Studienarbeit Nr. 2071

Visualisierung des Baldwin-Effekts am Beispiel des Mengenüberdeckungsproblems

Lukas König

Prüfer: Prof. Dr. Volker Claus
Betreuer: Dipl.-Inform. Botond Draskoczy
Beginn am: 04. Juli 2006
Beendet am: 03. Januar 2007
CR-Klassifikation: F.1.2, F.2.2, G.2.1, I.2.8

2. Juli 2012

Institut für Formale Methoden der Informatik,
Abteilung Formale Konzepte, Universität Stuttgart

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation und Kurzeinführung

Diese Arbeit beschreibt die Implementierung eines evolutionären Algorithmus für das Set Covering Problem (SCP) unter Einsatz von Baldwinschem Lernen und die eines darauf aufbauenden Programms zur Visualisierung Baldwinschen Lernens.

Das SCP ist ein kombinatorisches Problem, bei dem eine Menge M und eine Menge T von Teilmengen von M zusammen mit einer Kostenfunktion gegeben sind. Die Aufgabe ist es, eine Auswahl der Elemente aus T (also Teilmengen von M) zu finden, die M *überdeckt* und zusätzlich geringstmögliche Kosten hat. M heißt dabei überdeckt, wenn die Vereinigung aller Elemente der Auswahl die Menge M ergibt. Die Kostenfunktion wird je nach Komplexität in Unicost, Multicost und Generalcost unterteilt (siehe Abschnitt 2.6 für eine genaue Definition des SCP).

Wie Richard Karp bereits 1972 zeigte, ist das SCP NP-hart [Karp1972]. Es gibt aber viele praktische Anwendungen, die man auf das SCP reduzieren kann; unter den verbreitetsten ist bspw. das *Crew Scheduling* - Problem, bei dem es um die Zuordnung von Flugpersonal zu Flügen geht. Daher ist es wichtig, Heuristiken zu entwickeln, die gute Näherungslösungen mit vertretbarem Zeitaufwand finden.

Ein naheliegender Ansatz ist ein Greedy-Algorithmus, der die Elemente aus T nacheinander in die Auswahl aufnimmt, wobei unter den noch nicht aufgenommenen Elementen immer dasjenige als nächstes ausgewählt wird, bei welchem der Quotient

$$\frac{\text{Kosten dieser Teilmenge}}{\text{Anzahl der enthaltenen Indizes aus } M\text{, die noch nicht überdeckt sind}}$$

am kleinsten ist. Man kann zeigen, dass bei Unicost-SCP dieser Algorithmus bereits bis auf einen Faktor von höchstens $H(t)$ an die optimale Lösung herankommt, wobei t die Größe der größten Teilmenge in T ist und $H(t)$ die t-te harmonische Zahl. Für Generalcost-SCP ist dieser Algorithmus allerdings ungeeignet.

Die Arbeit von Marchiori und Steenbeek [Marchiori1998] beschreibt einen weiteren Greedy-Algorithmus, allerdings ausschließlich zum Lösen von Unicost-SCP, der in diesem Bereich bessere Lösungen liefert.

Innerhalb des Bereichs der evolutionären Algorithmen beschreiben Beasley et al. in ihrer Arbeit [Beasley1994] eine Heuristik zum Lösen des SCP, die prinzipiell auf einem genetischen Algorithmus basiert, aber ohne Generationenzyklus auskommt. In der Arbeit von Eremeev [Eremeev1999] wird ein evolutionärer Algorithmus mit nicht-binärer Repräsentation der Genome vorgestellt. Die Arbeit von Aickelin [Aickelin2000] beschreibt einen 2-Phasen-Algorithmus, dessen eine Phase durch einen evolutionären Algorithmus durchgeführt wird. Ein weiterer evolutionärer Ansatz wird in der Arbeit von Marchiori und Steenbeek [Marchiori2000] beschrieben. All diese Algorithmen arbeiten ohne Baldwinsches Lernen.

Unter Baldwinschem Lernen wird hier die Fähigkeit eines Individuums verstanden, zu „Lebzeiten“ in seiner „Umgebung“ nach möglichen Veränderungen seiner selbst zu suchen, die seine Fitness verbessern können und diese für die Dauer seines Lebens anzunehmen. Im Unterschied zu Lamarckschem Lernen wird diese Veränderung nicht ins Genom geschrieben und hat somit keinen Einfluss auf die Nachkommen des Individuums. Eine Beschreibung und Definition von Baldwinschem Lernen findet sich in Abschnitt 2.4.

Es konnte keine Arbeit gefunden werden, die einen evolutionären Algorithmus beschreibt, der Baldwinsches Lernen zur Unterstützung der Evolution beim Lösen des SCP verwendet. Dennoch konnten Hinton und Nowlan [Hinton1987], Gruau und Whitley [Gruau1993], Whitley, Gordon und Mathias [Whitley1994] und andere zeigen, dass Baldwinsches Lernen zu einer Verbesserung der Ergebnisse von evolutionären Algorithmen führen kann. Im ersten Teil dieser Arbeit wird untersucht, wie erfolgreich dieser Ansatz für das SCP angewendet werden kann.

Der zweite Teil dieser Arbeit befasst sich mit der Visualisierung von Suchräumen diskreter Optimierungsprobleme (insbesondere des SCP) und der Veränderung von Suchräumen durch Baldwinsches Lernen. Bei stetigen Problemem mit einer oder zwei Dimensionen kann eine Visualisierung der Suchräume durch einfache Funktionsplotter erfolgen. In der Arbeit von Whitley, Gordon und Mathias [Whitley1994] wurde unter anderem eine Visualisierung der Suchraumveränderung durch Baldwinsches Lernen bei stetigen Suchräumen mit einer oder zwei Dimensionen durchgeführt. Auch als Einstieg in diese Studienarbeit wurde ein solches Plot-Programm geschrieben, welches hier nur am Rande erwähnt wird. In der Informatik sind Optimierungsprobleme aber meist diskret und haben sehr viel mehr als zwei Dimensionen.

Bei diskreten Problemen stellt sich zunächst die Frage nach einem Nachbarschaftsbegriff im Suchraum, anhand dessen eine Visualisierung erfolgen könnte. Ohne eine Nachbarschaft sind die einzelnen Elemente des Suchraums isoliert und es ist nicht klar, wie eine Visualisierung aussehen soll. Bei (ein- oder zweidimensionalen) stetigen Problemen ist diese Nachbarschaft intuitiv durch die räumliche Nähe gegeben. Durch Einführen eines künstlichen Nachbarschaftsbegriffs im Suchraum eines diskreten Problems kann der Suchraum anhand dieser Nachbarschaft als Graph aufgefasst werden. Die Visualisierung des Suchraums reduziert sich dann auf eine Visualisierung des zugehörigen Graphen.

Die Nachbarschaft sollte in möglichst intuitiver Weise Elemente des Suchraums verbinden, die einander in ihrer Struktur „ähnlich“ sind. Da die Visualisierung letztendlich die Sicht eines evolutionären Algorithmus auf den Suchraum repräsentieren soll, bietet es sich an, bspw. dessen Mutationsoperator für die Definition der Nachbarschaft zu verwenden. Ein Individuum (oder Element des Suchraums) ist dann benachbart mit einem anderen, wenn aus dem ersten durch Anwenden des Mutationsoperator das zweite entstehen kann. Eine kompliziertere Nachbarschaft wäre über den Rekombinationsoperator denkbar. Diese müsste einer Menge von Eltern eine Menge von Kindern als Nachbarn zuordnen.

