txt>“ aufgerufen werden.

Anhang E: Online Quellen

Alle für die Literaturrecherche verwendeten Websites und .pdf Dokumente wurden zur Persistierung der Daten heruntergeladen und sind unter /Online_Quellen verfügbar. Zum Öffnen der Websites muss die .html Datei in einem beliebigen Browser geöffnet werden.

Literaturverzeichnis

[1] M. Mitchell, An Introduction to Genetic Algorithms (Complex Adaptive Systems), 1. MIT Press Paperback Hrsg., The MIT Press, 1998.

[2] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, 3. Hrsg., Springer, 1996.

[3] R. Polli, W. B. Langdon und N. F. McPhee, „A Field Guide to Genetic Programming," Lulu Pr, 2008. [Online]. Available: http://www.gp-field-guide.org.uk/. [Zugriff am 11 November 2015].

[4] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, 1. Hrsg., Addison Wesley Longman, Inc., 1989.

[5] J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, 6. Hrsg., The MIT Press, 1998.

[6] M. Hristakeva und D. Shrestha, „Solving the Knapsack Problem with Genetic Algorithms," Simspon College, [Online]. Available: http://www.micsymposium.org/mics_2004/Hristake.pdf. [Zugriff am 11 November 2015].

[7] H. Kellerer, U. Pferschy und D. Pisinger, Knapsack Problems, 1. Hrsg., Springer, 2005.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest und C. Stein, Introduction to Algorithms, 3. Hrsg., The MIT Press, 2009.

[9] E. W. Weisstein, „Unimodal," MathWorld, [Online]. Available: http://mathworld.wolfram.com/Unimodal.html. [Zugriff am 2 December 2015].

[10] R. Riolo, J. H. Moore und K. M., „Genetic Programming Theory and Practice XII," 2015. [Online]. Available: http://www.springer.com/us/book/9781493903740. [Zugriff am 02 December 2015].

[11] Wikipedia, „John Koza Wikipedia Seite," [Online]. Available: https://en.wikipedia.org/wiki/John_Koza. [Zugriff am 16 November 2015].

[12] Amazon.de, „Produktbeschreibung zu David Goldbergs "Genetic Algorithms in Search, Optimization and Machine Learning",“ www.amazon.de, 2015. [Online]. Available: http://www.amazon.de/Genetic-Algorithms-Optimization-Machine-Learning/dp/0201157675. [Zugriff am 16 November 2015].

[13] „Data for the 01 Knapsack Problem,“ 17 August 2014. [Online]. Available: https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html. [Zugriff am 18 December 2015].

[14] L. Moura, „Introduction to the Theory of NP-Completeness,“ University of Ottawa, 2002. [Online]. Available: https://www.site.uottawa.ca/~lucia/courses/4105-02/np.pdf. [Zugriff am 18 November 2015].

[15] E. Stickel, H.-D. Groffmann und K.-H. Rau, Hrsg., Gabler Wirtschaftsinformatik Lexikon, Gabler Verlag, 1998.

[16] H.-J. Schneider, Hrsg., Lexikon der Informatik und Datenverarbeitung, Bd. 4., Oldenbourg Verlag, 1998.

[17] R. Reagan, „Number of Bits in a Specific Decimal Integer,“ 13 December 2012. [Online]. Available: http://www.exploringbinary.com/number-of-bits-in-a-decimal-integer/. [Zugriff am 21 November 2015].

[18] Clay Math Institute, „Rules for the Millenium Problems,“ 2015. [Online]. Available: http://www.claymath.org/millennium-problems/rules-millennium-prizes. [Zugriff am 18 November 2015].

[19] Clay Math Institute, „P vs NP Problem,“ 2015. [Online]. Available: http://www.claymath.org/millennium-problems/p-vs-np-problem. [Zugriff am 18 November 2015].

[20] B. Esfahbod, „P np np-complete np-hard.svg,“ Wikimedia Commons, 1 November 2007. [Online]. Available: https://commons.wikimedia.org/wiki/File:P_np_np-complete_np-hard.svg. [Zugriff am 18 November 2015].

Abbildungsverzeichnis

Tabellenverzeichnis