Universität
Augsburg
University

Fakultät für Informatik
Organic Computing

Analyse und Vergleich ausgewählter evolutionärer Algorithmen

Bachelorarbeit

Rafael Giebisch

Abgabedatum	6. April 2021
Matrikelnummer	1568963
Studiengang	Bachelor Informatik
Gutachter	Prof. Dr. Jörg Hähner
	Prof. Dr. Elisabeth André
Betreuerin	Helena Stegherr

Kurzfassung

In dieser Arbeit wird die grundsätzliche Funktionsweise der genetischen Algorithmen sowie der Evolutionsstrategien vorgestellt. Mithilfe von konkret erläuterten Implementierungen werden diese anhand drei verschiedener Optimierungsprobleme verglichen. Dazu gehören zwei zu optimierende Funktionen sowie ein kombinatorisches Optimierungsproblem, dem *Traveling Salesman Problem*. Die Algorithmen werden anhand festgelegter Kriterien bewertet und zur besseren Einschätzung der Ergebnisse noch mit einem einfachen Näherungsalgorithmus, dem *Bergsteigeralgorithmus*, verglichen.

Ich versichere hiermit, dass ich die hier vorliegende Arbeit selbstständig angefertigt habe und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe sowie wörtlich übernommene Stellen als solche kenntlich gemacht habe.

Ort, Datum	Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Metaheuristiken werden Algorithmen genannt, die näherungsweise Lösungen zu beliebigen Optimierungsproblemen bestimmen sollen. Gerade für NP-schwere Probleme, bei denen die benötige Rechenleistung für die Bestimmung einer exakten Lösung mit herkömmlichen Methoden zu hoch wird, wurden diese entwickelt. Dabei soll eine Lösung, die für den gewünschten Verwendungszweck gut genug ist, in angemessener Zeit gefunden werden [SG13b]. Durch langjährige Forschung dieser Metaheuristiken, deren Ursprung man bis in die 1940er Jahre zurückverfolgen kann, entstanden hier die verschiedensten Strategien [SG13a]. Die Metaheuristiken können hierbei in drei Kategorien, die jeweils für eine Art der Erstellung neuer Lösungen stehen, eingeteilt werden. So gibt es die „lokale Suche“, die „konstruktiven Metaheuristiken“ und die „populationsbasierten Metaheuristiken“, zu denen die evolutionären Algorithmen zählen [SG13b].

Auch die evolutionären Algorithmen können für jedes beliebige Problem verwendet werden. Gerade ihre schier unzählige Menge an möglichen Implementierungen führt jedoch dazu, dass nur schwer eingeschätzt werden kann, welche Implementierung für welche Art von Problemstellung gut geeignet ist. Hinzu kommt, dass Metaheuristiken bei einem Optimierungsproblem sehr gut, bei einem anderen, möglicherweise sogar sehr ähnlichem, durchaus auch relativ schlecht abschneiden kann. So ist es nicht verwunderlich, dass auch das Vergleichen verschiedener Implementierungen ein großes Forschungsgebiet darstellt [MPR18].

Mit dieser Arbeit soll hierfür ein Teil beigetragen werden. In Kapitel 2 wird zunächst auf die Grundlagen und bisherige Forschung von evolutionären Algorithmen eingegangen. Anschließend werden die genetischen Algorithmen und die Evolutionsstrategien in Kapitel 3 und 4 genauer erläutert und die verschiedenen Operatoren vorgestellt. Für den späteren Vergleich beinhaltet Kapitel 5 noch einen kurzen Einblick in die Funktionsweise des Hill climbings. Im darauffolgenden Kapitel 6 werden die verwendeten Optimierungsprobleme vorgestellt und Kriterien für die Bewertung der Algorithmen festgelegt. Es folgt die Analyse der Algorithmen sowie eine Gegenüberstellung der Ergebnisse. Im letzten Kapitel wird ein Fazit gezogen und ein Ausblick für weitere Forschungen gegeben.

2 Einführung in evolutionäre Algorithmen

Zunächst sollen für alle weiteren Untersuchungen die Grundlagen erläutert werden. Evolutionäre Algorithmen (EA) sind eine Klasse von Optimierungsverfahren, die ursprünglicherweise von Beobachtungen der biologischen Evolution inspiriert sind [GKK13]. Sie sollen näherungsweise Lösungen zu Optimierungsproblemen bestimmen. Dabei geht es meist darum, eine Funktion entweder zu mini- oder maximieren [Wei15].

Zu den bekanntesten Vertretern der evolutionären Algorithmen gehören die genetischen Algorithmen und die Evolutionsstrategien [DM97]. Neben diesen extistieren noch weitere Ausprägungen, wie beispielsweise die evolutionäre Programmierung oder die Differential Evolution [MPR18]. Auch Mischformen der gerade genannten Strategien sind möglich [GA07].

2.1 Prinzipieller Aufbau evolutionärer Algorithmen

Sie alle haben gemeinsam, dass es eine Gruppe von Lösungen gibt, Population genannt, die sich durch verschiedene Operationen weiterentwickeln und sich dadurch an eine Optimallösung annähern sollen. Die Ausgangspopulation wird dabei meist zufällig generiert. Die Evolution danach erfolgt iterativ. Zunächst werden die einzelnen potentiellen Lösungen, in Anlehnung an die Natur Chromosomen, oder auch einfach Individuen genannt, mithilfe einer sogenannten Fitnessfunktion bewertet. Dabei soll beurteilt werden, wie gut eine Lösung für das jeweilige Problem geeignet ist. Dadurch können später vielversprechendere Chromosomen bevorzugt und womöglich schlechtere benachteiligt werden. Man spricht von der sogenannten Selektion. Nachdem geeignete Chromosomen, auch Eltern genannt, gefunden wurden, werden diese mit verschiedenen Operatoren bearbeitet. Ziel ist es, verschiedenste Nachfolgechromosomen, die Kinder, zu erzeugen. So entsteht langsam eine neue Population, die wiederum dieselben Schritte durchläuft. Damit dies nicht ins Unendliche fortgesetzt wird, werden am Anfang Abbruchbedingungen festgelegt. Meist sind diese eine Kombination aus einer Maximalanzahl der Generationen und eine Obergrenze für die zeitliche Ausführung des Algorithmus. Für die Individuen muss vor Ausführung der Algorithmen eine passende Kodierung beziehungsweise Datenstruktur gefunden werden [MPR18].

Ablauf Evolutionärer Algorithmen [MPR18]

0. Finde eine passende Datenstruktur für die Lösungen, bereite eine Fitnessfunktion vor und setze $g = 0$
1. Generiere und evaluiere eine Startpopulation S_g
2. Wende Selektionsoperatoren auf S_g an und erzeuge dadurch die Elternpopulation P
3. Wende Variationsoperatoren auf P an, bewerte diese und erzeuge dadurch die Kinderpopulation C
4. Führe Populationen S_g und C mit ausgewählten Operatoren zusammen und erhalte $S_g + 1$, inkrementiere g
5. Falls keine der Abbruchbedingungen erfüllt sind, gehe zu 2. Sonst: Gib Chromosom mit bester Fitness zurück

2.2 Auswahl der Fitnessfunktion

Die Fitnessfunktion, anhand derer entschieden wird, welche Lösungskandidaten wie gut für ein gegebenes Problem geeignet sind, sollte mit Bedacht gewählt werden. So ist sie absolut anwendungsspezifisch und beeinhaltet außerdem eine Dekodierungsfunktion, häufig auch noch eine Skalierung der Zielfunktionswerte. Durch die gewählte Lösungsrepräsentation, der Dekodierungs- und Skalierungsfunktion lässt sich in hohem Maße die Schwierigkeit des Optimierungsproblems für evolutionäre Algorithmen beeinflussen [Nis97]. Beispiele für Kodierungen sind hierbei in den Sektionen 3 und 6 zu finden.

2.3 Bisherige Forschung

Bereits seit Ende der 1950er, Anfang der 1960er wird daran geforscht, Grundmechanismen der Evolution für praktische Einsätze zu übernehmen. So ist es nicht verwunderlich, dass hier bereits viele Untersuchungen durchgeführt wurden. Allerdings gestaltet sich der Leistungsvergleich zwischen evolutionären Algorithmen untereinander, aber auch der Vergleich mit anderen Optimierungsverfahren als ausgesprochen schwierig. Oft sind deshalb auch die Ergebnisse in der Literatur uneinheitlich [Nis97]. Vergleiche zwischen verschiedenen genetischen Operatoren gibt es bereits, zum Beispiel für Selektionen in [LYS17], Rekombinationen in [Soo+13] und Mutationen in [AAT12]. Auch zwischen den genetischen Algorithmen und den Evolutionsstrategien gibt es bereits Vergleiche, beispielsweise in [HB91]. Jedoch beschränken sich all diese Vergleiche auf bestimmte Probleme und bestimmte Implementierungen der Algorithmen, welche teilweise nicht einsehbar sind. Deswegen lohnen sich hier auch noch weitere Vergleiche, bei denen die Algorithmen auf unterschiedliche Weise implementiert werden. Außerdem soll darauf geachtet werden, die Komplexität der verwendeten Algorithmen so gleich wie möglich zu halten, was bei vorhandener Forschung nicht oder nur unzureichend kommentiert wurde.

3 Genetische Algorithmen

Die genetischen Algorithmen (GA) sind die populärsten Verfahren der EAs [MPR18]. Sie wurden erstmals in den 1960er Jahren von John Holland vorgestellt und seitdem vielfach weiterentwickelt. Genetische Algorithmen sind deshalb so beliebt, da sie zu den verschiedensten und komplexesten Problemstellungen Lösungen finden. So gibt es keine einschränkenden Anforderungen an die Zielfunktion und der GA kann auch bei Problemstrukturen, über die wenig bekannt ist, eingesetzt werden. Die einzelnen Varianten unterscheiden sich in der Implementierung der verschiedenen Operationen wie Selektion, Rekombination und Mutation, weshalb es möglich ist,

einen relativ allgemein gehaltenen Pseudocode für den Genetischen Algorithmus anzugeben [BL04].

Pseudocode Genetischer Algorithmus [MPR18]

```
Beginn
  t ← 0
  Generiere P(t)
  Berechne Fitnesswerte für P(t)
  Solange (nicht Abbruchtkriterium erfüllt)
  Beginn
    t ← t + 1
    Bilden der neuen Population P(t):
      Selektion
      Rekombination
      Mutation
    Berechne Fitnesswerte für P(t)
  Ende
Ende
```

Ursprünglich wurden die Lösungskandidaten binär kodiert, während dies heute nicht mehr der Standard ist. Auch kann bei der Implementierung auf feste Längen der Chromosomen bestanden werden oder diese variabel gehalten werden. Ersteres ist dabei üblicher [MPR18]. Die einzelnen Bits auf dem String werden, wieder in Anlehnung an die Natur, *Gene* genannt. Der eigentliche Wert beziehungsweise die konkrete Ausprägung der einzelnen Gene, bei einer binären Kodierung also entweder 0 oder 1, werden als *Allele* bezeichnet. Wenn die Individuen unterschiedlich zur tatsächlichen Lösung kodiert sind und somit zur Lösungsbewertung eine Dekodierung nötig ist, spricht man hier von einer Genotyp-Phänotyp Relation [Nis97]. Da in den in Kapitel 6 verwendeten Algorithmen die Lösungskandidaten alle reell kodiert wurden, ist hier später eine Dekodierung vor Bewertung mithilfe der Fitnessfunktion nicht erforderlich. Wichtigstes Merkmal der GA ist die Rekombination, meist auch Crossover genannt. Im Nachfolgenden wird auf die einzelnen Schritte genau eingegangen und verschiedene Arten vorgestellt. An dieser Stelle soll außerdem erwähnt werden, dass es auch Implementierungen gibt, bei denen automatisch eine festgelegte Anzahl an Chromosomen in die Nachfolgegeneration übernommen wird. Dies soll verhindern, dass sich durch Rekombination und Mutation die gefundene Bestlösung verschlechtert, weswegen einige Chromosomen mit der höchsten Fitness direkt übernommen werden [GKK13].

3.1 Population

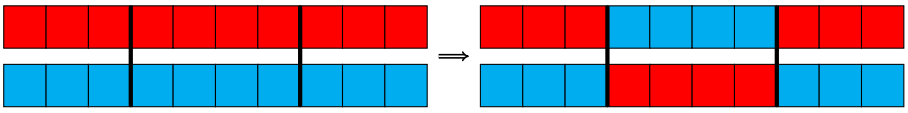
Die Ausgangspopulation des GA wird meist zufällig generiert. Dabei wird versucht, möglichst über den ganzen Wertebereich eine gleichmäßige Verteilung zu erhalten, um nicht von vornherein bestimmte Lösungen auszuschließen. Wenn diese zufällige Generierung der Lösungen nicht ausreicht, kann man zusätzlich noch eine Bias einführen. Die Populationsgröße kann selbst bestimmt werden, üblich sind Größen im hunderter Bereich [MPR18].

3.2 Selektion

Nachdem eine Population generiert wurde, werden Chromosomen für die Erzeugung der Nachkommen ausgewählt, wobei Chromosomen mit einer höheren Fitness bevorzugt werden sollen. Die Selektion ist damit traditionell der einzige Operator, der dafür sorgt, dass sich die Lösungskandidaten der Optimallösung annähern. Eine der bekanntesten Methoden für die Selektion ist die sogenannte *roulette-wheel method*. Die Methode orientiert sich an dem Spiel Roulette, wobei die Größe der Felder im Rouletterad proportional der Fitness der Chromosomen entsprechen. Es soll jedes Chromosom eine Chance haben, gewählt zu werden, um nicht zu schnell in ein lokales Optimum zu geraten, wenngleich Chromosomen mit einer besseren Fitness bevorzugt werden. Das Spiel wird so oft durchgeführt, wie Elternchromosomen benötigt werden [MPR18]. Sollten jedoch die Fitnesswerte zu sehr differieren, besteht das Risiko einer vorschnellen Konvergenz der Lösungen. Deswegen wurde die sogenannte *Rank based Selection* ausgearbeitet. Ähnlich wie bei der *roulette-wheel method* haben hier die Kandidaten mit einer höheren Fitness eine höhere Wahrscheinlichkeit gezogen zu werden, jedoch ist die Wahrscheinlichkeit nicht direkt proportional zum Fitnesswert. So werden die Individuen zunächst nach Fitnesswert sortiert und ihnen danach eine feste Wahrscheinlichkeit, abhängig von ihrem Rang, zugeordnet [Lin+10]. Die genauen Werte der Wahrscheinlichkeiten können beispielsweise linear oder exponentiell der Reihenfolge zugordnet werden [BT96]. Ein weiterer Selektionsoperator, der heutzutage häufig eingesetzt wird, ist die *Wettkampf-Selektion* bzw. *tournament selection*. Es werden aus der Population t_{size} viele Chromosomen zufällig gezogen, dannach wird deren Fitness verglichen. Einzig das Chromosom mit der besten Fitness wird ausgewählt. Dieser Vorgang wird sooft ausgeführt, wie Elternchromosomen benötigt werden. Vorteil ist, dass diese Art der Selektion sehr einfach umzusetzen ist, jedoch muss auf eine sinnvolle Wahl von t_{size} geachtet werden, welche die Bias gegenüber des besten Chromosoms festlegt [MPR18]. Häufig verwendet wird beispielsweise eine Größe von $t_{size} = 2$. Diese Art der *Wettkampf-Selektion* wird dann *binary tournament/binärer Wettkampf* genannt [BT96].

3.3 Crossover

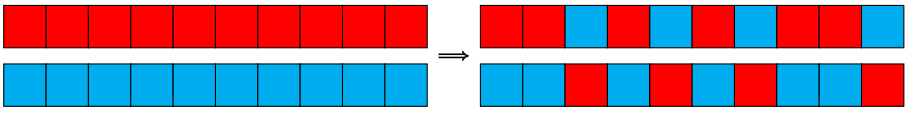
Die Mutation und das Crossover sind die beiden Operatoren des genetischen Algorithmus, die für die Erzeugung neuer Chromosomen verantwortlich sind. Crossover spielt beim GA dabei die übergeordnete Rolle, da die Mutation deutlich seltener zum Einsatz kommt. Beim Crossover werden zwei oder mehr Elternchromosomen miteinander kombiniert [MPR18]. Im Optimalfall sollen so die vorteilhaften Bestandteile der Eltern zusammengeführt werden [Wei15]. Häufig genutzte Crossoveroperatoren sind die *One Point*, *Two Point* oder sogar *N Point Crossover*. Dabei wird bei dem *One Point Crossover* ein Crossoverpunkt zufällig ausgewählt, respektive zwei oder N-viele bei den anderen. Für die Kinder werden dann die Gene der Eltern nacheinander übernommen, bis bei einem Crossoverpunkt zum jeweils anderen Elternstück gewechselt wird. In Abbildung 1 ist hierzu beispielhaft ein Schema für das *Two Point Crossover* dargestellt.