In dieser Arbeit wird für Suchräume des SCP eine Nachbarschaft über den Mutationsoperator eines evolutionären Algorithmus definiert und darauf aufbauend eine Visualisierung des entstehenden Suchraumgraphen vorgenommen. Dabei kann auch eine Veränderung des Suchraums durch Einsatz von Baldwinschem Lernen angezeigt werden. Die Veränderung des Suchraums betrifft bei Baldwinschem Lernen nicht die durch die Nachbarschaft definierte Topologie, sondern nur die Bewertung der Elemente.

Die Hoffnung ist, durch die Visualisierung eine bessere Vorstellung von Suchräumen zu gewinnen und diese auch in der Lehre weitervermitteln zu können.

Alle für diese Studienarbeit implementierten Programme wurden in Java (Version 1.5.0_06) geschrieben. Als Entwicklungsumgebung wurde Eclipse SDK (Version: 3.1.2) verwendet. Die Entwicklung erfolgte zum großen Teil auf dem Betriebssystem Microsoft Windows XP, alle Programme wurden jedoch auch unter Linux getestet; insbesondere wurden alle Testläufe mit Linux durchgeführt. Die Profiler-Läufe wurden mit dem Java-internen Profiler (Java Profile, Version 1.0.1) durchgeführt.

1.2 Gliederung und Anlage

Die vorliegende Arbeit unterliegt der folgenden Gliederung: In Abschnitt 2 werden die Grundlagen beschrieben, die zum Verständnis der Arbeit notwendig sind. In Abschnitt 3 wird der entworfene und implementierte evolutionäre Algorithmus beschrieben. In Abschnitt 4 wird eine Reihe von Testläufen beschrieben und ausgewertet, die mit dem evolutionären Algorithmus durchgeführt worden sind. In Abschnitt 5 wird der Entwurf und die Implementierung eines auf dem evolutionären Algorithmus aufbauenden Programms zum Visualisieren von Suchräumen und von Baldwinschem Lernen beschrieben. In Abschnitt 6 werden einige Vorschläge gemacht, in welcher Richtung weitergehende Forschung sinnvoll und interessant sein könnte.

Der Anhang listet alle durchgeführten Testläufe mit ihren jeweiligen Endergebnissen auf (Teil A) und gibt einen Überblick über die implementierten Klassen und Pakete (Teil B).

Anlage zu dieser Studienarbeit ist eine CD-ROM, die folgende Elemente enthält:

- Eine elektronische Version dieser Ausarbeitung.
- Das komplette implementierte Projekt, bestehend aus dem evolutionären Algoritmus und dem Visualisierungs-Applet; dabei sind alle Quelldateien und eine kompilierte Version. Dazu gehört auch eine Version des Frameworks von Dietmar Lippold, auf dem der evolutionäre Algorithmus aufbaut; *es wird darauf hingewiesen, dass jede Weitergabe dieses Frameworks oder seiner Teile untersagt ist.*
- Eine Dokumentation des Projekts im Javadoc-Format.
- Sämtliche Statistiken zu den durchgeführten Testläufen. Über die gedruckten Statistiken in dieser Arbeit hinausgehend sind darunter unter anderem Statistiken zu jeder einzelnen berechneten Generation und Diagramme im Excel-Format zu allen Testläufen. Außerdem gibt es eine große Excel-Tabelle mit allen Endergebnissen aus dem Anhang dieser Ausarbeitung.
- Einige Protokolle zur Entstehung der Studienarbeit.

2 Grundlagen

Dieser Abschnitt erläutert theoretische Grundlagen, auf denen die Arbeit aufbaut.

2.1 Mathematische Symbole und Schreibweisen

Die allgemein üblichen mathematischen Begriffe, Symbole, Funktionen, etc., sowie speziell in der Informatik gebräuchliche Begriffe wie *NP-Vollständigkeit*, etc. werden ohne vorherige Definition benutzt.

Boolesche Operatoren, Wahrheitswerte und Bitstrings Die Booleschen Operationen $\neg, \vee, \wedge$ werden über der Menge $\mathbb{B} = \{true, false\}$ wie üblich definiert. $true$ wird auch durch 1, $false$ durch 0 identifiziert.

Ein Bitstring B der Länge $n \in N$ ist ein n-Tupel mit einem Element der Menge $\mathbb{B}$ an jeder Position. (Formal ist B für $n > 0$ eine Funktion $B : \{1, ..., n\} \to \mathbb{B}$. Für $n = 0$ ist $B = \epsilon$ das leere Wort.)

$|B|$ bezeichnet die Länge n des Bitstrings. Ein Bitstring wird gewöhnlich abgekürzt geschrieben, indem die bei der Tupel-Schreibweise üblichen Klammern und Kommas weggelassen werden: $(b_1, b_2, ..., b_n) \mathrel{\widehat{=}} b_1 b_2 ... b_n$.

Für $i \in \{1, ..., n\}$ bezeichnet $B[i]$ das i-te Bit des Bitstrings (oder formal den Funktionswert $B(i)$).

Wenn B ein Bitstring ist, dann bezeichnet B^m für $m \in \mathbb{N}$ die m-fache Hintereinanderschreibung von B. Insbesondere ist $B^0 = \epsilon$ das leere Wort.

(Formal ist B^m für $m > 0$ die aus B entstehende, im Definitionsbereich erweiterte Funktion $B^m : \{1, 2, ..., m \cdot |B|\} \to \mathbb{B}$, wobei für alle $1 \leq k \leq m \cdot |B|$ gilt:

$$B^m[k] = B[((k-1) \mod |B|) + 1].$$

Für $m = 0$ gilt der Spezialfall $B^0 = \epsilon$.)

Für $n \in \mathbb{N}$ bezeichnet $\mathbb{B}^n$ die Menge aller Bitstrings der Länge n.

Potenzmenge Die Potenzmenge einer Menge M wird mit $\mathcal{P}(M)$ bezeichnet. Es gilt:

$$\mathcal{P}(M) = \{M' | M' \subseteq M\}.$$

Intervalle Ein Intervall $[i_1, i_2]$ von $i_1 \in \mathbb{R}$ bis $i_2 \in \mathbb{R}$ ist die Teilmenge der reellen Zahlen, die größer oder gleich i_1 und kleiner oder gleich i_2 sind:

$$[i_1, i_2] =_{def} \{x \in \mathbb{R} | i_1 \leq x \leq i_2\}.$$

Offene Intervalle werden mit $]i_1, i_2]$, $[i_1, i_2[$ bzw. $]i_1, i_2[$ bezeichnet und meinen die Mengen $[i_1, i_2] \backslash \{i_1\}$, $[i_1, i_2] \backslash \{i_2\}$ bzw. $[i_1, i_2] \backslash \{i_1, i_2\}$.

Graph Ein (gerichteter) Graph ist ein 2-Tupel (E, V), wobei V eine endliche Menge ist und $E \subseteq V \times V$. Die Elemente der Menge V heißen Knoten. Ein Element $e = (v_1, v_2) \in E$ heißt (gerichtete) Kante von v_1 nach v_2.

In dieser Arbeit werden keine ungerichteten Graphen betrachtet.

2.2 Evolutionäre Algorithmen

Dieser Abschnitt ist an das Buch von Karsten Weicker [Weicker2002] angelehnt.