Eltern Kinder

Abbildung 1: Schema des *Two Point Crossovers* mit schwarz dargestellten, zufällig gewählten Crossoverpunkten.

Ein deutlich einfacheres Verfahren ist das Uniform Crossover. Dabei haben jeweils die beiden Gene an einer bestimmten Stelle der Eltern die gleiche Wahrscheinlichkeit übernommen zu werden. Insgesamt ist das Verhältnis der Gene eines Elternindividuums zum anderen deutlich ausgeglichener. In Abbildung 2 ist diese Art des Crossovers schematisch dargestellt.



Eltern Kinder

Abbildung 2: Beim *Uniform Crossovers* besitzt jedes Gen eine 50%-ige Chance gewählt zu werden.

Das zweite Kind übernimmt immer die übriggebliebenen Gene und ist somit quasi immer das Gegenteil des ersten Kindes [Soo+13].

Das Ziel bei kombinatorischen Optimierungsproblemen ist typischerweise eine geeignete Permutation beziehungsweise Anordnung von gegebenen Einträgen zu finden [MPR18]. Da im Rahmen dessen keine Gene hinzugefügt oder entfernt werden dürfen, sind andere Crossoveroperatoren nötig. Eine Möglichkeit stellt das *Order Crossover (OX2)* dar. Dabei wird ein Teilstück eines Elternchromosoms kopiert. Die restlichen Plätze werden dann mit den noch fehlenden Genen in der Reihenfolge aufgefüllt, wie sie im zweiten Elternchromosom vorkommen [Dee11]. Zur Veranschaulichung ist hier Abbildung 3 gegeben.

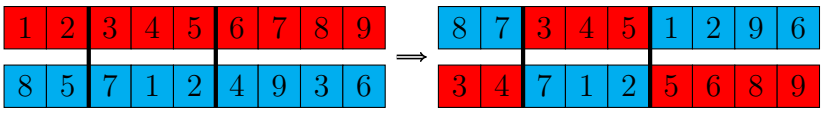


Abbildung 3: Beim *Order Crossover* wird ein Teilstück des Chromosoms übernommen und die restlichen Gene in der Reihenfolge übernommen, wie sie im zweiten Elternchromosom vorkommen.

Ein weiterer Operator für kombinatorische Probleme ist das *Partially Mapped Crossover*. Zunächst werden zwei Crossoverpunkte bestimmt und der jeweilige Teil dazwischen gegenübergestellt. Jedes Gen des einen Stückes wird dem an der gleichen Stelle befindlichen Gen des anderen Stücks zugewiesen. Nun werden die zwei Teilstücke übernommen und die restlichen Gene des anderen Elternteils einzeln kopiert. Ist ein Gen bereits im neuen Chromosom vorhanden, wird es mithilfe des vorhin erstellten „Mappings“ ersetzt. Sollte auch dieses bereits vorkommen, wird auch dieses wieder weiter ersetzt [US15]. Ein Beispiel ist hierfür in Abbildung 4 zu sehen.

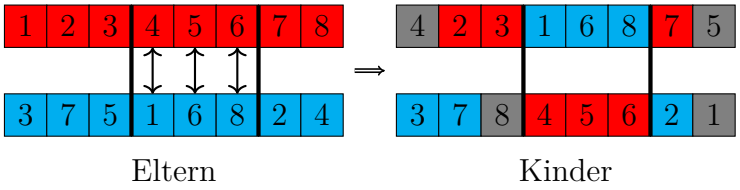


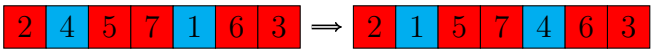
Abbildung 4: Beim *Partially Mapped Crossover* werden die Teilstücke zwischen den Crossoverpunkten aufeinander gemappt. Grau dargestellt sind die Gene, die durch das Mapping ausgetauscht werden.

3.4 Mutation

Da durch Rekombination von Chromosomen keine neuen Genbelegungen erzeugt werden können, aufgrund der Tatsache, dass nur die Teilbereiche des Suchraums erreicht werden können, die bereits in der Ausgangspopulation erhalten sind, bedarf es eines weiteren genetischen Operators, der Mutation [Wei15]. Die einfachste Methode der Mutation ist das sogenannte *bit flipping* für binäre Strings. Hierbei wird ein zufälliger Index im String bestimmt und dessen Bit invertiert.

Für reelle Kodierungen hingegen muss auf andere Mutationsoperatoren zurückgegriffen werden. Eine Methode ist hierfür die *Gaussian Mutation*. Bei dieser Mutation werden den ausgewählten Genen mit einer vorher festgelegten Wahrscheinlichkeit ein zufälliger Wert der Normalverteilung mit Erwartungswert $= 0$ hinzu addiert. Hierzu entstehen neue Werte, bei denen allerdings darauf geachtet werden muss, dass diese den Definitionsbereich nicht verlassen [Hin95]. Eine weitere Mutationsmethode ist die *Uniform Mutation*. Im Falle einer Mutation eines Genes wird der vorhandene Wert einfach durch einen komplett neuen Zufallswert ersetzt. Auch hier muss darauf geachtet werden, dass sich dieser im Definitionsbereich befindet [GKK13].

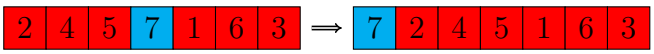
Für kombinatorische Probleme, bei denen verschiedene Werte in einem Chromosom nur einmal vorkommen dürfen, bedarf es anderer Mutationsoperatoren. Hierbei können beispielsweise *swappings* oder *insertions* vorgenommen werden. Bei der *Swapping Mutation* wird dabei die Position von zwei Genen im Chromsom getauscht, diese werden zufällig bestimmt. In Abbildung 5 ist hierfür ein Beispiel gegeben.



Vor Mutation Nach Mutation

Abbildung 5: Die zwei zufällig ausgewählten Gene, hier Blau markiert, tauschen bei der *Swapping Mutation* ihren Platz.

Bei der *Insertion Mutation* hingegen wird nur ein Gen zufällig ausgewählt. Dieses wird an eine neue, zufällige Stelle im Chromosom verschoben. Alle anderen Gene rücken dann einfach nach [MPR18]. Ein Beispiel hierfür ist in Abbildung 6 zu sehen.



Vor Mutation Nach Mutation

Abbildung 6: Ein zufällig ausgewähltes Gen wird bei der *Insertion Mutation* an eine andere, zufällige Stelle verschoben.

4 Evolutionsstrategien

Eine weitere Methode der evolutionären Algorithmen sind die Evolutionsstrategien (ES), die in den 1960er Jahren von Rechenberg und Schwefel entwickelt wurden [GKK13]. Wie auch andere EA ahmen die Evolutionsstrategien die biologische Evolution nach. Bei der Suche nach einer Optimallösung werden dabei Variations- und Selektionsoperatoren im Wechselspiel verwendet. Das Besondere der ES sind jedoch die sich selbst justierenden Mutationsoperatoren, die sich auf das gegebene Problem einstellen können und somit einen schnelleren Fortschritt bei der Suche ermöglichen sollen. Die dafür zuständigen Parameter werden Strategieparameter genannt. Bei den Evolutionsstrategien unterscheidet man zwischen zwei Strategien, der Komma-Strategie und der Plus-Strategie, auf die beide im Weiteren eingegangen werden soll [MPR18]. Insgesamt soll immer eine reellwertige Funktion $f : \mathbb{R}^n \to \mathbb{R}$ optimiert werden, entsprechend bestehen auch die Chromosomen aus reellwertigen Genen [GKK13].

Grundsätzlich sind die Evolutionsstrategien genau wie die genetischen Algorithmen populationsbasierte Verfahren. Auch hier werden sukzessive Nachfolgepopulationen generiert und mittels Rekombinationsoperatoren neue Individuen erzeugt. Im Unterschied zu den GA wird hier aber zunächst eine Nachfolgepopulation erstellt, die größer als die Elternpopulation ist [Hem97]. Das μ steht hier für die Anzahl von Individuen in der Elternpopulation und das λ für die Anzahl von Individuen in der Kinderpopulation [MPR18]. Ein übliches Verhältnis von $\mu : \lambda$ beträgt $1 : 7$, was bedeutet, dass jedes Chromosom im Mittel 7 Kinder erzeugt. Typische Werte sind $\mu = 15$ und $\lambda = 105$ [GKK13]. So wird aus den μ Individuen eine Population von λ Individuen mittels Rekombination und anschließender Mutation gebildet. Nun sollen wieder nur die besten μ Individuen ausgewählt werden. Bei der Komma-Strategie, oft auch unter der (μ, λ)-Evolutionsstrategie zu finden, wird hierbei nur aus den neu erstellten Individuen gewählt. Entgegen dazu wird bei der Plus-Strategie, analog $(\mu + \lambda)$-Evolutionsstrategie genannt, aus der Vereinigung der Elternindividuen und Kinderindividuen gewählt. Diesen Schritt nennt man auch Selektion, darf aber mit der vorher genau erläuterten Selektion bei den genetischen Algorithmen nicht verwechselt werden. Aus diesem Grund wird ab sofort die neu eingeführte Selektion, die die λ Individuen auf μ Individuen reduziert, als *ES-Selektion* bezeichnet [Hem97]. Vorteil der Plus-Strategie ist, dass sichergestellt werden kann, dass die besten Individuen in die nächste Generation übernommen werden. Dies birgt allerdings die Gefahr einer frühzeitigen Konvergenz in ein lokales Optimum, weswegen auch Mischformen der beiden Strategien anzutreffen sind [GKK13].

Insgesamt ergibt sich daraus dann folgender Pseudocode:

Pseudocode Evolutionsstrategien [MPR18]

```
Beginn
  t ← 0
  Generiere P(t)
  Berechne Fitnesswerte für P(t)
  Solange (nicht Abbruchtkriterium erfüllt)
  Beginn
    t ← t + 1
    Bilden der neuen Population P(t):
      Rekombination (λ Kinder werden aus μ generiert)
      Mutation
      Berechne Fitnesswerte
      ES-Selektion (besten μ Individuen werden auserwählt,
        entweder nur aus den Kindern (Komma-Strategie)
        oder aus Kindern+Eltern (Plus-Strategie))
  Ende
Ende
```

Bei den ES ist die Mutation der wichtigere Operator, im Vergleich zu der Rekombination der GA [MPR18]. Einige einfache Implementierungen der ES besitzen sogar nur den Mutationsoperator und verzichten auf die Rekombination. Hierbei werden dann bei der Mutation die λ Kinder generiert [GKK13]. Für die Rekombination existieren, wie auch bei den genetischen Algorithmen, verschiedene Arten. Auch hier können wieder verschiedene *N-Point Crossover* verwendet werden. Dadurch, dass die Gene der Chromosomen immer reellwertig sind, ermöglicht sich noch eine weitere Art der Rekombination, die *intermediäre Rekombination*. Dabei wird für jedes Gen des Kindchromosoms der Mittelwert der entsprechenden Gene der Elterchromosomen gebildet.

Bei der Mutation eines Chromosoms wird jedem Gen eine nach einer Normalverteilung mit Erwartungswert 0 und kleiner Streuung δ erzeugte Zufallszahl addiert. Die Streuung δ, auch Mutationsschrittweite oder *Stepsize* genannt, soll dabei stets so klein sein, dass das Kindchromosom sich nicht zu sehr vom Elternchromosom unterscheidet, und somit nicht ein komplett neues Chromosom, das auch zufällig hätte generiert werden können, entsteht. Die Zielfunktion kann in der Umgebung eines Chromosoms stark variieren, wofür eher eine kleine Stepsize δ sinnvoll wäre, während die Zielfunktion in der Umgebung eines anderen Chromosoms eventuell kaum variiert, also auch größere Werte von δ sinnvoll wären. Deswegen legt man δ nicht als eine globale Konstante an, sondern weist den einzelnen Chromosomen jeweils ihre eigenen Strategieparameter zu, welche sich über den weiteren Verlauf auch noch ändern können [GKK13]. Diese selbst-Adaptierung unterscheidet die ES

von den meisten anderen EA [MPR18]. Zur Auswahl steht hier die *deterministische Adaption* und die *dynamische Adaption*. Die determinstische Adaption fängt mit einer eher großen Mutationsschrittweite an, um am Anfang möglichst den ganzen Suchraum abzudecken. Diese Mutationsschrittweite wird dann deterministisch immer kleiner, um letztendlich eine lokale Optimierung zu ermöglichen. Bei der dynamischen Adaption wird im Vergleich zur deterministischen Adaption die Mutationsschrittweite nicht nur verkleinert, sondern kann auch im Laufe der Evolution wieder größer werden, wenn beispielsweise δ zu klein gewählt wurde. Ein Ansatz, die Schrittweite dynamisch zu regeln ist die sogenannte Rechenbergs 1/5-Regel. Diese besagt, dass wenn ein Fünftel der Mutationen erfolgreich sind, die Stepsize optimal gewählt ist. Wenn es weniger sind, sollte die Mutationsschrittweite verringert, wenn es mehr sind vergrößert werden. Eine weitere Möglichkeit die Mutationsschrittweiten dynamisch zu regeln besteht darin, diese einfach als weitere zusätzlich zu optimierende Parameter für alle Chromosomen aufzunehmen.

$$c = (x_1, ..., x_n, \delta_1, ..., \delta_n)$$

Die Mutation der Mutationsschrittweiten erfolgt dann nach folgender Formel:

$$\delta_i^{neu} = \delta_i^{alt} \cdot \exp(\tau_1 \cdot R + \tau_2 \cdot R_i)$$

Hierbei ist R eine am Anfang der ES, nach Standardnormalverteilung erzeugte Zufallszahl. R_i ist eine nach Standardnormalverteilung, aber für jedes δ_i neu erzeugte, Zufallszahl [GKK13]. τ_1 und τ_2 sind zwei feste Werte, die mit folgenden Beispielbesetzungen belegt werden können:

$$t_1 \approx \frac{1}{\sqrt{2n}}$$
$$t_2 \approx \frac{1}{\sqrt{2\sqrt{n}}}$$
[BS93]

oder

$$t_1 \approx 0.1$$
$$t_2 \approx 0.2$$
[Nis97]

Mit dieser Gleichung wird garantiert, dass die Streuungen δ_i immer positiv bleiben und der Erwartungswert des Wertes in der Exponentialfunktion bei 0 liegt. Damit ergibt sich eine mittlere Änderung der δ_i-Werte von 0. Um zu verhindern, dass Chromosomen vorzeitig konvergieren, kann auch eine untere Schranke für die δ-Werte eingeführt werden. Somit verhindert man, dass sich Chromosomen einen Selektionsvorteil verschaffen, indem sie sich auf Stellen in der Nähe eines lokalen Optimums konzentrieren [GKK13].

5 Hill climbing

Das Hill climbing (*Bergsteigeralgorithmus*) ist ein einfaches heuristisches Optimierungsverfahren. Es soll später lediglich als zusätzlicher Vergleich dienen, weswegen hier nur die Grundzüge erläutert werden sollen. Es wird zunächst mit einer zufällig generierten Anfangslösung gestartet. Danach werden alle „Nachbarn“ dieser Lösung, bei der Optimierung einer einfachen Funktion, beispielsweise links und rechts davon, untersucht und die gemäß der Bewertungsfunktion beste ausgewählt. Dies wiederholt sich solange, bis keine bessere Lösung mehr gefunden wird. Das Verfahren ist dann entweder am globalen, meist allerdings in einem lokalen Optimum stecken geblieben. In der einfachsten Form findet dieser Algorithmus sehr schnell gute Lösungen, um jedoch dem lokalen Optimum zu entkommen, müssen weitere Techniken wie das *Simulated Annealing*, *Tabu Search* oder das *Guided Local Search* verwendet werden [MPR18].

Pseudocode Hill climbing [PBE08]

```
Beginn
  i ← zufällig generierte Startlösung
  Wiederhole
    generiere ein s ∈ N(i) (Nachbarn von i)
    Falls f(s) > f(i) dann i ← s
  Bis f(s) ≤ f(i) für alle s ∈ N(i)
Ende
```

6 Vergleich der Metaheuristiken

Nachdem nun die Grundlagen ausreichend erläutert wurden, sollen nun die verschiedenen Metaheuristiken weiter analysiert und verschiedene Implementierungen miteinander verglichen werden.

6.1 Aufbau einer Testumgebung

Um die Heuristiken miteinander zu vergleichen, werden diese anhand von zwei unterschiedlichen Arten von Optimierungsproblemen bewertet, welche nun vorgestellt werden. Als erstes soll das Optimieren einer Funktion untersucht werden. Dies gehört zu den klassischen Aufgaben der evolutionären Algorithmen und so haben sich über den Verlauf der Zeit verschiedene Standardfunktionen bewährt, die alle evolutionären Algorithmen lösen können müssen. Eine davon ist die *Parabel-Funktion* f_1.

$$f_1(x) = \sum_{i=1}^{3} x_i^2 \qquad (\forall i \in \{1,2,3\} : -5.12 < x_i < 5.12)$$
$$min(f_1) = f_1(0,0,0) = 0$$

Hier soll das globale Minimum $min(f_1) = 0$ gefunden werden. Dieses Optimierungsproblem sollte relativ leicht zu lösen sein, da keine anderen lokalen Minima, außer dem globalen Minimum, existieren. Es wird jedoch als Grundlage dienen, denn wenn eine Implementierung schon hier keine gute Lösung findet, so wird auch bei komplexeren Aufgabenstellungen keine gute Lösung erwartet. Als zweite zu optimierende Funktion soll die etwas komplexere *Rosenbrock Funktion* f_2, auch *Rosenbrock's Sattel* genannt, dienen [GKK13].

$$f_2(x) = 100 \cdot (x_2^2 - x_1^2)^2 + (1 - x_1)^2 \qquad (\forall i \in \{1,2\} : -2.048 < x_i < 2.048)$$
$$min(f_2) = f_2(1,1) = 0$$

Die Funktion zeichnet sich durch ein parabelförmiges Tal aus, das leicht gefunden werden kann, allerdings dannach meist noch mit kleinen Schritten durchquert werden muss. Das gesamte Tal befindet sich relativ nahe am globalen Minimum, dessen genaue Bestimmung sich jedoch als schwierig erweist [Fob06].

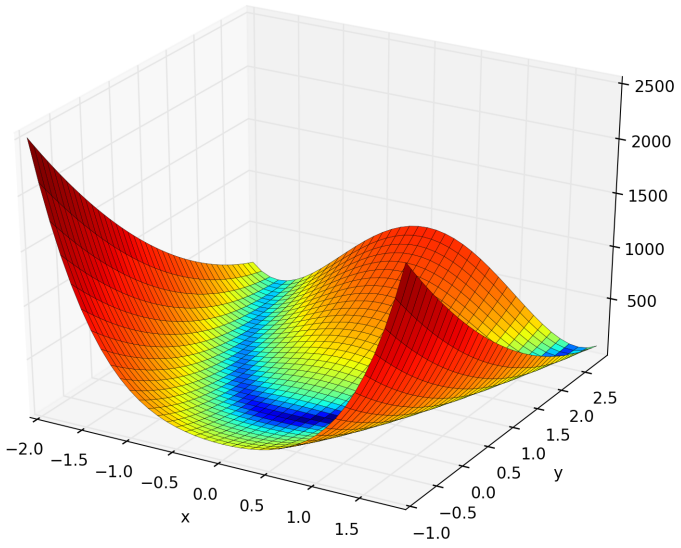


Abbildung 7: Rosenbrock Funktion mit blau gefärbtem Tal [Pyt].

Als zweite Art von Optimierungsproblemen sollen die Algorithmen auf kombinatorische Optimierungsprobleme untersucht werden. Das Ziel bei kombinatorischen

Optimierungsproblemen ist typischerweise eine geeignete Permutation beziehungsweise Anordnung von gegebenen Einträgen zu finden [MPR18]. Stellvertretend für diese Art wird hier das sogenannte *Traveling Salesman Problem*, auch als das *Problem des Handlungsreisenden* bekannt, weiter analysiert. Hier plant ein Handlungsreisender eine Rundreise durch n Städte, so dass die Reise möglichst kurz ist. Jede Stadt soll genau einmal besucht werden und der Handlungsreisende am Ende wieder an der ersten Stadt ankommen [GKK13]. Für die Entfernungen ist eine Distanzmatrix $D = [d_{i,j}]$ gegeben, die die Distanzen zwischen allen zwei Städten i und j beinhaltet. Mit $\pi(i)$ als Nachfolgerfunktion, also die Stadt, die nach Stadt i besucht wird und $i = 1, ..., N$, kann die gesamte Entfernung als

$$g(\pi) = \sum_{i=1}^{n} d_{i\pi(i)}$$

definiert werden. Diese Funktion $g(\pi)$ gilt es durch korrekte Reihenfolge der Städte zu minmieren [MPR18]. Statt einer Distanzmatrix können auch die Koordinaten der Städte gegeben sein, hier lassen sich die euklidischen Abstände aber leicht berechnen. Als Beispielproblem für das *TSP* soll hier das *eil101.tsp* dienen, das Koordinaten von 101 verschiedenen Städten beinhaltet.

Um die Ergebnisse der Testläufe miteinander vergleichen zu können, wird nun festgelegt, anhand welcher Kriterien die Metaheuristiken bewertet werden. Als wichtigstes Kriterium wird dabei die Qualität der Lösung bewertet. Da bei allen verwendeten Optimierungsproblemen die genaue Lösung bekannt ist, kann hier einfach die Differenz zu eben dieser betrachtet werden. Darüber hinaus soll auch die Schnelligkeit der Algorithmen verglichen werden. Je schneller eine „gute“ Lösung gefunden wird desto besser. Um den Zusammenhang zwischen Qualität und Aufwand zu messen, bietet sich das Verhältnis der Zeit, die benötigt wird, eine Lösung innerhalb der besten 5% zu finden, zur Zeit, die die beste Lösung brauchte, an.

$$r_{0.05} = \frac{\text{Zeit für Lösung innerhalb der besten 5\%}}{\text{Zeit für beste Lösung}}$$

Desweiteren soll die Robustheit der Algorithmen gegenüber verschiedenen Problemen verglichen werden sowie die Komplexität der Implementierung, also wie einfach oder schwer der Algorithmus zu implementieren ist. Alle Tests werden wiederholt durchgeführt, um zu zeigen, dass die Testergebnisse jederzeit repliziert werden können. Außerdem soll dadurch verhindert werden, dass lediglich Ausreißer miteinander verglichen werden. Auch werden die Tests, soweit möglich, in zufälliger Reihenfolge ausgeführt, um auch nicht berücksichtigte, externe Störungsfaktoren auszuschließen. Die Experimente werden also ganz nach den DOE (*Design of experiments*) durchgeführt [Bar+95]. Zur Messung der Zeit wurden außerdem alle Tests auf dem gleichem System durchgeführt (Windows 10 64-Bit, Intel Core i7-2600 @ 3.40 GHz, 8.00 GB DDR3 RAM).

6.2 Analyse genetischer Algorithmen

Im Folgenden werden zunächst verschiedene genetische Algorithmen genauer untersucht. Es werden jeweils zwei verschiedene Selektionen, Rekombinationen und Mutationen, sowie verschiedene Kombinationen dieser auf die oben genannten Testfunktionen angewendet.

Tabelle 1: Die verschiedenen Selektionen, Rekombinationen und Mutationen, die vorgestellt wurden und nun miteinander verglichen werden

Selektion	Rank based Selection	RBS
	Tournament Selection	TS
Crossover	Two Point Crossover	TPC
	Uniform Crossover	UC
Mutation	Uniform Mutation	UM
	Gaussian Mutation	GM

Für alle Testdurchläufe wurde eine Populationsgröße von $n = 30$ festgelegt, sowie eine Crossoverwahrscheinlichkeit von $p_c = 0.9$ und eine Mutationswahrscheinlichkeit von $p_m = 0.1$. Eine umfassende Untersuchung geeigneter Parametereinstellungen würde eine eigene Arbeit für sich beanspruchen, weshalb hier auf Werte, die in vorhandener Literatur verwendet wurden, zurückgegriffen wird [Has+19]. Diese wurden anschließend durch empirische Tests für gut genug empfunden.

Für die zwei Testfunktionen f_1 und f_2 werden die Individuen als Arrays von floats kodiert, da eine sehr hohe Genauigkeit beim Optimieren von Funktionen erwartet wird, die über eine binäre Kodierung nur sehr unhandlich verwirklicht werden kann. Desweiteren gilt eine vorgegebene Obergrenze von $g = 100$ Generationen als Abbruchkriterium. Auf eine zeitliche Obergrenze wurde hier verzichtet, da die verschiedenen Implementierungen fast identische Rechenzeit pro Generation benötigen, mehr dazu in Tabelle 2. Für mehr Details über die konkrete Implementierung der genetischen Algorithmen ist der komplette Code im Anhang zu finden.

In Abbildung 8 ist die Optimierung von f_1 zu sehen. Dafür wurde jeweils der Durchschnitt aus 10 Läufen abgebildet. Es ist sofort zu erkennen, dass alle Implementierungen ein sinnvolles Endergebnis liefern, da sich alle der Geraden $min(f_1) = 0$ nähern. Wie vorhin schon erwähnt, ist die benötigte Rechenzeit für diese Durchläufe bei allen Implementierungen vergleichbar, wie in Tabelle 2 zu sehen, der maximale Unterschied von der geringsten Durchschnittszeit zur größten war eine Steigerung von $\approx 9.7\%$. Ins Auge fällt hier jedoch, dass Implementierungen mit der *Tournament Selection* konsequent die besten Zeiten lieferten. Die einfache Funktionsweise der *Tournament Selection* ermöglicht also nicht nur eine einfache und schnelle Imple-

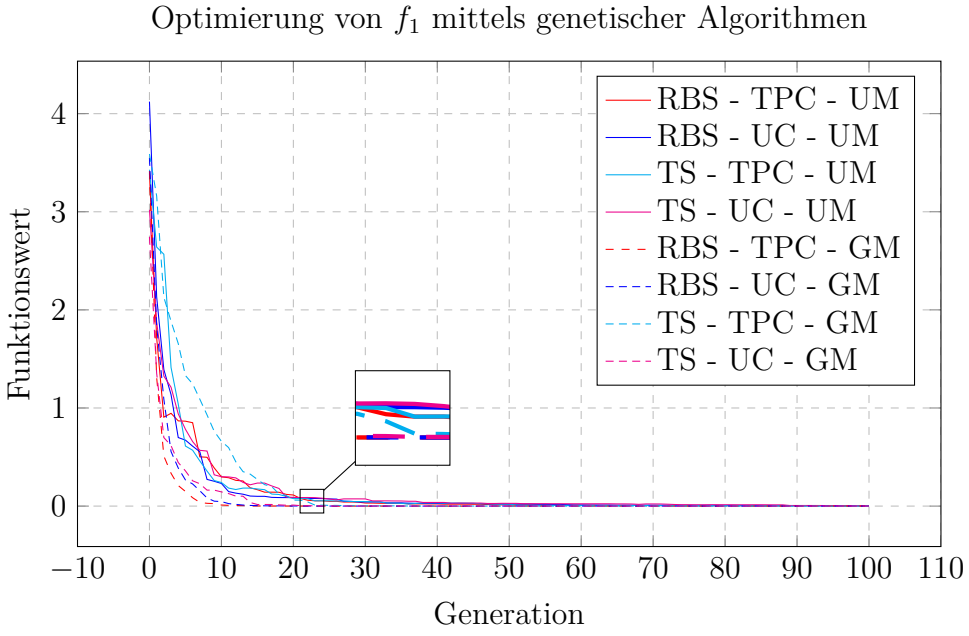


Abbildung 8: Vergleich bei der Optimierung von f_1 mithilfe aller möglichen Kombinationen der in Tabelle 1 gelisteten Implementierungen.

mentierung, sondern auch eine geringe Laufzeit. Gerade das wiederholte Auswählen der Chromosomen basierend auf deren Fitness bei der *Rank based Selection* erweist sich im Vergleich dazu als aufwendiger zu implementieren und benötigt mehrere Zwischenschritte, wie dem Emulieren des Rouletterads, was wiederum zu längerer Laufzeit führt.

Auf den in Tabelle 2 angegebenen maximalen Speicherbedarf der Implementierungen soll im Weiteren nicht eingangen werden, da nicht garantiert werden kann, dass die kleinen Unterschiede nicht von Python, der Programmiersprache, die für die Tests verwendet wurde, selbst stammen. Die Schwankungen könnten beispielsweise durch Pythons Interpreter oder von externen Hilfsfunktionen kommen, die hier gar nicht verglichen werden sollen. Hier würde sich aber eine weitere Studie mit einer anderen Programmiersprache lohnen. Die Ergebnisse sind dennoch abgebildet, um zu beweisen, dass der Aufbau und die Komplexität der Implementierungen grundsätzlich sehr ähnlich sind.

Wie in Abbildung 8 zu sehen, verbessern sich alle verwendeten Implementierungen der GA in den ersten paar Generationen vergleichbar schnell. Ein bedeutenswerter Unterschied entsteht jedoch bereits bei etwa Generation 23, bei der alle Implementierungen mit der *Gaussian Mutation*, in der Grafik mit gestrichelten Linien dargestellt, bereits bessere Ergebnisse erzielen konnten als deren Gegenstücke mit *Uniform Mu-*

Tabelle 2: Durchschnittliche Ergebnisse nach jeweils 10 Durchläufen für f_1 (gerundet).

Implementierung	Evaluationen	Ausführungszeit	Speicherauslastung (MiB)
RBS - TPC - UM	5825	270ms	23.0976
RBS - UC - UM	5832	277ms	23.1796
TS - TPC - UM	5808	258ms	23.0507
TS - UC - UM	5857	264ms	23.1484
RBS - TPC - GM	5824	277ms	23.2148
RBS - UC - GM	5831	283ms	23.1718
TS - TPC - GM	5834	264ms	23.1640
TS - UC - GM	5807	268ms	23.2304

tation. Dies dürfte auch im direkten Vergleich der Funktionsweisen ersichtlich werden. Da bei der *Uniform Mutation* ein neuer Zufallswert im Definitionsbereich generiert wird und das Gen dadurch ersetzt wird, gehen dadurch sämtliche Fortschritte verloren, während bei der *Gaussian Mutation* ein Zufallswert in der Nähe des alten Gens erzeugt wird und sich die Lösung so mit kleinen Schritten der Ideallösung nähern kann. Dies wirkt sich auch auf die gefundene Endlösung der Implementierungen aus.

Tabelle 3: Durchschnittliche Ergebnisse nach jeweils 10 Durchläufen für f_1 (gerundet).

Implementierung	Endergebnis	Varianz σ^2	Standard-abweichung σ	Median
RBS - TPC - UM	0.00234811	0.000006045326	0.002458724	0.00145901
RBS - UC - UM	0.00484236	0.000028577849	0.005345825	0.00365604
TS - TPC - UM	0.00240084	0.000007675687	0.002770503	0.00132486
TS - UC - UM	0.00410444	0.000008102363	0.002846465	0.00394301
RBS - TPC - GM	0.00000916	0.000000000048	0.000006931	0.00000992
RBS - UC - GM	0.00002522	0.000000000435	0.000020859	0.00002824
TS - TPC - GM	0.00002605	0.000000000246	0.000015676	0.00002257
TS - UC - GM	0.00002702	0.000000000560	0.000023672	0.00003235

Wie in Tabelle 3 zu sehen ist, liegen die durchschnittlichen Endergebnisse, die mittels *Gaussian Mutation* gefunden wurden, um ein vielfaches näher an der Optimallösung. Zugleich sehen wir auch eine geringere Varianz der Ergebnisse. Dies dürfte ebenfalls damit zusammenhängen, dass bei der *Uniform Mutation* Zufallszahlen aus einem viel größeren Bereich gezogen werden als es bei der *Gaussian Mutation* der Fall

ist. Direkt davon abhängig schwankt also auch das Endergebnis dementsprechend mehr. Zumindest bei diesem Optimierungsproblem ist also die *Gaussian Mutation* der *Uniform Mutation* eindeutig vorzuziehen.

Beim Crossover werden die Implementierungen mit *Two Point Crossover* mit deren Gegenstücke mit *Uniform Crossover* verglichen. Hier ist aus Tabelle 3 schnell zu erkennen, das *Two Point Crossover* stets bessere Endergebnisse liefert, dies steht auch im Einklang mit vorhandenen Untersuchungen [Soo+13]. Zumindest bei diesem Test konnte dabei auch eine geringere Varianz festgestellt werden.