Als evolutionäre Algorithmen werden im Allgemeinen Strategien zur Lösung von Optimierungsproblemen bezeichnet, deren Mechanismen von den Prinzipien der natürlichen Evolution inspiriert sind. Je nach Abstrahierungsgrad werden dabei die in der Natur zwischen Organismen parallel stattfindenden Prozesse auf ein mehr oder weniger vereinfachtes Modell abgebildet, in welchem in einem ständigen Kreislauf sog. evolutionäre Operatoren auf eine Population künstlicher Individuen angewendet werden. Die künstlichen Individuen sind dabei Lösungskandidaten einer Instanz eines Optimierungsproblems, die sich im Verlauf der

künstlichen Evolution der optimalen Lösung annähern sollen. Während die evolutionären Operatoren üblicherweise der Natur nachempfunden sind (es handelt sich gewöhnlich um Elternselektion, Rekombination, Mutation und Umweltselektion), müssen zusätzlich einige Rahmenbedingungen hergestellt werden, die in der Natur nicht vorhanden sind. Dazu gehören beispielsweise das Erzeugen einer Startpopulation und das Setzen einer Terminierungsbedingung, welche den evolutionären Zyklus unterbricht. Abbildung 1 zeigt schematisch einen einfachen evolutionären Zyklus.

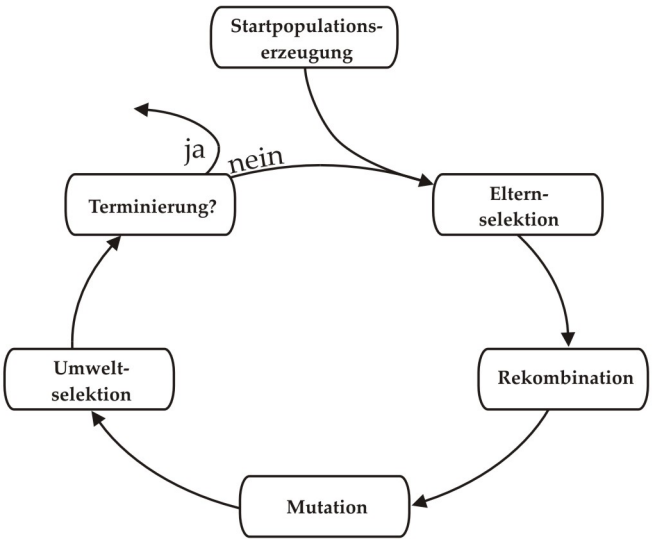

Abbildung 1: *Der künstliche evolutionäre Zyklus*

Biologisch geht man heute bei Lebewesen meist von einer gewissen Trennung zwischen dem Genotyp und dem durch ihn kodierten Phänotyp aus; diese Trennung ist als die Weisman-Barriere bekannt. Der Genotyp bestimmt den Phänotyp, er enthält dauerhafte Informationen, die sich nur langsam und unter großem Einfluss von Zufall verändern und die im Prozess der Evolution gewährleisten, dass Eigenschaften über Generationen hinweg erhalten bleiben können; der Phänotyp muss sich dagegen in der Umwelt bewähren, nur er wird einer Auslese unterzogen und er hat umgekehrt keinerlei Einfluss auf den Genotyp.

Bei evolutionären Algorithmen wird gewöhnlich ebenfalls zwischen Genotyp und Phänotyp unterschieden und die beschriebene Trennung wird auch hier oft eingehalten. Während der Genotyp die Kodierung einer Problemlösung enthält und durch Rekombination und Mutation verändert werden kann, wird dem Phänotyp über eine Fitnessfunktion eine Güte zugewiesen, die beeinflusst, wie lange das Individuum überlebt und ob es sich fortpflanzen darf. Man kann an dieser Stelle auch zulassen, dass das Individuum (dessen Phänotyp) in jeder Generation eine Art Interaktion mit seiner Umwelt durchführen darf, indem es eine lokale Suche im Lösungsraum durchführt und daraus Schlüsse zu einer Verbesserung seiner Fitness zieht; das kann als Lernen aufgefasst werden. Wenn dabei die Trennung zwischen Genotyp und Phänotyp eingehalten wird, können Effekte wie der Baldwin-Effekt (siehe Abschnitt 2.4) auftreten, wird doch eine Rückkopplung von Phänotyp zu Genotyp zugelassen findet Lamarcksche Evolution (siehe Abschnitt 2.3) statt.

Um zu beschreiben, was Genotyp und Phänotyp sind, benötigt man zunächst eine Definition

von Optimierungsproblemen.

Definition (Optimierungsproblem)

> Ein Optimierungsproblem ist gegeben durch einen Suchraum Ω, eine Bewertungsfunktion $f : \Omega \to \mathbb{R}$, die jedem Lösungskandidaten einen Gütewert zuweist, und eine Vergleichsrelation $\succ \in \{<, >\}$.
>
> Ist $\succ = >$, heißt das Problem Maximierungsproblem, sonst Minimierungsproblem.
>
> Die Menge der globalen Optima ist definiert als
>
> $$X = \left\{ x \in \Omega | \forall x' \in \Omega : f(x) \succeq f(x') \right\}.$$

Die Lösung eines Optimierungsproblems ist demnach das Finden mindestens eines globalen Optimums.

Der Phänotyp eines Individuums ist ein Element aus dem Suchraum Ω, weshalb Ω auch *phänotypischer Suchraum* genannt wird. Der *Genotyp* oder das *Genom* ist ein Element aus dem *genotypischen Raum* Γ, der je nach Art des Problems spezifisch gewählt werden kann; insbesondere kann $\Gamma = \Omega$ gelten. Es muss aber gewährleistet sein, dass für jedes Element aus Ω ein Element in Γ existiert. Falls Γ ein von Ω verschiedener Raum ist, benötigt man für die Zuordnung vom genotypischen zum phänotypischen Suchraum, insbesondere für die Bewertung von Genomen, eine Dekodierungsfunktion *dec*, die jedem Genom den zugehörigen Phänotyp zuweist.

Die Umkehrfunktion von *dec*, die jedem Phänotyp ein Genom zuweist, soll *cod* genannt werden.

Die Bewertung F eines Genoms $g \in \Gamma$ ist dann $F = f(dec(g))$. Im Folgenden wird auf diese Schreibweise aber verzichtet und der einfacheren Lesbarkeit wegen $F = f(g)$ geschrieben; die vorherige Anwendung einer Dekodierungsfunktion wird dabei implizit vorausgesetzt.

Der genotypische Suchraum wird bei dem später in dieser Arbeit beschriebenen Algorithmus eine Teilmenge von $\{0, 1\}^n$ sein, wobei $n \in \mathbb{N}$ die Anzahl der durch das zu lösende Mengenüberdeckungsproblem definierten Teilmengen $|T|$ ist (siehe Abschnitt 2.6 und 3.1.1). Ein Genotyp ist in diesem Fall einfach ein Bitstring der Länge n (mit einigen Zusatzeigenschaften, die später beschrieben werden).

Ein Individuum wird in der Literatur häufig als ein 3-Tupel aus Genotyp, Zusatzinformation und Fitness definiert [Weicker2002]. Da im Folgenden aber auf die Zusatzinformation verzichtet wird, wird das Individuum als 2-Tupel definiert.

Definition (Individuum)

> Ein Individuum I ist ein 2-Tupel $(I.G, I.F)$, wobei $I.G \in \Gamma$ der Genotyp ist und $I.F = f(I.G)$ die Güte des Individuums.
>
> Die Menge $Ind = \Gamma \times \mathbb{R}$ ist der Raum aller Individuen.

Dass die Fitness Teil des Individuums ist, ist nicht nur wegen der Analogie zu Organismen in der Umwelt intuitiv richtig, sondern stellt sich auch in den meisten Fällen als sinnvolle Implementierungsmaxime heraus; da die Berechnung der Fitness oft aufwändig ist und durchaus mehrere Abfragen der Fitness bei unverändertem Genom vorkommen können, ist

es eine effiziente Lösung, sie beim Individuum zu speichern und nur bei Veränderung des Genoms neu zu berechnen.