Im Vergleich zu den letzten zwei Ergebnissen ist der Unterschied bei der Selektion zwischen *Rank based Selection* und *Tournament Selection* weitaus geringer. Hier waren, wie in Tabelle 3 zu sehen ist, Implementierungen mit der *Rank based Selection* aber immerhin in drei von vier Durchläufen schneller, in welchen sie auch eine geringere Varianz aufwies.

Durch relativ ähnliche Rechenzeit und Speicherlauslastung kann also nach diesem ersten Test die Hauptkriterien als Endergebnis und Varianz festgelegt werden. Die Kombination aus *RBS - TPC - GM* erwies sich dabei als besonders Präzise. Sie lieferte das beste durchschnittliche Endergebnis mit der geringsten Varianz. Nun sollen die Implementierungen noch auf f_2 angewendet werden, um die Ergebnisse bestätigen zu können.

Für die Funktion f_2 wurden die selben Parameter verwendet, lediglich die Anzahl der Generationen wurde aufgrund der deutlich höheren Komplexität auf 500 gesetzt. In Abbildung 9 ist nun, parallel zur obigen Abbildung, die Optimierung von f_2 mittels der verschiedenen genetischen Algorithmen abgebildet. Dabei wurde jeweils der Durchschnitt aus 20 Durchläufen verwendet.

Anders als bei der Optimierung von f_1 ist nun zu erkennen, dass sich die Implementierungen mit *Gaussian Mutation*, wieder gestrichelt dargestellt, langsamer an die Ideallösung angleichen. Dies hängt damit zusammen, dass sich Implementierungen mit der *Gaussian Mutation* langsam durch das Tal bewegen müssen, während bei der *Uniform Mutation* jedes Mal neue „Positionen“ zufällig generiert werden. Dabei kommt es unweigerlich vor, dass relativ schnell eine Lösung gefunden wird, die nahe an der Optimallösung liegt, während die *Gaussian Mutation*, falls die Ausgangspopulation keine gute Lösung hervorbrachte, sehr lange braucht, um eine Annäherung zu finden. Dennoch sehen wir auch hier, dass letztendlich die *Gaussian Mutation* bessere Endergebnisse brachte. Wie in Tabelle 5 zu sehen ist, lieferten drei von vier Implementierungen mit *Gaussian Mutation* bessere durchschnittliche Endergebnisse als deren Gegenstücke. Relativ gesehen finden die beiden Mutationen in etwa gleich schnell eine gute Lösung, wie aus Tabelle 4 herauszulesen ist. Dabei bedeutet ein niedriger Wert, dass schneller ein Ergebnis gefunden wurde, das nahe bei der gefundenen Bestlösung liegt. Gerade die Kombination aus *TS - UC - UM* sticht hier besonders mit einem schnell gelieferten Ergebnis heraus.

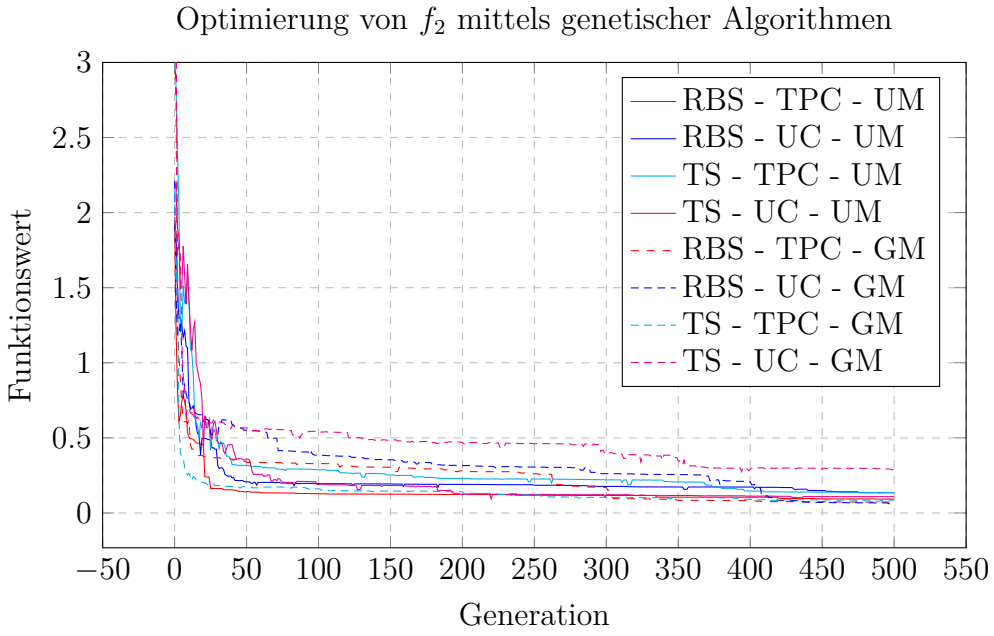


Abbildung 9: Vergleich bei der Optimierung von f_2 nach jeweils 20 Durchläufen.

Tabelle 4: Durchschnittliche $r_{0.05}$ Ergebnisse nach jeweils 20 Durchläufen (gerundet).

Implementierung	$r_{0.05}$
RBS - TPC - UM	0.839
RBS - UC - UM	0.918
TS - TPC - UM	0.817
TS - UC - UM	0.438
RBS - TPC - GM	0.821
RBS - UC - GM	0.884
TS - TPC - GM	0.801
TS - UC - GM	0.723

Tabelle 5: Durchschnittliche Ergebnisse nach jeweils 20 Durchläufen für f_2 (gerundet).

Implementierung	Endergebnis	Varianz σ^2	Standard-abweichung σ	Median
RBS - TPC - UM	0.09221	0.0161112	0.1269	0.04655
RBS - UC - UM	0.13500	0.0228960	0.1513	0.07649
TS - TPC - UM	0.13586	0.0257752	0.1605	0.06366
TS - UC - UM	0.10863	0.0156732	0.1251	0.05950
RBS - TPC - GM	0.07231	0.0054093	0.0735	0.04073
RBS - UC - GM	0.06707	0.0099267	0.0996	0.02572
TS - TPC - GM	0.08537	0.0064474	0.0802	0.07626
TS - UC - GM	0.29204	0.4303466	0.6560	0.13084

Insgesamt liegen aber alle Ergebnisse weiter von der Optimallösung entfernt als noch bei f_1. Dies hängt zum einen mit der gestiegenen Komplexität der zu optimierenden Funktion zusammen, als auch mit der Anzahl der Variablen einer Lösung. Da die Gene der Individuen stets float kodiert sind, besitzen bei diesem Optimierungsproblem die Chromosome nur zwei Gene. Dies bedeutet, dass die Funktion der Rekombination bei zwei Chromosomen hier nicht voll ausgeschöpft werden kann. So kann beispielsweise das *Two Point Crossover* nach der Bestimmung zweier Crossover Punkte keine ganzen Sequenzen übernehmen, da sowieso nur zwei Gene zur Verfügung stehen. Aufgrund der kurzen Länge des Chromosoms konnte also das Potential der Übernahme längerer Gensequenzen nicht genutzt werden, und so konnte keine Tendenz gegenüber der Gegenstücke mit *Uniform Crossover* betrachtet werden. Beide Crossover-Verfahren haben bei jeweils zwei von vier Tests das bessere durchschnittliche Endergebniss geliefert, *Two Point Crossover* hatte aber immerhin bei drei von vier Tests, wie in Tabelle 5 zu sehen ist, eine geringere Varianz.

Es war gut zu sehen, dass obwohl das Optimierungsproblem f_2 nicht optimal mit den verwendeten genetischen Algorithmen gelöst werden kann, sich trotzdem gewisse Tendenzen beobachten ließen. So hatten alle Implementierungen ähnliche Speicherauslastung und Ausführungszeit, doch ähnlich wie bei f_1 lieferten Implementierungen mit *Gaussian Mutation* am Ende die besten Ergebnisse mit geringerer Varianz. Die Kombination *RBS - UC - GM* hat bei diesem Test die besten Ergebnisse geliefert, doch wie bereits erläutert, hatte die Implementierung des Crossovers keine große Auswirkung, und so ist die Kombination *RBS - TPC - GM*, der Sieger des vorherigen Tests, auf dem zweiten Platz gelandet.

Nachdem nun verschiedene genetische Algorithmen auf die Optimierung der Funktionen f_1 und f_2 untersucht wurden, soll nun noch die zweite vorgestellte Art

Tabelle 6: Die verschiedenen Selektionen, Rekombinationen und Mutationen für das Problem des Handlungsreisenden.

Selektion	Rank based Selection	RBS
	Tournament Selection	TS
Crossover	Order Crossover	OC
	Partially Mapped Crossover	PMC
Mutation	Swapping Mutation	SM
	Insertion Mutation	IM

der Optimierungsprobleme behandelt werden. Da es sich um ein kombinatorisches Optimierungsproblem handelt, können die bereits untersuchten Crossover- und Mutationsverfahren hier nicht verwendet werden. Bei kombinatorischen Problemen darf sich lediglich die Reihenfolge der Gene eines Chromosoms verändern, nicht aber deren Werte. So kann bei den bisher benutzten Crossover- und Mutationsverfahren nicht garantiert werden, dass ein Wert verloren geht oder ein unerwünschter hinzukommt. Deswegen werden für das *Traveling Salesman Problem* die in Tabelle 6 gegebenen Kombinationen verwendet und untersucht.

Wie auch bei vorangegangenen Tests inspirierte hier vorhandene Literatur die Parametereinstellungen. So wurde eine Crossoverwahrscheinlichkeit von $p_c = 0.8$ und eine Mutationswahrscheinlichkeit von $p_m = 0.3$ gewählt [Has+19]. Aufgrund der erneut gestiegenen Komplexität des Optimierungsproblems wurde außerdem die Populationsgröße auf $n = 200$, sowie die maximale Generationenzahl auf $g = 2000$ erhöht. Zusätzlich zum Vergleich bei der Optimierung von *eil101.tsp* in Abbildung 10, bei der zu sehen ist, wie sich die absolute Distanz and die Optimallösung langsam angleicht, ist in Abbildung 11 außerdem eine der entstandenen Lösungen visuell dargestellt. Hier können also auch die Unterschiede zur Optimallösung bildlich veranschaulicht werden, eben durch abweichende, und damit ungünstigere, Routen. Wie in Abbildung 10 zu sehen ist, leichter aber natürlich in Tabelle 7, liefern vor allem Implementierungen mit *Insertion Mutation* die besten Endergebnisse. In keinem Test konnte das Gegenstück mit *Swapping Mutation* die besseren Endergebnisse erzielen. Gerade über viele Generationen hinweg erscheint dies auch als schlüssig. Während bei der *Swapping Mutation* jeweils zwei Städte/Gene bei gegebenem Individuum getauscht werden, verändert sich bei der *Insertion Mutation* die Position lediglich eines Gens. Dies führt dazu, dass besser kleine Schritte zur Optimierung unternommen werden können. Diese Beobachtung ist auch bei der Betrachtung der $r_{0.05}$ Werte festzustellen. Implementierungen mit der *Swapping Mutation* kommen schneller in die Nähe der Endlösung, aber wie in Tabelle 7 eben zu sehen ist, ist diese meist nicht sehr

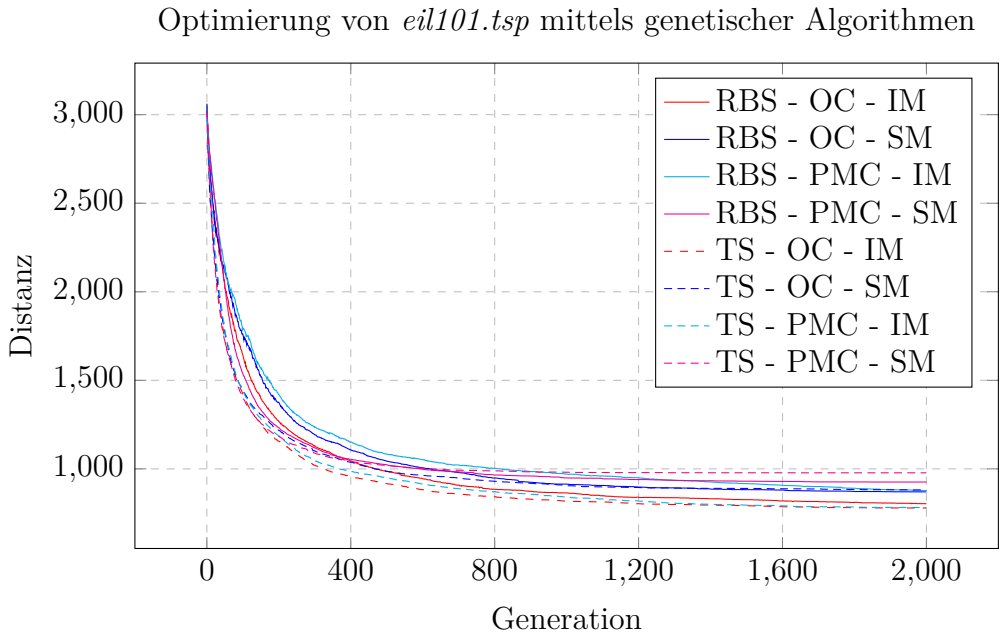


Abbildung 10: Vergleich bei der Optimierung von *eil101.tsp*.

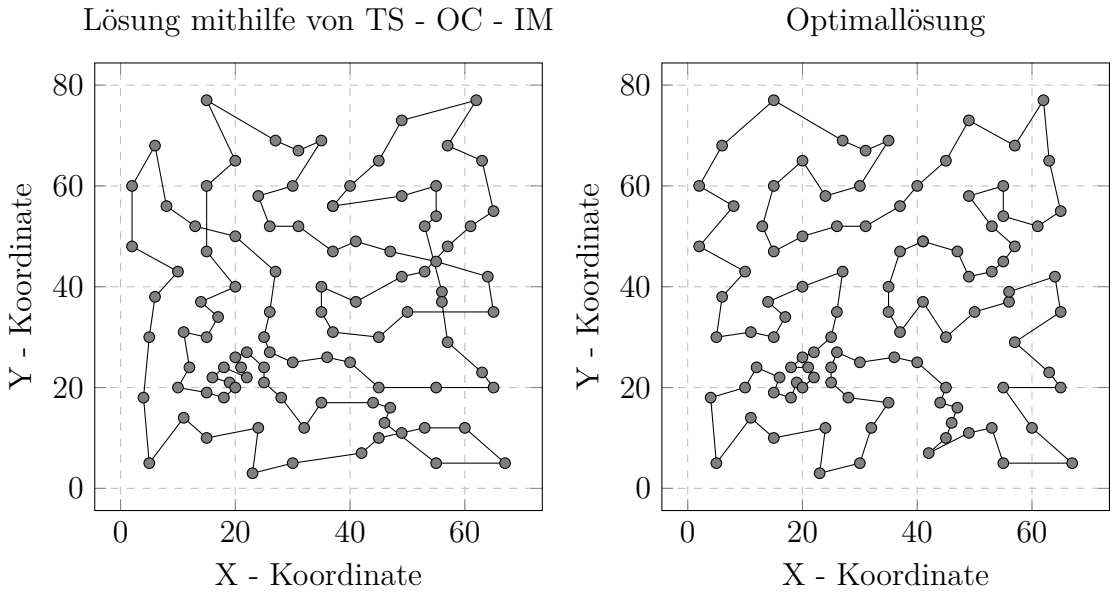


Abbildung 11: Vergleich einer gefundenen Lösung mit der Optimallösung für *eil101.tsp*.

Tabelle 7: Durchschnittliche Ergebnisse nach jeweils 10 Durchläufen für das *TSP* (gerundet).

Implementierung	Endergebnis	Varianz σ^2	Standard-abweichung σ	Median
RBS - OC - IM	803.52	1543.40	39.28	814.82
RBS - OC - SM	870.30	3578.20	59.81	870.97
RBS - PMC - IM	876.56	2751.16	52.45	876.13
RBS - PMC - SM	925.75	2060.80	45.39	925.23
TS - OC - IM	779.08	2887.45	53.73	766.37
TS - OC - SM	879.14	1230.41	35.07	881.37
TS - PMC - IM	781.99	1307.73	36.16	775.22
TS - PMC - SM	977.73	2806.39	52.97	983.91

genau. Zumindest in diesem Beispiel sollte die *Insertion Mutation* also vorgezogen werden. Genauso eindeutig ist auch der Vergleich zwischen *Order Crossover* und *Partially Mapped Crossover*, bei dem Letzteres in keinem einzigen Test das bessere durchschnittliche Endergebnis erzeugen konnte. Im Kontrast dazu ist es schwierig die Performance von *Rank based Selection* und *Tournament Selection* bei diesem Test zu vergleichen. Beide haben in jeweils zwei von vier Tests besser abgeschnitten, den größten Unterschied gab es bei der Kombination aus *TS - PMC - IM* mit einer durchschnittlichen Distanz von ≈ 782 im Vergleich zu einer Distanz von ≈ 877 bei der Implementierung mit *RBS - PMC - IM*. Es ist also, ähnlich wie bei der Optimierung von f_2, der Sieger nicht ganz so eindeutig feststellbar. Da allerdings die *Tournament Selection* die zwei besten durchschnittlichen Endergebnisse, sowie die zwei Ergebnisse mit der geringsten Varianz lieferte, kann diese durchaus als besser eingestuft werden. Insgesamt lieferte die Kombination aus *TS - OC - IM* bei diesem Test die besten Ergebnisse, der Unterschied bei Ausführungszeit und Speicherauslastung war vernachlässigbar.

6.3 Analyse der Evolutionsstrategien

In diesem Kapitel sollen nun die Evolutionsstrategien analysiert werden. Auch wenn sich die Funktionsweise der genetischen Algorithmen und der Evolutionsstrategien teils groß unterscheidet, so wurde dennoch darauf geachtet, möglichst viele Ähnlichkeiten beizubehalten. So werden die zwei Selektionsmethoden durch die zwei in Sektion 4 vorgestellten Evolutionsstrategien (μ, λ)-ES und $(\mu + \lambda)$-ES ersetzt. Das *Two Point Crossover* und das *Uniform Crossover* kommen angepasst auch bei den Evolutionsstrategien zum Einsatz. Als Mutation dienen bei dem kombinatorischen

Problem wieder die *Insertion Mutation* und die *Swapping Mutation*, auch diese natürlich angepasst. Für die Optimierung der Funktion wird hingegen immer die ES-eigene Mutation, wie in Sektion 4 vorgestellt, verwendet.

Tabelle 8: Die für die Evolutionsstrategien verwendeten Selektionen, Rekombinationen und Mutationen.

Selektion		(μ, λ) $(\mu + \lambda)$	K P
Crossover		Two Point Crossover Uniform Crossover	TPC UC
Mutation	für TSP	Uniform Mutation Gaussian Mutation	UM GM
	für f_1 & f_2	ES-Mutation	

Damit ergeben sich die in Tabelle 8 vorgestellten Kombinationen. Für alle Durchläufe wurden dabei $\mu = 15$, $\lambda = 105$, $\tau_1 = 0.1$ und $\tau_2 = 0.2$ gewählt, so wie in vorgegangenen Kapiteln beschrieben. Als Mindeststepsize für die Optimierung der Funktionen wurde $s_{min} = 0.000001$ festgelegt, um möglichst genaue Endlösungen zu erhalten. Darüberhinaus wurde die Generationenzahl auf 28 verringert. Grund dafür war, dass dadurch der Vergleich mit den genetischen Algorithmen vereinfacht werden soll. Denn bis zu dieser Generation gab es in etwa gleich viele Evaluationen wie bei den genetischen Algorithmen bei Generation 100.

In Abbildung 12 ist nun die Optimierung von f_1 mittels der Evolutionsstrategien abgebildet. Zu sehen ist, dass sich die beiden Arten der Selektion, einmal gestrichelt und einmal mit durchgezogener Linie dargestellt, sich im Schnitt in etwa gleich schnell an die Optimallösung angleichen. Wie in Tabelle 9 zu sehen, sind die durchschnittlichen Endergebnisse mit $(\mu + \lambda)$-Selektion allerdings deutlich genauer als deren Gegenstück mit (μ, λ)-Selektion. Dies ist der Tatsache geschuldet, dass im Gegensatz zur (μ, λ)-Selektion keine Chromosomen, mit eventuell sehr guter Fitness, aussortiert werden müssen. Somit müssen keine Rückschläge ertragen werden und der Graph der Optimierung geht hier nur nach unten. Hier muss jedoch abgewogen werden, ob es sich lohnt, das Risiko, an einem lokalen Optimum hängen zu bleiben, auf sich zu nehmen, oder doch die Komma-Selektion zu verwenden. In diesem Beispiel gibt es allerdings keine lokalen Optima, wodurch die Plus-Selektion deutlich zu präferieren ist. Insgesamt konnte sie allerdings nicht durch die erwartete, schnellere Annäherung am Anfang überzeugen, sondern durch ein deutlich besseres Endergebnis. Die Implementierungen mit *Two Point Crossover* weisen genau wie bei der Optimierung von f_1 mittels genetischer Algorithmen ein nicht nur besseres Endergebnis, sondern auch eine

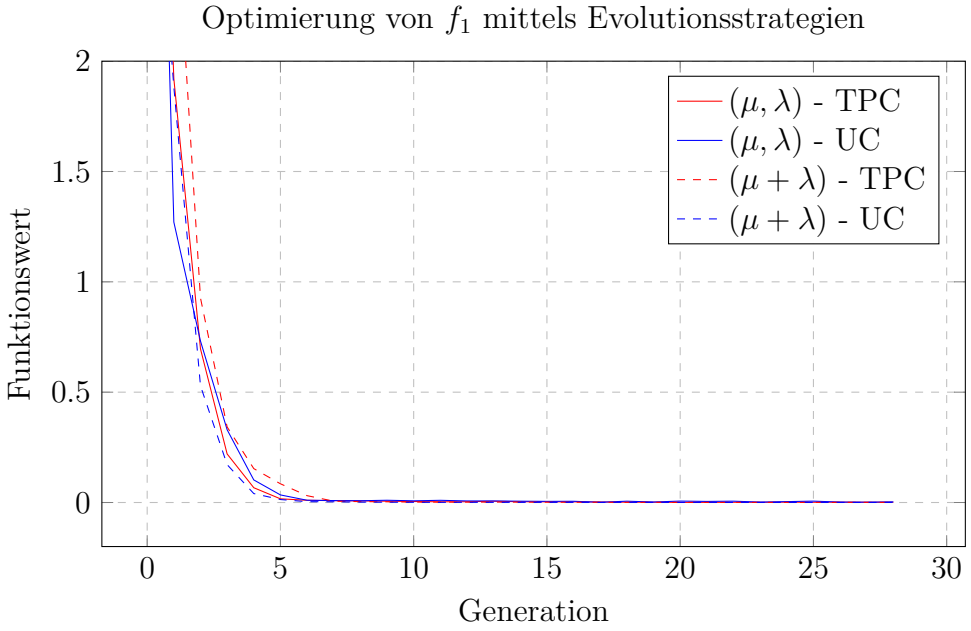


Abbildung 12: Vergleich bei der Optimierung von f_1 nach jeweils 10 Durchläufen.

geringere Varianz als deren Gegenstücke mit *Uniform Crossover* auf. An dieser Stelle soll auch noch auf die relativ geringen Mediane hingewiesen werden. Es scheinen viele Durchläufe ein sehr gutes Ergebnis geliefert zu haben, während vereinzelte Ausreißer den Durchschnitt nach oben treiben. In Abbildung 13 sind daher exemplarisch für die Kombination *K - TPC* die einzelnen Testläufe dargestellt. Bei der (μ, λ)-Selektion ist es definitionsbedingt möglich, dass sich die Werte auch wieder verschlechtern können und somit eventuell am Ende ein schlechteres Ergebnis zur Verfügung steht. Dies könnte den geringen Median, aber vergleichweise hohen Durchschnitt erklären. Bei der Anzahl der Evaluationen gibt es selbstverständlich keine Unterschiede. Da im Gegensatz zu den genetischen Algorithmen das Crossover und die Mutation jedes Mal durchgeführt werden, basiert hier die Anzahl der Evaluationen nicht auf Zufall. So galt für alle Testdurchläufe für f_1 eine Anzahl an Evaluationen von 5924.

Für f_2 wurde nun wieder die Generationenzahl angeglichen, um auf eine ähnliche Anzahl an Evaluationen zu kommen und so später einen besseren Vergleich zu gewährleisten. Die Anzahl an Generationen wurde dabei auf 138 festgesetzt, womit es zu 29134 Evaluationen für alle Tests kommt. Bei der Optimierung von f_2, abgebildet in Abbildung 14, kommt nun der Unterschied zwischen den Selektionsarten deutlicher zum Vorschein. Implementierungen mit der (μ, λ)-Selektion scheinen immer über denen mit $(\mu + \lambda)$-Selektion zu schweben. Während sich ganz am Anfang beide auf Augenhöhe verbessern, wird doch relativ schnell klar, dass die ständigen Ausreißer

Tabelle 9: Durchschnittliche Ergebnisse nach jeweils 10 Durchläufen für f_1 (gerundet).

Implementierung	Endergebnis	Varianz σ^2	Standard-abweichung σ	Median
K - TPC	0.0011671	0.000005332	0.0023091	0.0000107
K - UC	0.0029277	0.000079876	0.0089373	0.0000183
P - TPC	0.0006005	0.000001160	0.0010773	0.0001361
P - UC	0.0003859	0.000000920	0.0009594	0.0000456

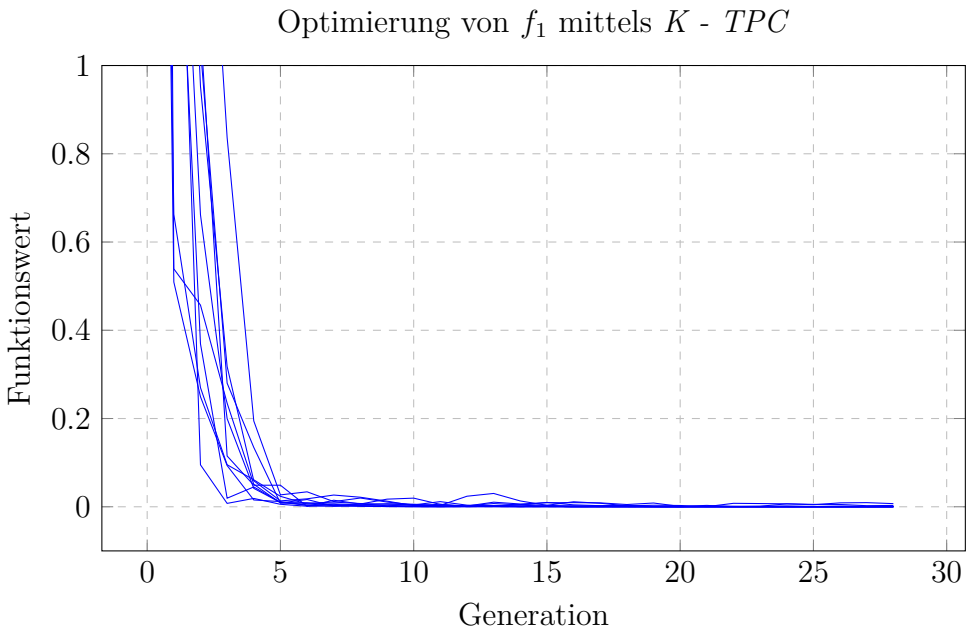


Abbildung 13: Die einzelnen Testdurchläufe für f_1 mittels K - TPC im Vergleich.

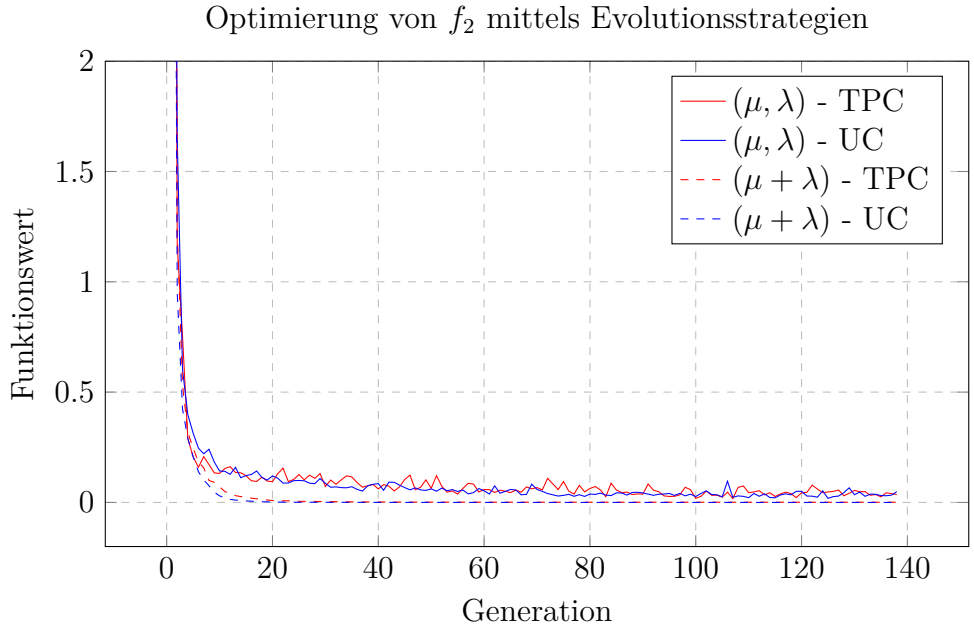


Abbildung 14: Vergleich bei der Optimierung für f_2 nach jeweils 20 Durchläufen.

nach oben die (μ, λ)-Selektion daran hindert, näher an die Optimallösung vorzudringen. Im Gegensatz dazu ist bei der $(\mu + \lambda)$-Selektion schön zu sehen, dass die Kurve nur nach unten geht. Hier ist eine Verschlechterung der Lösung auch nicht möglich, da Lösungen der Elterngeneration auch in die Nachfolgegeneration übernommen werden können. So ist also bei diesem Problem in dieser Implementierung die $(\mu + \lambda)$-Selektion als die Bessere anzusehen. Dies bestätigt natürlich nicht nur der gezeigte Optimierungsverlauf, sondern auch die Endergebnisse und Varianzen, die in Tabelle 10 gegeben sind. Der geringere $r_{0.05}$-Wert der Komma-Selektion in Tabelle 11 zeigt dabei, dass schneller eine Lösung in der Nähe der Bestlösung gefunden wurde. Bei einem gleich guten Ergebnis wie bei der $(\mu + \lambda)$-Selektion wäre dies natürlich gut. Hier allerdings ist das Ergebnis deutlich schlechter und zeigt deswegen auf, dass für einen deutlich größeren Zeitraum kaum Verbesserungen gefunden werden konnten und der Algorithmus somit in einem lokalen Optimum hängt.

Beim Vergleich zwischen den zwei Rekombinationsoperatoren bestätigt sich das Ergebnis aus vorangegangenem Test. Genau wie bei diesem konnte einmal die Implementierung mit *Two Point Crossover* und einmal die Implementierung mit *Uniform Crossover* das bessere durchschnittliche Endergebnis erzielen. Insgesamt schnitt aber die Kombination aus $(\mu + \lambda)$-Selektion und *Uniform Crossover* bei diesem Test im Durchschnitt am besten ab.

Tabelle 10: Durchschnittliche Ergebnisse für f_2 nach jeweils 20 Durchläufen (gerundet).

Implementierung	Endergebnis	Varianz σ^2	Standard-abweichung σ	Median
K - TPC	0.03547720	0.00440862893	0.066397	0.0056925
K - UC	0.04881560	0.01337853620	0.115665	0.0009203
P - TPC	0.00088841	0.00001267751	0.003560	0.0000383
P - UC	0.00001638	0.00000000056	0.000023	0.0000082

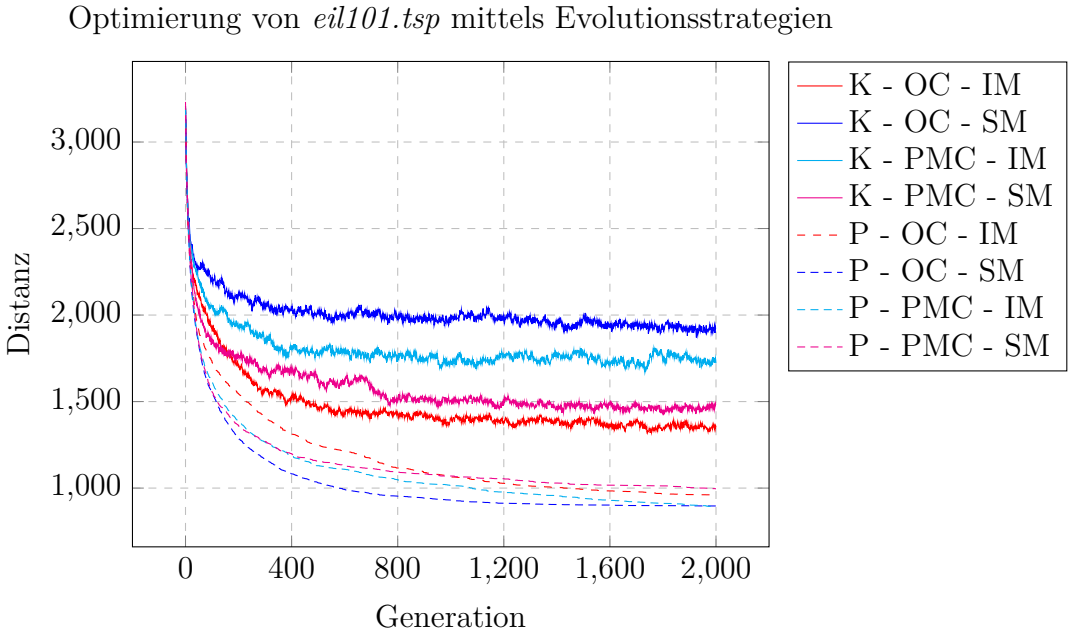


Abbildung 15: Vergleich bei der Optimierung von *eil101.tsp*.