Eine Population könnte als Multimenge von Individuen definiert werden, denn es ist denkbar und oft auch erwünscht, dass Individuen mehrfach vorkommen, und die Population ist zunächst nicht sortiert. Dennoch wird im Folgenden eine Population als Tupel definiert, um einfacher bestimmte Individuen benennen, bzw. unmittelbarer eine Sortierung angeben zu können.

Definition (Population)

Eine Population P der Größe $n \in \mathbb{N}$ ist ein n-Tupel aus Individuen:

$$P = (I_1, ..., I_n),$$

wobei $\forall i \in \{1, ..., n\} : I_i \ \in Ind.$

2.2.1 Evolutionäre Operatoren

Die evolutionären Operatoren sind im künstlichen evolutionären Zyklus einerseits dafür verantwortlich, eine Population während ihrer Entwicklung genügend variabel zu halten, damit nicht alle Individuen in einem lokalen Optimum steckenbleiben; andererseits müssen sie die Population aber auch durch Selektionsdruck an einer in zu großem Maße zufälligen Verteilung über den Suchraum hindern.

Selektion In der Natur steuert die Selektion durch die Anzahl an Nachkommen die Häufigkeit von Genomen, wobei „bessere“ Genome tendenziell häufiger vorkommen. Die Güte oder Fitness von Genen spiegelt sich in der Fähigkeit des zugehörigen Phänotyps wider, sich fortzupflanzen. Diese Fähigkeit kann durch vielerlei Faktoren beeinflusst sein, z. B. die Überlebenschancen in der aktuellen Umwelt oder das Vermögen, einen Geschlechtspartner zu finden. Die relative Fitness eines Genotyps G in einer (natürlichen) Population ist definiert als

$$Fitness(G) = \frac{\#Nachkommen(G)}{\#Nachkommen(G')},$$

wobei G' der Genotyp mit den meisten Nachkommen der Population ist.

Während sich in der Natur also die Fitness implizit aus der Anzahl der Nachkommen eines Organismus ergibt, und es damit auch kein absolutes Ziel gibt, auf das die Evolution hinsteuert, hat man bei evolutionären Algorithmen eine feste Vorstellung von einem Ziel, nämlich der Lösung eines Optimierungsproblems. Um zu gewährleisten, dass die Evolution in die richtige Richtung geht, setzt man die Fitnessfunktion fest und macht umgekehrt die Selektion von dieser Fitness abhängig. Als Fitnessfunktion wählt man die Bewertungsfunktion f des jeweiligen Optimierungsproblems. Von zwei Individuen hat dann immer dasjenige eine höhere Chance sich fortzupflanzen, welches die bessere Fitness hat.

Allgemein kann die Selektion formal als eine Abbildung aus dem Raum der Populationen in den Raum der Populationen definiert werden. Da sie jedoch keine neuen Individuen erzeugen kann, sondern nur schon vorhandene aussuchen, kann der Selektionsoperator auch konkreter auf die Selektion von Indizes einer Population zurückgeführt werden.

Definition (Selektionsoperator)

Sei $P = (I_1, ..., I_r)$ eine Population und $\xi \in \Xi$ ein Zustand des Zufallsgenerators.

- Ein Selektionsoperator S bildet die Population P der Größe r auf eine Population der Größe $s \in \mathbb{N}$ ab und ist formal eine Abbildung

$$S^{\xi} : Ind^r \to Ind^s,$$

- Konkrete Selektionsoperatoren können auch über den Indexselektionsoperator 15

$$IS^{\xi} : Ind^r \to \{1, ..., r\}^s$$

definiert werden. In diesem Fall wird eine Selektion $S^{\xi}(P) = (I_{j_1}, ..., I_{j_s})$, $j_k \in \{1, ..., r\}\ \forall\ k$ durch die Indexselektion $IS^{\xi}(P) = (j_1, ..., j_s)$ bestimmt.

Die Selektion wird in der Natur durch viele verschiedene Faktoren gesteuert, die in einem evolutionären Algorithmus vereinfachend auf zwei Stellen im evoultionären Zyklus reduziert werden. Die Selektion der Individuen vor der Rekombination entspricht dabei einer Selektion nach der Fähigkeit, die Fortpflanzung zu vollziehen; die Selektion nach der Mutation entspricht einer Auslese aufgrund der Überlebensfähigkeit in der jeweiligen Umwelt. Manchmal wird eine der beiden Selektionsarten weggelassen oder durch eine fitnessunabhängige Selektion ersetzt, die beispielsweise dazu dienen kann, eine im Lauf des Zyklus aufgeblähte Population auf eine bestimmte Größe zu verkleinern.

Es wird zwischen duplikatfreien und nicht-duplikatfreien, sowie zwischen probabilistischen und deterministischen Selektionsmethoden unterschieden.

In dieser Arbeit werden ausschließlich nicht-duplikatfreie, probabilistische Selektionsmethoden verwendet.

Rekombination Der Rekombinationsoperator entspricht in der Natur der Fortpflanzung von Individuen, bei der meist zwei Eltern einige Nachkommen zeugen. Die Genome der Eltern vermischen sich dabei in einer zu definierenden Weise und ergeben so die Genome der Kinder. Der Rekombination kommt im evolutionären Algorithmus eine Rolle zwischen Variabilität erhalten und Selektionsdruck erzeugen zu.

Der Rekombinationsoperator soll für Gene und analog auch für Individuen definiert werden, um eine leichtere Definition der konkreten Operatoren zu ermöglichen:

Definition (Rekombinationsoperator)

Sei $\xi \in \Xi$ ein Zustand des Zufallsgenerators.

- Sei Γ der genotypische Suchraum eines Optimierungsproblems. Ein Rekombinationsoperator mit $r \geq 2$ Eltern und $s \geq 1$ Kindern ($r, s \in \mathbb{N}$) wird definiert durch die Abbildung
$$R^{\xi} : \Gamma^r \to \Gamma^s.$$
- Seien $I_1, ..., I_r$ (Eltern-) Individuen (für ein $r \geq 2$) und R^{ξ} ein Rekombinationsoperator wie bereits definiert mit $R^{\xi}(I_1.G, ..., I_r.G) = (G_1, ..., G_s)$, sei f die Bewertungsfunktion. Dann ist der zugehörige Rekombinationsoperator für Individuen $\overline{R}^{\xi}$ definiert als
$$\overline{R}^{\xi} : Ind^r \to Ind^s$$
mit
$$\overline{R}^{\xi}(I_1, ..., I_r) = ((G_1, f(G_1)), ..., (G_s, f(G_s))).$$

Mutation Die Mutation dient der Erforschung des Suchraums, indem sie meist geringe zufällige Veränderungen an den Genomen der Population vornimmt und auf diese Weise, stärker noch als die Rekombination neuartige Individuen erzeugt. Sie sorgt für eine Variabilität der Individuen einer Population.

Der Mutationsoperator soll wie der Rekombinationsoperator für Gene und Individuen definiert werden.

Definition (Mutationsoperator)

Sei $\xi \in \Xi$ ein Zustand des Zufallsgenerators.