Tabelle 11: Durchschnittliche $r_{0.05}$ Ergebnisse nach jeweils 20 Durchläufen für f_2 (gerundet).

Implementierung	$r_{0.05}$
K - TPC	0.593
K - UC	0.364
P - TPC	0.821
P - UC	0.807

Für den letzten Test der Evolutionsstrategien soll nun wieder das bereits erläuterte *TSP* dienen. Hierbei musste die Generationenzahl nicht verändert werden, sondern konnte wieder auf 2000 limitert werden. Dabei ergab sich dann für jeden Durchlauf eine feste Anzahl von 420015 Evaluationen und damit nur eine kleine Abweichung im Vergleich zu den genetischen Algorithmen bei gleicher Generationenzahl. Die Mindeststepsize hingegen wurde auf $s_{min} = 0.5$ erhöht, so dass nach dem Runden eine Ganzzahl, konkret $s = 1$, gilt. Die für das Optimierungsproblem *TSP* gewählten Mutationsoperatoren *Insertion Mutation* und *Swapping Mutation* werden entweder einfach oder mehrfach ausgeführt, jedoch nie nur einen Bruchteil davon, weshalb hier später nur Ganzzahlen verwendet werden können. Da dies wieder ein kombinatorisches Optimierungsproblem ist, wird wieder das *Order Crossover* und das *Partially Mapped Crossover* verwendet. Beim Betrachten der Optimierung mittels der verschiedenen Implementierungen in Abbildung 15 fällt sofort auf, dass eine Art der Implementierung deutlich bessere Ergebnisse liefert, und zwar die mithilfe der $(\mu + \lambda)$-Selektion, hier gestrichelt dargestellt. Dies ist sicherlich der bisher deutlichste Unterschied zwischen zwei Implementierungen. So schweben die Optimierungskurven mit der (μ, λ)-Selektion förmlich über den anderen und kommen diesen auch über den gesamten Verlauf nicht näher. Dass die Kurven mit der Plus-Selektion einen deutlich glatteren Verlauf ohne Ausbrüche nach oben besitzen, wurde bereits ausreichend erläutert. Erstaunlich ist hier aber, dass die Optimierung der Komma-Selektion sich im Vergleich so langsam verbessert. Dies ist aber eine Eigenschaft des *TSPs*, denn bei kombinatorischen Optimierungsproblemen reichen schon kleine Veränderungen aus, um große Sprünge im Endergebnis zu erhalten. So wurde hier anscheinend ein natürliches Endergebnis erzielt, denn ab einer gewissen Fitness ist die zufällige Mutation in beide Richtungen, das heißt also eine Verbesserung oder Verschlechterung der Fitness, gleich wahrscheinlich, und so schwankt der Wert stetig um diesen Punkt. Auch bei Optimierungen mit der $(\mu + \lambda)$-Selektion wird es immer unwahrscheinlicher, dass die Mutation eine Verbesserung hervorbringt, aber da hier keine Verschlechterung möglich ist, reicht dies dennoch aus, um das Ergebnis langsam zu verbessern. Beim

Vergleich der zwei Rekombinationsoperatoren *Order Crossver* und *Partially Mapped Crossover* konnte bei diesem Test keine eindeutige Tendenz festgestellt werden. Beide lieferten jeweils in der Hälfte aller durchgeführten Tests das bessere Endergebnis. Da bei den Evolutionsstrategien das Crossover aber auch im Hintergrund steht, soll hier nicht weiter darauf eingegangen werden. Auch zwischen *Swapping Mutation* und *Insertion Mutation* konnte beim Endergebnis keine eindeutige Tendenz betrachtet werden. Jedoch besaßen die Implementierungen mit *Swapping Mutation* meist eine deutlich geringere Standardabweichung, der Median hingegen war bei den Implementierungen mit $(\mu + \lambda)$-Selektion und *Insertion Mutation* am geringsten, was bedeutet, dass diese eigentlich die besten Werte lieferten, jedoch einige wenige Tests den Durchschnitt nach oben gezogen haben. Insgesamt lieferte im Durchschnitt die Kombination aus *P - PMC - IM* die besten Endergebnisse mit dem gleichzeitig geringsten Median.

Tabelle 12: Durchschnittliche Ergebnisse für das TSP nach jeweils 10 Durchläufen (gerundet).

Implementierung	Endergebnis	Varianz σ^2	Standard-abweichung σ	Median
K - OC - IM	1351.53	485523	696.79	1020.95
K - OC - SM	1920.92	719894	848.46	2516.95
K - PMC - IM	1721.59	646596	804.11	1941.795
K - PMC - SM	1479.54	607378	779.34	1054.00
P - OC - IM	960.32	81528	285.53	805.60
P - OC - SM	896.71	1751	41.84	893.62
P - PMC - IM	894.22	49360	222.17	804.55
P - PMC - SM	996.11	28640	169.23	940.78

6.4 Gegenüberstellung der Metaheuristiken

Nachdem nun die einzelnen Evolutionsstrategien weiter analysiert wurden, sollen diese noch gegenübergestellt und verglichen werden. Auch hier werden wieder die oben benutzten Optimierungsprobleme verwendet. Als Grundlage für den Vergleich werden dabei immer nur die Implementierungen verwendet, die bei den bisherigen Tests am besten abgeschnitten haben. Auf die anderen Implementierungen soll im Weiteren nicht eingegangen werden, da bei einer realen Optimierung mittels evolutionärer Algorithmen auch davon ausgegangen wird, dass hier die beste Implementierung verwendet wird. Um den Vergleich anschaulicher machen zu können wird die x-Achse

Tabelle 13: Durchschnittliche Ergebnisse von jeweils 10 Durchläufen für f_1 der besten Implementierungen (gerundet).

Algorithmus	Endergebnis	Standard-abweichung σ	Median
Genetischer Algorithmus	0.0000091696	0.0000069	0.0000099
Evolutionsstrategie	0.0003859299	0.0009594	0.0000456
Hill climbing	0.0008201471	0.0003850	0.0006958

der Graphen angeglichen. Hierfür werden die einzelnen Werte der Evolutionsstrategien um den Wert

$$\vartheta = \frac{Generationen_{GA}}{Generationen_{ES}}$$

hochskaliert. Die tatsächliche Anzahl an Generationen und Evaluationen verändert sich hierbei nicht, die Optimierungskurven lassen sich jedoch besser miteinander vergleichen. Desweiteren kommen zu den Vergleichen noch die Ergebnisse eines einfachen Hill climbing Algorithmus dazu. Auf diesen soll nicht weiter eingangen werden, er dient lediglich als Vergleich zu einem simplen Algorithmus und soll damit eine bessere Einschätzung der Ergebnisse der evolutionären Algorithmen gewährleisten. Es wurde hierbei auf faire Bedingungen geachtet, so ist auch hier die Anzahl an Evaluationen beim Hill climbing Algorithmus der von vorhandenen Tests angepasst.

Beim Optimieren der Funktion f_1 verbleiben also die Implementierung mit *RBS - TPC - GM* als Vertretung der genetischen Algorithmen und die Kombination aus *P - UC* der Evolutionsstrategien. In Abbildung 16 dargestellt sind die Optimierungskurven dieser zwei Algorithmen. Zur besseren Einschätzung der Performance dieser zwei evolutionären Algorithmen ist hier erstmals auch die Kurve eines einfachen Hill climbing Algorithmus dargstellt. Auf den ersten Blick sehen diese Kurven sehr ähnlich aus, doch wie mit einem Blick auf Tabelle 13 zu sehen ist, sind die komplizierten Algorithmen GA und ES doch um ein vielfaches genauer. Für eine grobe Näherung ist auch hier das Hill climbing gut geeignet. Es handelt sich jedoch auch um ein sehr einfaches Optimierungsproblem, bei dem die Algorithmen nicht in einem lokalen Optimum feststecken können.

Der eigentliche Vorteil der Evolutionsstrategien, nämlich das Anpassen der Strategieparameter, konnte bei diesem Test nicht überzeugen. So gelang der getestete genetische Algorithmus schneller zu einer guten Lösung und lieferte gleichzeitig am Ende das genaueste Endergebnis. Eventuell lag dies aber auch an dem gegebenen

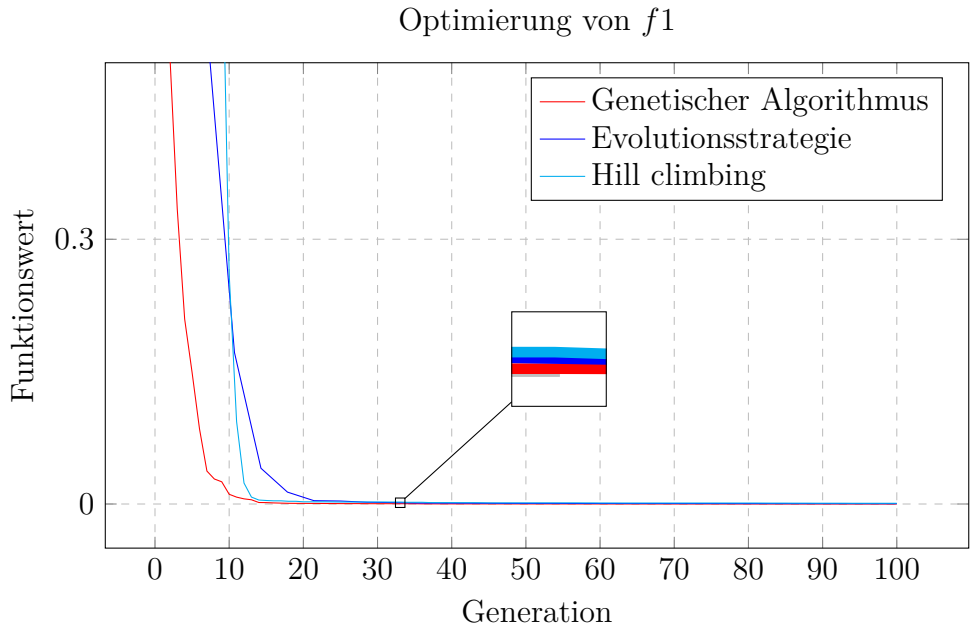


Abbildung 16: Der beste gefundene GA, die beste gefundene ES und Hill climbing im Vergleich bei der Optimierung von f_1.

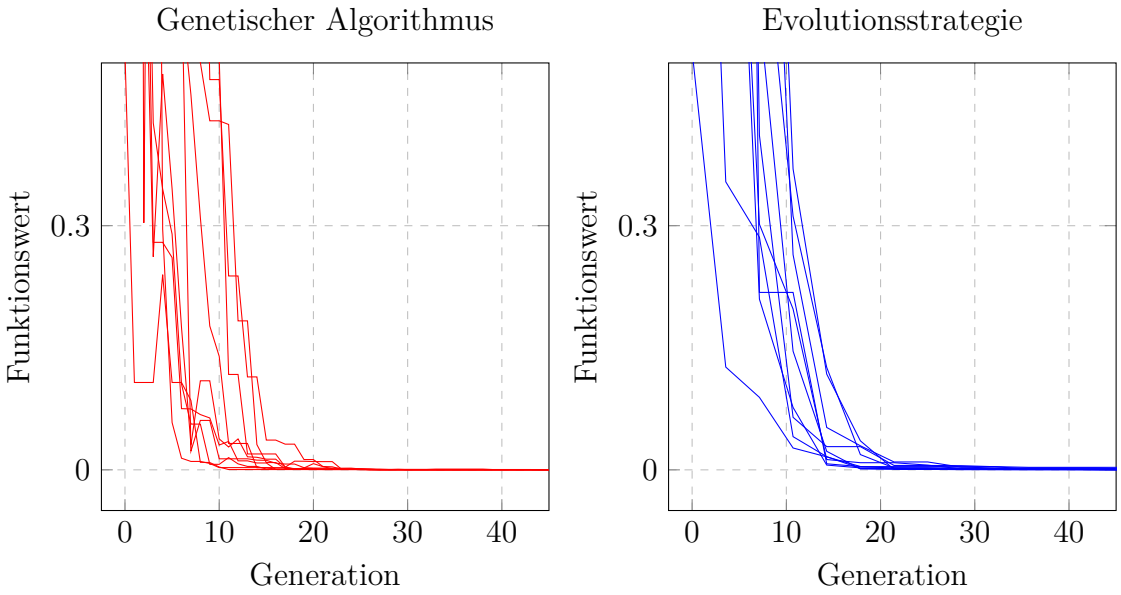


Abbildung 17: Vergleich der ersten Generationen der einzelnen Testdurchläufe bei Funktion f_1.

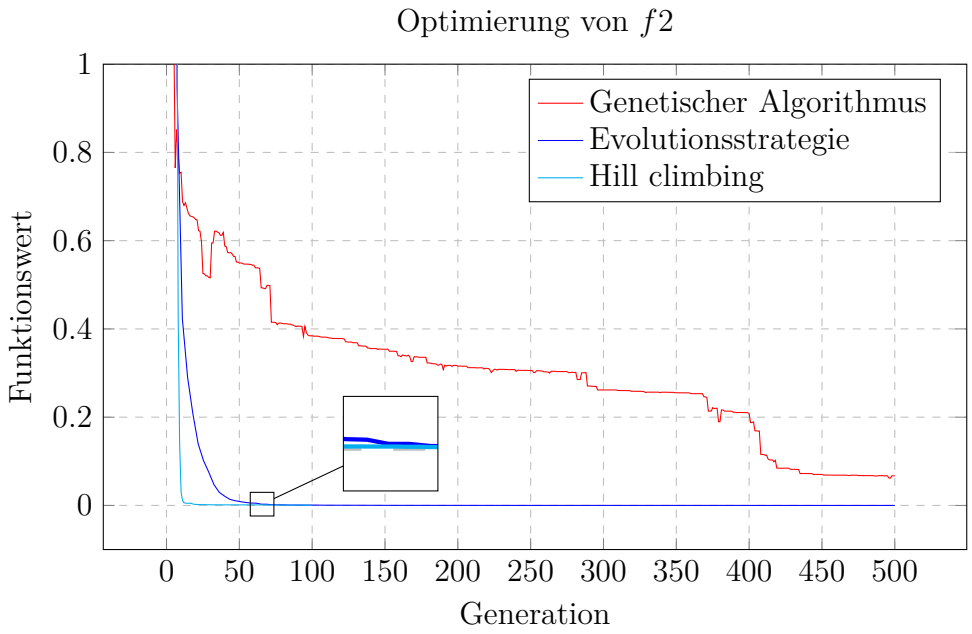


Abbildung 18: Der beste gefundene GA, die beste gefundene ES und Hill climbing im Vergleich bei der Optimierung von f_2.

Optimierungsproblem, bei dem die Evolutionsstrategie eventuell diesen Vorteil nicht ganz ausspielen konnte, weswegen dies nun im zweiten Vergleich überprüft werden soll.

Auch beim Vergleich der Optimierung von f_2 soll nun wieder das Hill climbing als Referenz herangezogen werden und gerade bei diesem Test überrascht dessen Performance am meisten. In der Abbildung 18 ist schön zu sehen, wie deutlich besser als die evolutionären Algorithmen diese einfache Optimierung am Anfang funktioniert. Gerade der Unterschied zu dem genetischen Algorithmus ist hier gewaltig und selbst bis zum Ende der 500 Generationen schafft dieser nicht an das Hill climbing anzuschließen. Der Grund hierzu wurde vorhin schon einmal kurz angerissen. Der Hauptgrund hierfür dürfte sein, dass bei den genetischen Algorithmen die Rekombination der Hauptoperator ist, bei diesem konkreten Problem jedoch die Rekombination, bedingt durch nur zwei Parameter, kaum Einfluss auf die Verbesserung des Endergebnisses haben kann. Das selbe Problem haben natürlich auch die Evolutionsstrategien, jedoch ist hier der Hauptoperator die Mutation, weshalb dies keine große Auswirkung hat. Gerade bei so einem Problem kann also durchaus auch ein einfaches heuristisches Optimierungsverfahren gut abschneiden. Hier soll noch kurz erwähnt werden, dass dies nur die verwendete Implementierung betrifft. Eventuelle Implementierungen der genetischen Algorithmen mit einer binären Kodierung oder anderen Rekombinati-

Tabelle 14: Durchschnittliche Ergebnisse von jeweils 20 Durchläufen für f_2 der besten Implementierungen (gerundet).

Algorithmus	Endergebnis	Standard-abweichung σ	Median
Genetischer Algorithmus	0.0670722	0.099633	0.02572681
Evolutionsstrategie	0.0000163	0.000023	0.00000828
Hill climbing	0.0005398	0.000289	0.00045838

onsmethoden könnten hier durchaus auch besser abschneiden. Mit einem Blick auf die Tabelle 14 ist zu sehen, dass die verwendete Evolutionsstrategie am Ende das deutlich beste durchschnittliche Endergebnis lieferte. So konnte das Anfangs gut abschneidende Hill climbing den gewonnenen Vorsprung nicht aufrecht erhalten. Die deutlich schnellere Annäherung an die Optimallösung der Evolutionsstrategie und das gleichzeitig bessere Endergebnis als der genetische Algorithmus lassen darauf schließen, dass sich hier die adaptiv anpassenden Strategieparamter auszahlen, im Gegensatz zur Optimierung von f_1. Grund könnte hierfür sein, dass die Optimierung von f_1 sehr kurz dauerte und so durch diesen Vorgang keine Zeit gespart werden konnte. Bei diesem Test ist jedoch zu sehen, dass der genetische Algorithmus relativ lange braucht, um sich der Optimallösung passabel zu nähern und so konnte hier das Anpassen der Stepsize der Evolutionsstrategie überzeugen.

Zu guter Letzt ist in Abbildung 19 nun noch der Vergleich beim *Traveling Salesman Problem* zu sehen. Hier ist nun sehr schön der Vorteil der evolutionären Algorithmen erkennbar. Während das Hill climbing anfangs wieder einmal sehr schnell eine gute Lösung findet, so verbessert sich diese Lösung auch bei längerem Ausführen des Algorithmus kaum. Es bleibt relativ schnell bei einem lokalen Optimum hängen. Die evolutionären Algorithmen haben dieses Problem nicht so früh. Dies liegt vor allem an der hohen Populationszahl, die dafür sorgt, dass viele verschiedene Lösungsansätze weiter untersucht werden. So ist sowohl bei dem genetischen Algorithmus wie auch der Evolutionsstrategie eine Verbesserung bis ganz zum Ende zu beobachten. Dieses Verfolgen der unterschiedlichen Lösungskandidaten wirkt sich aber natürlich vor allem am Anfang auf die Schnelligkeit der Suche aus, weswegen beide anfangs hinter dem einfachen Optimierungsverfahren liegen. Dennoch lohnt sich dieser Mehraufwand, da bei gleicher Anzahl an Evaluationen am Ende doch die besseren Ergebnisse erwartet werden können. In diesem Test hat dabei der genetische Algorithmus das beste durchschnittliche Endergebnis geliefert, in Tabelle 15 sind hierzu noch die genauen Endwerte abgebildet.

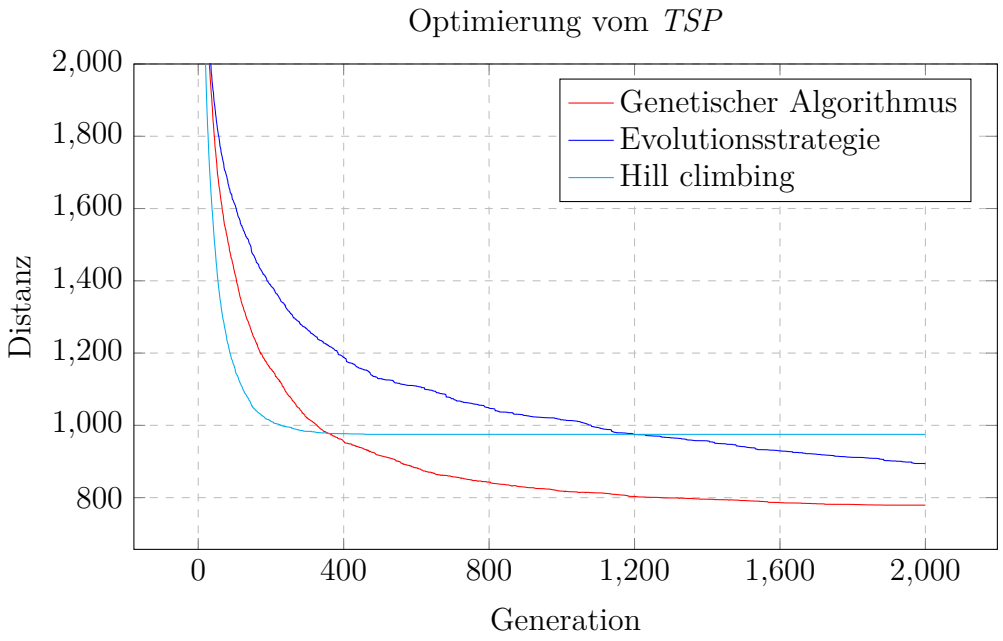


Abbildung 19: Der beste gefundene GA, die beste gefundene ES und Hill climbing im Vergleich bei der Optimierung vom *TSP*.

Sehr auffällig ist hier die besonders hohe Standardabweichung der Evolutionsstrategie. Dies liegt daran, dass die einzelnen Durchläufe dieser besonders vom Zufall abhängen. Zusätzlich zu allen kleinen zufälligen Entscheidungen, wie beispielsweise bei Rekombination und Mutation pro Gen, kommt hier noch eine Zufallsvariable $N(0, 1)$ hinzu, die über den ganzen Verlauf der Evolutionsstrategie gleich bleibt und somit deutlich wichtiger ist. Diese sorgt dafür, dass die einzelnen Durchläufe noch unterschiedlicher sind und so kommt es hier zu einer größeren Standardabweichung. Der gesamte durchschnittliche Verlauf der zwei evolutionären Algorithmen ist aber durchaus vergleichbar, wenngleich die Optimierung des genetischen Algorithmus besser ausfällt.

Tabelle 15: Durchschnittliche Ergebnisse von jeweils 10 Durchläufen für das *TSP* der besten Implementierungen (gerundet).

Algorithmus	Endergebnis	Standard-abweichung σ	Median
Genetischer Algorithmus	779.08	53.73	766.37
Evolutionsstrategie	894.22	222.17	804.55
Hill climbing	975.06	65.72	979.06

6.5 Ergebnisse der Vergleiche

Der verwendete genetische Algorithmus hat bei der Optimierung der Funktion f_1 als auch beim kombinatorischen Optimierungsproblem *TSP* das deutlich beste Ergebnis geliefert. So lieferte er bei Funktion f_1 ein etwa 42-fach besseres Ergebnis als das zweitplatzierte Resultat der Evolutionsstrategie, respektive 1,15-fach besser beim *TSP*. Damit konnte schön gezeigt werden, wie vielfältig genetische Algorithmen eingesetzt werden können und mit unterschiedlichen Aufgabenstellungen zurecht kommen. Die verwendete Evolutionsstrategie hingegen konnte vor allem beim Optimieren von Funktion f_2 überzeugen und lieferte hier das mit Abstand beste Ergebnis. So ist hier der genetische Algorithmus mit einem durchschnittlichen Endergebnis, das um mehrere Größenordnungen schlechter ausfiel, auf dem letzten Platz gelandet. Dennoch waren auch die Ergebnisse der anderen Tests befriedigend und die Evolutionsstrategien dadurch durchaus auch für eine Vielzahl an Anwendungen zu gebrauchen. Insgesamt konnten somit einige Erkenntnisse vorangengener Literatur bestätigt werden, namentlich die sehr gute Funktionsweise der Evolutionsstrategien bei reellen Problemen, zum Beispiel aus [Wie+08], und die beeindruckende Anpassungsfähigkeit bei gleichzeitig sehr guter Performance der genetischen Algorithmen bei beliebigen Problemen, beispielsweise aus [LH13] oder [NS10]. Die Tests stimmen außerdem mit dem sogenannten *No Free Lunch Theorem* überein. Dieses besagt, dass es keine universelle Metaheuristik gibt, die bei allen Problemen gut performt [Ige14]. Da keiner der verwendeten Algorithmen bei allen drei Problemen am besten abschnitt, entsteht zumindest bis hierhin auch kein Widerspruch.

7 Fazit und Ausblick

In dieser Arbeit wurde die Funktionsweise genetischer Algorithmen und Evolutionsstrategien vorgestellt. Daraufhin wurden bestimmte Implementierungen dieser auf zwei unterschiedliche Arten von Optimierungsproblemen angewendet und die

Ergebnisse miteinander verglichen. Zur besseren Einschätzung der Resultate wurde zusätzlich noch der *Bergsteigeralgorithmus* herangezogen, der bei jedem Test zumindest von einem der zwei evolutionären Algorithmen übertroffen wurde. Die Evolutionsstrategien lieferten bei der Optimierung komplexer, reeller Probleme herausragende Werte, während die genetischen Algorithmen die besten Ergebnisse bei gegebenem kombinatorischen Optimierungsproblem lieferte, aber auch akzeptable Werte bei der Optimierung einer Funktion lieferte. Unter anderem durch die große Anzahl an möglichen Implementierungen der evolutionären Algorithmen lassen sich nur schwer allgemeine Aussagen über die erwartete Performance dieser Algorithmen zu bestimmten Problemen treffen, weshalb hier noch weitere Nachforschungen nötig sind. In diesen Tests wurde dabei versucht, die Komplexität der Implementierung der verschiedenen Algorithmen möglichst vergleichbar zu halten, auch hier würden sich noch weitere Untersuchungen anbieten, da sich diese eventuell noch unterschiedlich gut optimieren lassen. Insgesamt konnten einige wichtige Erkenntnisse aus den Resultaten gewonnen und einige vorhandene Vermutungen bestätigt werden, auch wenn einige Ergebnisse noch weiter statistisch überprüft werden müssen und untersucht werden sollte, ob die Unterschiede denn signifikant sind. Selbstverständlich kann jedoch immer nur ein Teil dieses komplexen Themengebiets erfasst werden, weswegen vermutlich noch viele weitere Untersuchungen folgen werden.

8 Literatur

[AAT12] Otman Abdoun, Jaafar Abouchabaka und Chakir Tajani. „Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem“. In: *CoRR* abs/1203.3099 (2012). arXiv: 1203.3099. URL: http://arxiv.org/abs/1203.3099.

[Bar+95] R. Barr u. a. „Designing and Reporting on Computational Experiments with Heuristic Methods“. In: *Journal of Heuristics* (1995). DOI: https://doi.org/10.1007/BF02430363.

[BL04] Maik Buttelmann und Boris Lohmann. „Optimierung mit Genetischen Algorithmen und eine Anwendung zur Modellreduktion (Optimization with Genetic Algorithms and an Application for Model Reduction)“. In: *at - Automatisierungstechnik* 52.4 (1Apr. 2004), S. 151–163. DOI: https://doi.org/10.1524/auto.52.4.151.29416. URL: https://www.degruyter.com/view/journals/auto/52/4/article-p151.xml.

[BS93] Thomas Bäck und Hans-Paul Schwefel. *An Overview of Evolutionary Algorithms for Parameter Optimization*. 1993. DOI: https://doi.org/10.1162/evco.1993.1.1.1.

[BT96] T. Blickle und L. Thiele. „A Comparison of Selection Schemes Used in Evolutionary Algorithms“. In: *Evolutionary Computation* 4.4 (1996), S. 361–394. DOI: 10.1162/evco.1996.4.4.361.

[Dee11] Hadush Deep KusumMebrahtu. „New Variations of Order Crossover for Travelling Salesman Problem“. Español. In: *International Journal of Combinatorial Optimization Problems and Informatics* (2011). URL: https://www.redalyc.org/articulo.oa?id=265219618002.

[DM97] Dipankar Dasgupta und Zbigniew Michalewicz. „Evolutionary Algorithms — An Overview“. In: *Evolutionary Algorithms in Engineering Applications*. Hrsg. von Dipankar Dasgupta und Zbigniew Michalewicz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, S. 3–28. ISBN: 978-3-662-03423-1. DOI: 10.1007/978-3-662-03423-1_1. URL: https://doi.org/10.1007/978-3-662-03423-1_1.

[Fob06] Thomas Fober. „Experimentelle Analyse Evolutionärer Algorithmen auf dem CEC 2005 Testfunktionensatz“. In: (2006).

[GA07] Crina Grosan und Ajith Abraham. „Hybrid Evolutionary Algorithms: Methodologies, Architectures, and Reviews“. In: *Hybrid Evolutionary Algorithms*. Hrsg. von Ajith Abraham, Crina Grosan und Hisao Ishibuchi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, S. 1–17. ISBN: 978-

3-540-73297-6. DOI: 10.1007/978-3-540-73297-6_1. URL: https://doi.org/10.1007/978-3-540-73297-6_1.

[GKK13] I. Gerdes, F. Klawonn und R. Kruse. *Evolutionäre Algorithmen: Genetische Algorithmen — Strategien und Optimierungsverfahren — Beispielanwendungen*. Computational Intelligence. Vieweg+Teubner Verlag, 2013. ISBN: 9783322868398. URL: https://books.google.de/books?id=TBd6rWMLCqcC.

[Has+19] Ahmad Hassanat u. a. „Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach“. In: *Information* 10.12 (Dez. 2019), S. 390. ISSN: 2078-2489. DOI: 10.3390/info10120390. URL: http://dx.doi.org/10.3390/info10120390.

[HB91] Frank Hoffmeister und Thomas Bäck. „Genetic Algorithms and evolution strategies: Similarities and differences“. In: *Parallel Problem Solving from Nature*. Hrsg. von Hans-Paul Schwefel und Reinhard Männer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, S. 455–469. ISBN: 978-3-540-70652-6.

[Hem97] Thomas Hemmer. „Konzeption und Realisierung einer Evolutionsstrategie zum Abgleich eines anthropometrisch-kinematischen Menschmodells mit dreidimensionalen Sensordaten“. In: (1997).

[Hin95] R. Hinterding. „Gaussian mutation and self-adaption for numeric genetic algorithms“. In: *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*. Bd. 1. 1995, S. 384–. DOI: 10.1109/ICEC.1995.489178.

[Ige14] C. Igel. „No Free Lunch Theorems: Limitations and Perspectives of Metaheuristics“. In: *Theory and Principled Methods for the Design of Metaheuristics*. 2014.

[LH13] Seng Poh Lim und Habibollah Haron. „Performance comparison of Genetic Algorithm, Differential Evolution and Particle Swarm Optimization towards benchmark functions“. In: Dez. 2013, S. 41–46. ISBN: 978-1-4799-0285-9. DOI: 10.1109/ICOS.2013.6735045.

[Lin+10] Guangming Lin u. a. „Ranking Based Selection Genetic Algorithm for Capacity Flow Assignments“. In: *Computational Intelligence and Intelligent Systems*. Hrsg. von Zhihua Cai u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 97–107. ISBN: 978-3-642-16388-3.

[LYS17] Saneh Lata, Saneh Yadav und Asha Sohal. „Comparative Study of Different Selection Techniques in Genetic Algorithm“. In: *International Journal of Engineering Science* (Juli 2017).

[MPR18] Rafael Mart, Panos M. Pardalos und Mauricio G. C. Resende. *Handbook of Heuristics*. 1st. Springer Publishing Company, Incorporated, 2018. ISBN: 3319071254.

[Nis97] Volker Nissen. *Einführung in Evolutionäre Algorithmen*. 1997. DOI: 10.1007/978-3-322-93861-9.