- Sei Γ der genotypische Suchraum eines Optimierungsproblems. Ein Mutationsoperator wird definiert durch die Abbildung
$$M^{\xi} : \Gamma \to \Gamma.$$
- Sei I ein Individuum und M^{ξ} ein Mutationsoperator wie bereits definiert mit $M^{\xi}(I.G) = G'$, sei f die Bewertungsfunktion. Dann ist der zugehörige Mutationsoperator für Individuen $\overline{M}^{\xi}$ definiert als
$$\overline{M}^{\xi} : Ind \to Ind$$
mit
$$\overline{M}^{\xi}(I) = (G', f(G')).$$

2.2.2 Rahmenbedingungen

Neben der Festlegung einer genotypischen Kodierung und der entsprechenden Dekodierungsfunktion sowie einer Auswahl von Operatoren, müssen bei einem evolutionären Algorithmus

auch die Erzeugung einer Startpopulation und eine Terminierungsbedingung definiert werden. Außerdem gibt es eine Reihe von Parametern wie die Populationsgröße, die angegeben werden müssen. Daraus ergibt sich eine Vielzahl von Stellschrauben, die z. T. bei nur kleinen Änderungen bereits große Sprünge in der Qualität der Ergebnisse bewirken können. Bei der Implementierung eines evolutionären Algorithmus stellt es sich daher als besonders wichtig heraus, eine modulare Programmierweise anzuwenden, die es erlaubt, Operatoren, aber auch Elemente wie die Startpopulationserzeugung oder die Terminierungsbedingung einfach auszutauschen, um Testläufe mit vielerlei verschiedenen Einstellungen vornehmen zu können. Bei der vorliegenden Arbeit wurde bei der Suche nach optimalen Einstellungen neben der Durchführung von Testläufen (deren Anzahl den Rahmen einer Studienarbeit nicht sprengen durfte) auf gängige oder bereits erprobte Operatoren zurückgegriffen und ansonsten versucht, sich an die Richtlinien im Buch von Karsten Weicker gehalten [Weicker2002].

Eine weitere Rahmenbedingung stellt der verwendete Zufallsgenerator dar. Für diese Studienarbeit wurde der Java-interne Zufallsgenerator verwendet. Inwieweit die Qualität der Ergebnisse von der Art des Zufallsgenerators beeinflusst wird und welche Stellung dabei der Java-interne Zufallsgenerator einnimmt, wurde nicht untersucht.

2.2.3 Populationsentwicklung innerhalb eines Zyklus

Dieser Abschnitt ist an das Buch von Volker Nissen [Nissen1997] angelehnt.

Im Rahmen des oben beschriebenen evolutionären Algorithmus ergeben sich eine Vielzahl von Freiheiten bezüglich der Populationsentwicklung innerhalb eines Zyklus. Es wurde weitestgehend versucht, diese Freiheiten bei der Implementierung beizubehalten, so ist es z. B. möglich, im Programm jeden Operator mit unterschiedlichen Parametern für die Eingangs- und Ausgangsgröße der Population zu verwenden.

Die Umgebung für die durchgeführten Testläufe wurde jedoch grundsätzlich auf zwei feste Varianten beschränkt. Diese ergeben sich aus der Wahl zwischen der *Plus-Strategie* und der *Komma-Strategie*.

Komma-Strategie Hier wird innerhalb eines Zyklus die alte Generation nur zur Rekombination herangezogen und danach komplett durch die neuen Individuen ersetzt (ausgenommen davon sind evtl. einige Individuen bei „elitärer“ Komma-Strategie, s. u.).

Plus-Strategie Hier werden auch Individuen aus der alten Generation in die neue Generation übernommen. Dabei wird zunächst eine Population doppelter Größe erzeugt; das sind die ursprünglich als Eltern herangezogenen Individuen vereinigt mit den rekombinierten Kindern. Zum Schluss des Zyklus wird die Population durch die Umweltselektion wieder auf die ursprüngliche Größe verkleinert.

Zusätzlich dazu wurde die Möglichkeit implementiert, eine Auswahl von Individuen als *elitär* zu kennzeichnen (beispielsweise die besten p Prozent). Diese Kennzeichnung bewirkt, dass die betreffenden Individuen garantiert unverändert in die nächste Generation übernommen werden. Beide obigen Varianten wurden sowohl mit als auch ohne elitäre Individuen getestet (siehe Abschnitt 4).

2.3 Lamarcksche Evolution

Bei Lamarckscher Evolution handelt es sich um eine Art der Vererbung, welche die Grenzen zwischen Genotyp und Phänotyp bricht; dem Phänotyp ist es dabei möglich, Fertigkeiten, die er zu Lebzeiten erworben hat, direkt an das Erbgut seiner Nachkommen weiterzugeben. Eine Giraffe ist nach Lamarck beispielsweise zu ihrem langen Hals gekommen, weil sich ihre Vorfahren nach immer höher hängenden Blättern und Zweigen reckten und (durch das Gesetz von Gebrauch und Nichtgebrauch) zum Zeitpunkt ihrer Fortpflanzung einen längeren Hals hatten als bei ihrer Geburt; diesen etwas verlängerten Hals gaben sie an die Nachkommen direkt weiter, sodass im Laufe vieler Generationen ein sehr langer Hals entstand. Diese von Lamarck Anfang des 19. Jahrhunderts vorgeschlagene Vererbungstheorie [Lamarck1809] wurde durch die Arbeit Darwins und die später entstandene Genetik weitgehend abgelöst und gilt heute für die natürliche Evolution als widerlegt.

Während im Allgemeinen also davon ausgegangen wird, dass eine Vererbung erworbener Fertigkeiten in der Natur nicht (oder fast nicht) vorkommt, ist es in einem evolutionären Algorithmus einfach, eine solche Rückkopplung einzubauen und die Genome der Kinder eines Individuums mit dessen phänotypischen Veränderungen auszustatten. Diese Veränderungen sollten natürlich nicht zufällig sein, sondern, dem Lernen in der Natur nachempfunden, zu einer höheren Fitness des Individuums führen. Konkret bedeutet das, dass einem Individuum vor der Rekombination die Möglichkeit gegeben wird, sein Genom zu verändern und das Resultat in der „Umwelt“, also bezüglich der Fitnessfunktion, zu testen. Falls eine Verbesserung erzielt wurde, wird das verbesserte Genom für die Rekombination herangezogen, sonst das alte. Auf diese Weise kann Gelerntes im Lamarckschen Sinne auf Nachkommen übertragen werden. In welcher Weise es einem Individuum ermöglicht werden soll, sein Genom zu verändern, wird in Abschnitt 2.5 diskutiert.

2.4 Der Baldwin-Effekt

Der Baldwin-Effekt ist eine andere, von J. Mark Baldwin beschriebene Auswirkung von erworbenen Fertigkeiten auf das Erbgut einer Spezies, die aber im Gegensatz zur Lamarckschen Evolution die Trennung von Genotyp und Phänotyp aufrechterhält [Baldwin1896]. Nach Baldwin können erworbene Fertigkeiten sich zwar nicht direkt auf die Nachkommen auswirken, aber dennoch über viele Generationen hinweg die Entwicklung der Spezies beeinflussen, so dass schließlich das ursprünglich nur Gelernte auch in das Erbgut übergeht. Baldwin nannte seinen Effekt organische Selektion, weil er es den einzelnen Organismen einer Spezies erlaubt, in gewisser Weise die Zukunft ihrer Spezies indirekt durch ihr Lernverhalten bewusst zu gestalten.

2.4.1 Baldwinsches Lernen

Baldwin definiert Lernen in ähnlicher Weise wie es auch bei evolutionären Algorithmen sinnvoll ist. Er postuliert, dass ein Lebewesen bei seiner Geburt mit einem Satz „potenzieller Fähigkeiten“ ausgestattet ist, derer es sich bedienen kann. Im Lauf seines Lebens wählt es einige vorteilhafte davon aus und trainiert sie, während es andere verkümmern lässt. Diese potenziellen Fähigkeiten können, etwas abstrahiert, als eine Nachbarschaft $N(I)$ eines Individuums I angesehen werden, die aus allen Individuen besteht, zu denen sich I potenziell

entwickeln kann. Die Nachbarschaft ist in diesem Sinn definiert als

$$N(I) =_{def} \{I' | I \text{ kann sich im Lauf des Lebens zu } I' \text{ entwickeln } \}.$$

(Optimales) Lernen ist in dieser Sichtweise die Weiterentwicklung eines Individuums zu dem besten seiner Nachbarn. Dieser Nachbarschaftsbegriff ist stark abhängig von der aktuellen Umwelt und im Allgemeinen wird es einem Organismus nicht gelingen, sich genau zu dem Nachbarn weiterzuentwickeln, der die meisten Nachkommen zeugen wird (und somit definitionsgemäß die beste Fitness hat). In einem evolutionären Algorithmus kann man aber über die feste Fitnessfunktion den besten Nachbarn genau bestimmen. Es fehlt in diesem Bereich nur noch eine Definition der Nachbarschaft eines Individuums – siehe dazu Abschnitt 2.5, um Baldwinsches Lernen in einem evolutionären Algorithmus zu ermöglichen.

2.4.2 Der Übergang in die Gene

Verantwortlich dafür, dass erworbenen Fertigkeiten schließlich in den Genen abgebildet werden können, ist nach Baldwin der Fitnessvorteil, den eine bestimmte Fertigkeit mit sich bringt. Dadurch, dass die Fertigkeit von dem Organismus, der sie erlangt hat, auch seinen Kinder „beigebracht“ wird, und von den Kindern wieder ihren Kindern, wird das, was zunächst ein Vorteil für einige wenige war, nach einigen Generationen zu einem Nachteil für die, die es noch nicht können. Diese Entwicklung ist verlangsamt auch ohne Lehren der älteren Generation denkbar und wurde bei künstlicher Evolution bereits nachgewiesen (s. u.). Der ursprünglich flüchtige Fitnessvorteil wird so zu einer festen Umwelbedingung innerhalb der Population, wodurch jene, welchen es schwerfällt, die Fertigkeit zu erlangen, vorrangig ausselektiert werden. Zwischen denen, die die Fähigkeit erlangen können, ergeben sich weitere qualitative Abstufungen. Im Vorteil ist, wer die Fertigkeit schon möglichst schnell nach der Geburt besitzt, aber auch, wer die Fertigkeit bis zu einem hohen Maß an Perfektion trainieren kann. Nach Baldwin sind diese qualitativen Unterschiede durch den Genotyp des Individuums festgelegt; dadurch wird sich also auf genetischer Ebene Erbgut mit größerer Wahscheinlichkeit durchsetzen, wenn es dem Phänotyp bessere Voraussetzungen für das Erwerben der vorteilhaften Fertigkeit liefert. So kann im Laufe vieler Generationen die Fertigkeit sogar ganz in die Gene einer Spezies übergehen; Individuen sind dann von Geburt an mit der ursprünglich zu erwerbenden Fertigkeit ausgestattet.

Die Frage nach der Existenz des Baldwin-Effekts in der Natur wird bis heute kontrovers diskutiert. Einer der Kritikpunkte an der Theorie, der von Baldwin selbst eingeräumt wurde, ist, dass der Effekt eine über lange Zeit sehr konstante Umwelt voraussetzt (zumindest auf den Vorteil einer speziellen Fertigkeit bezogen). In den meisten Fällen scheint eine allgemeine Flexibilität der Organismen bei Geburt besser zu sein als angeborene feste Regeln, nach denen sie handeln müssen. Bei der Theorie über eine dem Menschen angeborene Sprachfähigkeit wird aber beispielsweise von deren Anhängern der Baldwin-Effekt häufig in der Argumentation verwendet, weil er eine Erklärung dafür liefert, wie die Sprache, die in fast jeder Umwelt einen Vorteil bietet, in die Gene gelangt sein könnte.

Bei künstlicher Evolution im Computer wurde der Effekt tatsächlich schon mehrfach nachgewiesen, das erste Mal von Hinton und Nowlan [Hinton1987]. In ihrem Experiment ließen sie einen evolutionären Algorithmus die Lösung eines extremen Optimierungsproblems suchen, bei dem der Suchraum 2^{20} Elemente hatte, von denen nur ein einziges eine „gute“ Fitness hatte, alle anderen aber eine gleichwertig schlechte. Da es hier keine Abstufungen hin zu irgendwelchen Optima gibt, könnte ein evolutionärer Algorithmus ohne Lernfähigkeit nur

raten und würde etwa 2^{20} Versuche brauchen, um die Lösung zu finden. Durch eine einfache Form von Lernen ohne Rückkopplung auf den Genotyp konnte die Lösung mit bedeutend weniger Versuchen gefunden werden, was als eine Art Baldwin-Effekt interpretiert wurde.

2.4.3 Motivation des Baldwin-Effekts gegenüber Lamarckscher Evolution

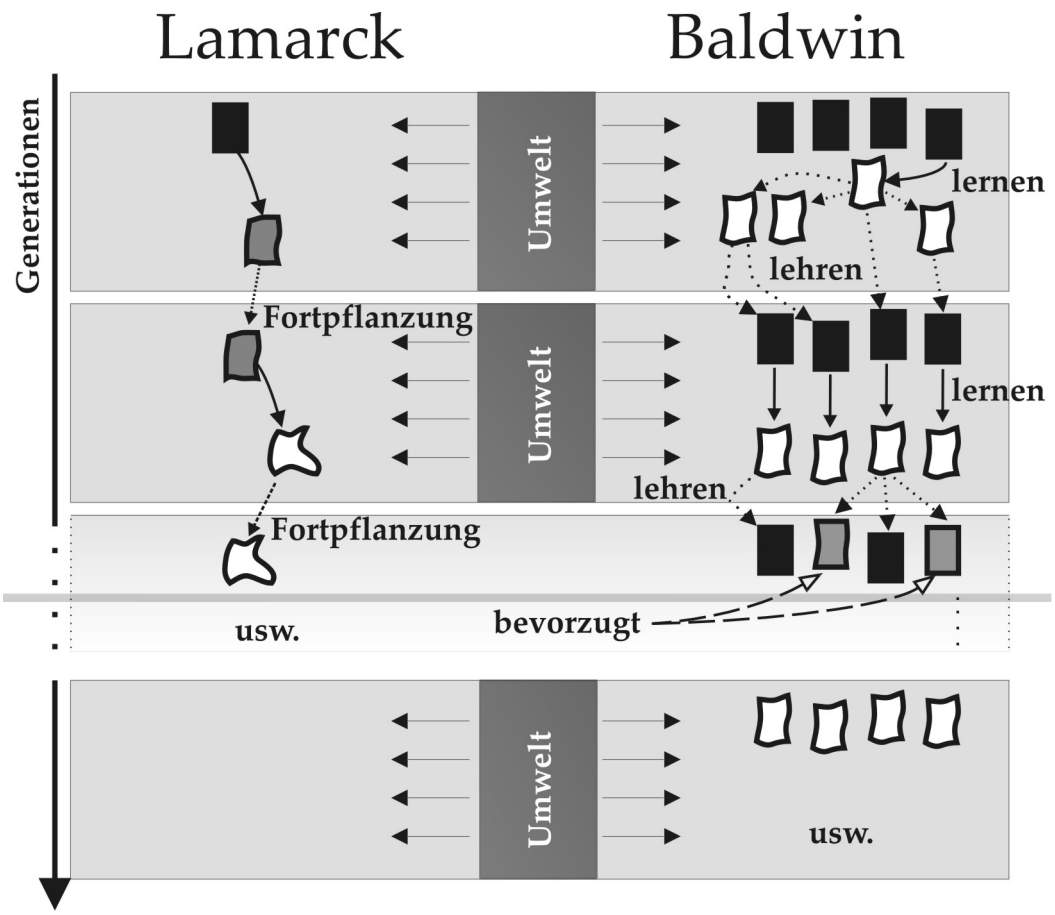

Abbildung 2: *Vergleich zwischen Baldwin und Lamarck: Während bei Lamarck der Genotyp im Lauf des Lebens verändert und somit direkt an die Nachkommen weitergegeben wird, findet bei Baldwin der Übergang in die Gene erst über viele Generationen hinweg durch eine Kombination von Lehren und Lernen von vorteilhaftem Verhalten statt; Gene, die das Lernen dieses Verhaltens begünstigen, werden durch die Selektion bevorzugt.*