[NS10] T. R. Gopalakrishnan Nair und Kavitha Sooda. *Comparison of Genetic Algorithm and Simulated Annealing Technique for Optimal Path Selection In Network Routing*. 2010. arXiv: 1001.3920 [cs.NE].

[PBE08] Harun Pirim, Engin Bayraktar und Burak Eksioglu. „Tabu Search: A Comparative Study“. In: Sep. 2008. ISBN: 978-3-902613-34-9. DOI: 10.5772/5637.

[Pyt] DEAP (Distributed Evolutionary Algorithms in Python). *Benchmarks*. Zuletzt aufgerufen: 29.12.2020. URL: http://deap.gel.ulaval.ca/doc/0.7/api/benchmarks.html.

[SG13a] Kenneth Sörensen und Fred Glover. „Metaheuristics“. In: Jan. 2013, S. 960–970. ISBN: 978-1-4419-1137-7. DOI: 10.1007/978-1-4419-1153-7_1167.

[SG13b] Kenneth Sörensen und Fred W. Glover. „Metaheuristics“. In: *Encyclopedia of Operations Research and Management Science*. Hrsg. von Saul I. Gass und Michael C. Fu. Boston, MA: Springer US, 2013, S. 960–970. ISBN: 978-1-4419-1153-7. DOI: 10.1007/978-1-4419-1153-7_1167. URL: https://doi.org/10.1007/978-1-4419-1153-7_1167.

[Soo+13] G. K. Soon u. a. „A comparison on the performance of crossover techniques in video game“. In: *2013 IEEE International Conference on Control System, Computing and Engineering*. 2013, S. 493–498. DOI: 10.1109/ICCSCE.2013.6720015.

[US15] Dr. Anantkumar Umbarkar und P. Sheth. „CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A REVIEW“. In: *ICTACT Journal on Soft Computing (Volume: 6 , Issue: 1)* 6 (Okt. 2015). DOI: 10.21917/ijsc.2015.0150.

[Wei15] K. Weicker. *Evolutionäre Algorithmen*. Springer Fachmedien Wiesbaden, 2015. ISBN: 9783658099589. URL: https://books.google.de/books?id=%5C_bjpCQAAQBAJ.

[Wie+08] Daan Wierstra u. a. „Natural Evolution Strategies“. In: Juni 2008, S. 3381–3387. DOI: 10.1109/CEC.2008.4631255.

9 Anhang

Der Vollständigkeit halber sind hier alle übrigen Werte, die nicht direkt für die Argumentation benötigt wurden, angegeben.

Sektion: 6.2

Tabelle 16: Durchschnittliche $r_{0.05}$ Ergebnisse nach jeweils 10 Durchläufen für f_1 (gerundet)

Implementierung	$r_{0.05}$
RBS - TPC - UM	0.882
RBS - UC - UM	0.873
TS - TPC - UM	0.941
TS - UC - UM	0.961
RBS - TPC - GM	0.951
RBS - UC - GM	0.961
TS - TPC - GM	0.971
TS - UC - GM	0.971

Tabelle 17: Durchschnittliche Ergebnisse nach jeweils 20 Durchläufen für f_2 (gerundet)

Implementierung	Evaluationen	Ausführungszeit	Speicherauslastung (MiB)
RBS - TPC - UM	29020	363ms	23.1953
RBS - UC - UM	29071	383ms	23.2187
TS - TPC - UM	29010	386ms	23.2343
TS - UC - UM	29057	403ms	23.1445
RBS - TPC - GM	29099	382ms	23.1875
RBS - UC - GM	29025	412ms	23.2382
TS - TPC - GM	29026	408ms	23.2656
TS - UC - GM	29005	423ms	23.2539

Tabelle 18: Durchschnittliche $r_{0.05}$ Ergebnisse nach jeweils 10 Durchläufen für das TSP (gerundet)

Implementierung	$r_{0.05}$
RBS - OC - IM	0.565
RBS - OC - SM	0.487
RBS - PMC - IM	0.726
RBS - PMC - SM	0.377
TS - OC - IM	0.498
TS - OC - SM	0.429
TS - PMC - IM	0.579
TS - PMC - SM	0.229

Tabelle 19: Durchschnittliche Ergebnisse für die genetischen Algorithmen nach jeweils 10 Durchläufen beim TSP (gerundet)

Implementierung	Evaluationen	Ausführungszeit	Speicherauslastung (MiB)
RBS - OC - IM	440630	69 sec	12.492
RBS - OC - SM	440129	67 sec	12.378
RBS - PMC - IM	440457	73 sec	12.500
RBS - PMC - SM	440212	74 sec	12.457
TS - OC - IM	440846	69 sec	12.394
TS - OC - SM	439642	66 sec	12.406
TS - PMC - IM	439695	75 sec	12.453
TS - PMC - SM	440005	80 sec	12.351

Sektion: 6.3

Tabelle 20: Durchschnittliche Ergebnisse für die Evolutionsstrategien nach jeweils 10 Durchläufen bei f_1 (gerundet)

Implementierung	Evaluationen	Ausführungszeit	Speicherauslastung (MiB)
K - TPC	5924	336 ms	48.5
K - UC		332 ms	48.4
P - UC		328 ms	48.5
P - TPC		331 ms	48.5

Tabelle 21: Durchschnittliche $r_{0.05}$ Ergebnisse nach jeweils 10 Durchläufen für f_1 (gerundet)

Implementierung	$r_{0.05}$
K - TPC	0.667
K - UC	0.567
P - TPC	0.900
P - UC	0.900

Tabelle 22: Durchschnittliche Ergebnisse für die Evolutionsstrategien nach jeweils 20 Durchläufen bei f_2 (gerundet)

Implementierung	Evaluationen	Ausführungszeit	Speicherauslastung (MiB)
K - TPC	29134	741 ms	48.5
K - UC		714 ms	48.6
P - UC		686 ms	48.6
P - TPC		693 ms	48.7

Tabelle 23: Durchschnittliche Ergebnisse für die Evolutionsstrategien nach jeweils 10 Durchläufen beim TSP (gerundet)

Implementierung	Evaluationen	Ausführungszeit	Speicherauslastung (MiB)
K - TPC - IM	420015	75 sec	48.9
K - TPC - SM		77 sec	48.9
K - UC - IM		82 sec	48.6
K - UC - SM		81 sec	49.0
P - TPC - IM		71 sec	48.9
P - TPC - SM		66 sec	48.8
P - UC - IM		75 sec	48.8
P - UC - SM		76 sec	48.9

Tabelle 24: Durchschnittliche $r_{0.05}$ Ergebnisse nach jeweils 10 Durchläufen für das TSP (gerundet)

Implementierung	$r_{0.05}$
K - TPC - IM	0.282
K - TPC - SM	0.164
K - UC - IM	0.167
K - UC - SM	0.354
P - TPC - IM	0.673
P - TPC - SM	0.450
P - UC - IM	0.756
P - UC - SM	0.621

Source Code

Nachfolgend sind alle verwendeten Implementierungen der Selektions-, Rekombinations- und Mutationsoperatoren der genetischen Algorithmen abgebildet. Die entsprechenden Operatoren für die Evolutionsstrategien funktionieren analog zu diesen.

```
def tournament_select_chromosomes(population):
"""Uses tournament selection to select chromosomes out of
the population

Args:
    population

Returns:
    [chromosomes]: selected chromosomes
"""
new_population = []
for _ in population:
    random_choice = random.choices(population, k=2)
    best_chromosome = sorted(random_choice, key= lambda x: x.fitness, reverse=
  True)[0]
    new_population.append(best_chromosome)

return new_population
```

```
def rank_based_select_chromosomes(population):
  """Uses rank based selection to select chromosomes out of
  the population

  Args:
      population

  Returns:
      [chromosomes]: selected chromosomes
  """
  sorted_chromosomes = sorted(population, key= lambda x: x.fitness, reverse=
  True)

  weight = [i**2 for i in range(len(population), 0, -1)]

  # selects amount-many chromosomes based on their weight
  # the higher the weight the better the chances to be selected
  # It's possible to get duplicates
  chromosomes = random.choices(sorted_chromosomes, weights=weight, k=len(
  population))

  return chromosomes
```

```
def two_point_crossover(population):
  """Uses two point crossover to create a new population, based
  on given probability args.crossover_p

  Args:
      population ([chromosomes])

  Returns:
      new population
  """
  new_population = []
  while len(population) > 1:
      parents = random.sample(population, k=2)
      if random.random() < args.crossover_p:
          parent1, parent2 = parents[0].genes, parents[1].genes

          # get the two crossover points
          index1 = random.randint(0, len(parent1) - 1)
          index2 = random.randint(0, len(parent2) - 1)

          if index1 > index2:
              index1, index2 = index2, index1

          child = parent1[0:index1] + parent2[index1:index2] + parent1[index2:]
          child2 = parent2[0:index1] + parent1[index1:index2] + parent2[index2
  :]

          new_population.append(Chromosome(child))
          new_population.append(Chromosome(child2))
      else:
      # add parents without crossover
          new_population.extend(parents)
      population.remove(parents[0])
      population.remove(parents[1])

  # if len(population) was odd, one last chromosome will be added without
  crossover
  if population:
      new_population.append(population[0])

  return new_population
```

```
def uniform_crossover(population):
  """Uses uniform crossover to create a new population, based
  on given probability args.crossover_p

  Args:
      population ([chromosomes])

```

```
    Returns:
        new population
    """
    new_population = []
    while len(population) > 1:
        parents = random.sample(population, k=2)
        if random.random() < args.crossover_p:
            parent1, parent2 = parents[0].genes, parents[1].genes

            child1 = []
            child2 = []

            for child in [child1, child2]:
                for i,_ in enumerate(parent1):
                    if random.randint(0, 1) == 0:
                        child.append(parent1[i])
                    else:
                        child.append(parent2[i])

            new_population.append(Chromosome(child1))
            new_population.append(Chromosome(child2))
        else:
        # add parents without crossover
            new_population.extend(parents)
        population.remove(parents[0])
        population.remove(parents[1])

    # if len(population) was odd, one last chromosome will be added without
    crossover
    if population:
        new_population.append(population[0])

    return new_population
```

```
def uniform_mutation(population):
    """Uses uniform mutation to mutate a whole population, based
    on given probability args.mutation_p

    Args:
        population ([chromosomes])

    Returns:
        population: mutated Population
    """
    new_population = []
    for chromosome in population:
        new_chromosome = []
        for gene in chromosome.genes:
            if random.random() < args.mutation_p:
```

```
                gene = random.uniform(args.min, args.max)
            new_chromosome.append(gene)
        new_population.append(Chromosome(new_chromosome))
    return new_population
```

```
def gaussian_mutation(population):
    """Uses gaussian mutation to mutate a whole population, based
    on given probability args.mutation_p

    Args:
        population ([chromosomes])

    Returns:
        population: mutated Population
    """
    new_population = []
    for chromosome in population:
        new_chromosome = []
        for gene in chromosome.genes:
            if random.random() < args.mutation_p:
                gene = numpy.random.normal(gene, 0.3)
            new_chromosome.append(gene)
        new_population.append(Chromosome(new_chromosome))
    return new_population
```

```
def order_crossover(population):
    """Uses order crossover to create a new population, based
    on given probability args.crossover_p

    Args:
        population ([chromosomes])

    Returns:
        new population
    """
    new_population = []
    while len(population) > 1:
        # select two parents, can't be the same
        parents = random.sample(population, k=2)
        if random.random() < args.crossover_p:
            parent1, parent2 = parents[0].route, parents[1].route

            # get the two crossover points
            index1 = random.randint(0, len(parent1) - 1)
            index2 = random.randint(0, len(parent1) - 1)

            if index1 > index2:
                index1, index2 = index2, index1

```

```
            ## child 1
            saved_part = parent1[index1:index2 + 1]
            # fill with the cities that are left (ordered)
            child = [gene for gene in parent2 if gene not in saved_part]
            # insert saved_part at index1
            child[index1:index1] = saved_part

            ## child 2
            saved_part2 = parent2[index1:index2 + 1]
            # fill with the cities that are left (ordered)
            child2 = [gene for gene in parent1 if gene not in saved_part2]
            # insert saved_part2 at index1
            child2[index1:index1] = saved_part2

            new_population.append(Chromosome(child))
            new_population.append(Chromosome(child2))
        else:
            # add parents without crossover
            new_population.extend(parents)
        population.remove(parents[0])
        population.remove(parents[1])

    # if len(population) was odd, last chromosome will be added without crossover
    if population:
        new_population.append(population[0])

    return new_population
```

```
def partially_mapped_crossover(population):
    """Uses partially mapped crossover to create a new population, based
    on given probability args.crossover_p

    Args:
        population ([chromosomes])

    Returns:
        new population
    """
    def get_mapping(gene):
        """Changes genes according to the dictionary, loops until
        nothing changes anymore
        """
        t = mapping[gene]
        while (x := mapping[t]) != t:
            t = x
        return t

    new_population = []
    while len(population) > 1:
```

```
22          parents = random.sample(population, k=2)
23          if random.random() < args.crossover_p:
24              parent1, parent2 = parents[0].route, parents[1].route
25
26              # get the two crossover points
27              index1 = random.randint(0, len(parent1) - 1)
28              index2 = random.randint(0, len(parent2) - 1)
29
30              if index1 > index2:
31                  index1, index2 = index2, index1
32
33              mapping_section1 = parent1[index1:index2 + 1]
34              mapping_section2 = parent2[index1:index2 + 1]
35
36              ## child 1
37              mapping = dict()
38              # standard mapping
39              for i in range(1, len(parent1) + 1):
40                  mapping[i] = i
41              # changing mapping from mapping_section1 to mapping_section2
42              for i, gene in enumerate(mapping_section1):
43                  mapping[gene] = mapping_section2[i]
44
45              child = [get_mapping(gene) for gene in parent2 if gene not in
    mapping_section2]
46              # insert saved_part at index1
47              child[index1:index1] = mapping_section1
48
49              ## child 2
50              # standard mapping
51              for i in range(1, len(parent1) + 1):
52                  mapping[i] = i
53              # changing mapping from mapping_section2 to mapping_section1
54              for i, gene in enumerate(mapping_section2):
55                  mapping[gene] = mapping_section1[i]
56
57              child2 = [get_mapping(gene) for gene in parent1 if gene not in
    mapping_section1]
58              # insert saved_part at index1
59              child2[index1:index1] = mapping_section2
60
61              new_population.append(Chromosome(child))
62              new_population.append(Chromosome(child2))
63          else:
64              # add parents without crossover
65              new_population.extend(parents)
66          population.remove(parents[0])
67          population.remove(parents[1])
68
```

```
69      # if len(population) was odd, last chromosome will be added without crossover
70      if population:
71          new_population.append(population[0])
72
73      return new_population
```

Der vollständige Code ist auf der beigefügten CD zu finden.

eil101.tsp

Hier ist der Inhalt der *eil101.tsp* Datei angegeben. Sie beeinhaltet die X- und Y-Koordinaten der 101 Städte. Die kleinste Distanz/Lösung für dieses Problem beträgt 629.

City ID	X	Y
1	41	49
2	35	17
3	55	45
4	55	20
5	15	30
6	25	30
7	20	50
8	10	43
9	55	60
10	30	60
11	20	65
12	50	35
13	30	25
14	15	10
15	30	5
16	10	20
17	5	30
18	20	40
19	15	60
20	45	65
21	45	20
22	45	10
23	55	5
24	65	35
25	65	20
26	45	30
27	35	40
28	41	37
29	64	42
30	40	60
31	31	52
32	35	69
33	53	52
34	65	55

City ID	X	Y
35	63	65
36	2	60
37	20	20
38	5	5
39	60	12
40	40	25
41	42	7
42	24	12
43	23	3
44	11	14
45	6	38
46	2	48
47	8	56
48	13	52
49	6	68
50	47	47
51	49	58
52	27	43
53	37	31
54	57	29
55	63	23
56	53	12
57	32	12
58	36	26
59	21	24
60	17	34
61	12	24
62	24	58
63	27	69
64	15	77
65	62	77
66	49	73
67	67	5
68	56	39

City ID	X	Y
69	37	47
70	37	56
71	57	68
72	47	16
73	44	17
74	46	13
75	49	11
76	49	42
77	53	43
78	61	52
79	57	48
80	56	37
81	55	54
82	15	47
83	14	37
84	11	31
85	16	22
86	4	18
87	28	18
88	26	52
89	26	35
90	31	67
91	15	19
92	22	22
93	18	24
94	26	27
95	25	24
96	22	27
97	25	21
98	19	21
99	20	26
100	18	18
101	35	35