Abbildung 2 zeigt die wesentlichen Unterschiede zwischen Lamarckscher Evolution und dem Baldwin-Effekt. In dieser Arbeit wird vor allem der Baldwin-Effekt zur Unterstützung von evolutionären Algorithmen untersucht, die Lamarcksche Evoultion wird nur am Rande behandelt. Intuitiv könnte man jedoch annehmen, dass der Baldwin-Effekt nichts anderes ist als verlangsamte Lamarcksche Evolution. Von diesem Standpunkt aus wäre es in einem evolutionären Algorithmus, wo die Rückkopplung von Phänotyp zu Genotyp kein Problem darstellt, immer sinnvoller, Lamarck statt Baldwin zu benutzen. Gruau und Whitley zeigten jedoch, dass je nach Problem manchmal Lamarcksche Evolution, manchmal aber auch der Baldwin-Effekt sowohl schnellere als auch bessere Lösungen liefern konnte [Gruau1993]. Auf theoretischerer Ebene argumentiert Mayley [Mayley1996], dass Lernen den Vorteil der

Flexibilität hat, während ein zu schneller Übergang in die Gene ein Individuum in einer sich wandelnden Umwelt zu statisch werden lässt. In wechselnden Umwelten (die es auch bei evolutionären Algorithmen geben kann – wenn auch nicht im vorliegenden Fall) könnte sich dies als ein Vorteil des Baldwin-Effekts herausstellen. Schließlich wurde von Whitley, Gordon und Mathias eine Untersuchung der Optimierung von stetigen Funktionen unter Zuhilfenahme des Baldwin-Effekts und von Lamarckscher Evolution vorgenommen, wie sie bei bestimmten physikalischen Optimierungsproblemen vorkommen [Whitley1994]. Dabei konnte ebenfalls gezeigt werden, dass der Baldwin-Effekt unter Umständen schneller zu besseren Ergebnissen kommt als Lamarcksche Evolution.

Für letztere Arbeit wurde unter anderem bei stetigen Funktionen eine Visualisierung der suchraumverändernden Wirkung durch den Baldwin-Effekt durchgeführt. Eine solche Visualisierung von stetigen Funktionen mit zuschaltbarem Baldwin-Effekt wurde auch im Vorfeld dieser Studienarbeit zur Einarbeitung in das Thema implementiert (siehe Abbildung 3).

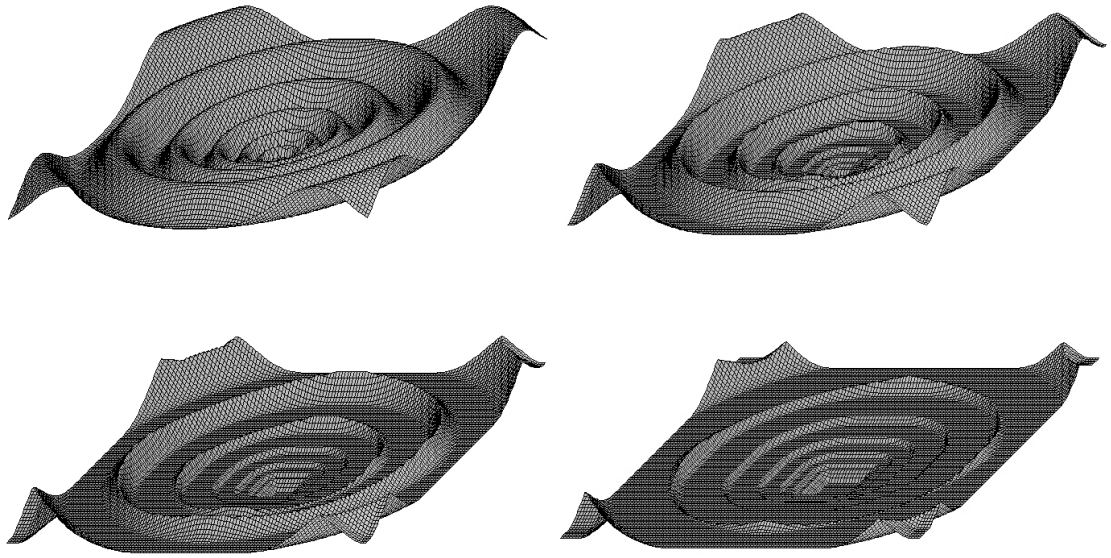

Abbildung 3: *Beispiel einer Visualisierung von Baldwin-Lernen bei stetigen Suchräumen – Entwicklung des durch die zweidimensionale Schaffer2-Funktion gegebenen stetigen Suchraums (l. o.) beim Anwenden von Baldwinschem Lernen auf 9 Längeneinheiten (r. o.), 18 LE (l. u.) und 30 LE (r. u.) von jedem Bildpunkt aus in beiden Dimensionen. Aus den schwer überbrückbaren lokalen Minima entstehen Plateaus, die von einem evolutionären Algorithmus leichter überquert werden können. Die Schaffer2-Funktion ist definiert als* $F(x,y) = \left(x^2+y^2\right)^{0,25} \cdot \left[sin^2\left(50 \cdot \left(x^2+y^2\right)^{0,1}\right) + 1,0\right]$.

Neben einer Erforschung der Wirkung des Baldwin-Effekts durch Testläufe ist ein wesentlicher Teil dieser Studienarbeit eine Visualisierung der suchraumverändernden Wirkung des Baldwin-Effekts bei diskreten Problemen, insbesondere dem SCP. Solche Probleme haben keine stetigen Funktionen als Suchräume sondern lassen sich nur über eine Graphstruktur darstellen, wodurch die Visualisierung wesentlich erschwert wird. Das Ziel ist es, einen besseren Einblick in die Struktur von Suchräumen zu gewinnen und den Einfluss von Lernen auf die evolutionäre Entwicklung genauer zu studieren.

2.5 Die Nachbarschaft eines Individuums

Wie bereits in Abschnitt 2.4.1 beschrieben, definierte Baldwin das Lernen zu Lebzeiten eines Organismus als Weiterentwicklung zu einem fortpflanzungsfähigeren Organismus unter Ausnutzung der angeborenen potenziellen Fähigkeiten. Optimales Lernen wäre in diesem Sinne die Entwicklung zu dem allerbesten (fortpflanzungsfähigsten) Organismus, der aus den angeborenen Fähigkeiten „entstehen“ kann. In der Natur hat ein Organismus keinen Überblick über alle Organismen, zu denen er sich potenziell weiterentwickeln kann (seine Nachbarn), und er kann auch nicht genau abschätzen, wie fortpflanzungsfähig die verschiedenen Ausprägungen seiner selbst sein werden; deshalb ist hier kein optimales Lernen möglich.

In einem evolutionären Algorithmus kann die Nachbarschaft eines Individuums dagegen eindeutig und eigentlich beliebig definiert werden (genauer wird im folgenden die Nachbarschaft auf dem genotypischen Suchraum Γ definiert). Man kann prinzipiell als Nachbarschaft eines Genoms jede Teilmenge der gültigen Genome benutzen. Die Nachbarschaft ist also allgemein eine Funktion $N : \Gamma \to \mathcal{P}(\Gamma)$. Vorausgesetzt, es sei bereits eine Nachbarschaft definiert, kann man (optimales) Lernen im Baldwinschen Sinn definieren. Da sich Lernen nur auf die Fitness $I.F$ eines Individuums I auswirkt (und insbesondere nicht auf sein Genom $I.G$), kann Lernen als eine neue Fitnessfunktion aufgefasst werden, die sich aus der ursprünglichen Fitnessfunktion des Optimierungsproblems ergibt.

Definition (Gelernte Fitness)

> Sei Γ der genotypische Suchraum, $N : \Gamma \to \mathcal{P}(\Gamma)$ eine Nachbarschaft und f die Fitnessfunktion des zu lösenden Optimierungsproblems.
>
> Für ein Genom G ist die gelernte Fitness $f_N : \Gamma \to \mathbb{R}$ definiert als
>
> $$f_N(G) = \max(\{f(G')|G' \in N(G)\} \cup \{f(G)\})$$
>
> Dabei sei die Funktion max relativ zur Ordnung $\succ$ des Optimierungsproblems definiert. (Für $i \in \{1, ..., n\}$ gilt: $\max(\{a_1, ..., a_n\}) = a_i \Leftrightarrow_{def} \forall j \in \{1, ..., n\} : a_i \succ a_j$. Bei Minimierungsproblemen ist max also die üblicherweise als min geschriebene Funktion.)

Da ein Genom definitionsgemäß nicht unbedingt in seiner eigenen Nachbarschaft liegen muss, darf das Individuum die beste Fitness aus seiner Nachbarschaft vereinigt mit ihm selbst wählen. Das erscheint plausibel (warum sollte ein Individuum gezwungen werden sich zu verschlechtern, nur weil alle Nachbarn schlechter sind) und ist auch in der Praxis sinnvoll, weil die eigene Fitness sehr schnell berechnet werden kann und somit die Abfrage kaum zusätzliche Kosten verursacht.

Nun fehlt noch die konkrete Definition der Nachbarschaft, die im folgenden immer über den Mutationsoperator gegeben sein wird:

Definition (Nachbarschaft)

Sei M ein Mutationsoperator und Ξ der Raum des Zufallsgenerators.

- Sei Γ der genotypische Suchraum. Die Nachbarschaft eines Genoms G bezüglich des Mutationsoperators M ist eine Funktion $N_M : \Gamma \rightarrow \mathcal{P}(\Gamma)$ mit

$$N_M(G) = \{G' \mid \exists \xi \in \Xi : M^\xi(G) = G'\}$$

- Die Nachbarschaft eines Individuums I bezüglich des zu M gehörigen Mutationsoperators auf Individuen $\overline{M}$ ist dann eine Funktion $\overline{N}_M : Ind \rightarrow \mathcal{P}(Ind)$ mit

$$\overline{N}_M(I) = \{I' \mid \exists \xi \in \Xi : \overline{M}^\xi(I) = I'\}$$

Falls klar (oder unwichtig) ist, welche Mutation der Nachbarschaft zugrundeliegt, kann statt N_M bzw. $\overline{N}_M$ auch N bzw. $\overline{N}$ geschrieben werden.

Anhand einer Nachbarschaft kann in naheliegender Weise ein Graph aufgebaut werden, indem die Knoten Individuen repräsentieren und eine Kante die Bedeutung „ist Nachbar von“ erhält.

Definition (Nachbarschaftsgraph)

Sei Ind die Menge der Individuen und $\overline{N} : Ind \rightarrow \mathcal{P}(Ind)$ eine Nachbarschaft auf Individuen.

Der Nachbarschaftsgraph ist ein gerichteter Graph (V, E) mit

- Knotenmenge $V = Ind$ und
- Kantenmenge $E = \{(I_1, I_2) \in (V \times V) | I_2 \in \overline{N}(I_1)\}$

2.6 Das Mengenüberdeckungsproblem (SCP)

Das Mengenüberdeckungsproblem (engl. Set Covering Problem) ist ein Optimierungsproblem, bei dem nach einer kostenminimalen Überdeckung einer Menge durch eine Auswahl ihrer Teilmengen gefragt wird. Es werden die drei Varianten unicost, multicost und generalcost unterschieden.

Bei jeder Variante ist eine Menge M und eine Menge $T \subseteq \mathcal{P}(M)$ von Teilmengen von M gegeben. Zusätzlich gibt es eine Kostenfunktion κ, die Teilmengen der Menge T Kosten zuordnet; die Kostenfunktion ist also auf Elementen der Menge $\mathcal{P}(\mathcal{P}(M))$ definiert. Je nach Variante des SCP darf κ unterschiedlich komplex sein. Die Aufgabe ist es, eine Teilmenge $T' \subseteq T$ zu finden, so dass die Vereinigung der Mengen in T' die Menge M ergibt und die Kosten $\kappa(T')$ minimal sind.

Bezeichnungen

Die Elemente der zu überdeckenden Menge M heißen *Indizes* und werden o.B.d.A mit $1, 2, 3, ...$ bezeichnet.

Teilmengen von M heißen dementsprechend *Indexteilmengen*; das sind insbesondere die Elemente von T.

Teilmengen der Menge T (also auch T selbst) sind Mengen von Indexteilmengen und werden deshalb *Indexteilmengen-Familien* oder Itm-Familien genannt.

Definition (Das Optimierungsproblem SCP)

Sei M eine (endliche) Menge, $T \subseteq \mathcal{P}(M)$. Sei $\kappa : \mathcal{P}(T) \to \mathbb{R}$ eine (totale) Kostenfunktion.

Das Optimierungsproblem SCP besteht aus

- dem Suchraum $\Omega = \{T' | T' \in \mathcal{P}(T) \text{ und } \bigcup_{t \in T'} t = M\}$,
- der Bewertungsfunktion κ und
- der Vergleichsrelation $\succ = <$ (SCP ist also ein Minimierungsproblem).

Eine Instanz des SCP wird durch ein Tupel (M, T, κ) angegeben.

Varianten: In allen drei hier betrachteten Varianten ist die Kostenfunktion κ grundsätzlich auf Funktionen beschränkt, die folgende Monotoniebedingung erfüllen: Seien $T_1, T_2 \subseteq T$, dann gilt

$$|T_1| < |T_2| \Rightarrow \kappa(T_1) \leq \kappa(T_2).$$

(Würde auch diese Einschränkung fallen gelassen, verhielte sich der Suchraum so zufällig, dass die hier angewandten Algorithmen, die möglichst angenähert „stetige“ Suchräume voraussetzen, vermutlich keine sinnvollen Ergebnisse erzielen würden.)

Bei *generalcost* ist außer der Monotonie keine weitere Einschränkung gegeben.

Bei *multicost* wird zusätzlich gefordert, dass κ „linear“ ist. Es soll also für alle Teilmengen $T_1, T_2 \subseteq T$ gelten:

$$\kappa(T_1) + \kappa(T_2) = \kappa(T_1 \cup T_2).$$

Bei *unicost* wird zusätzlich zu den beiden obigen Bedingungen gefordert, dass κ für alle Itm-Familien, die nur eine Indexteilmenge enthalten, konstant ist, d. h. es existiert ein c, so dass für alle $t \in T$ gilt:

$$\kappa(\{t\}) = c.$$

Das Finden einer Lösung für das SCP ist bereits in der unicost-Variante NP-hart [Karp1972].

Beispiel (Ein multicost-SCP)

Gegeben seien $M = \{1, 2, 3, 4\}$ und $T = \{t_1, t_2, t_3, t_4\}$ mit

$t_1 = \{1, 2\}$, $\kappa(\{t_1\}) = 1$,

$t_2 = \{1\}, \kappa(\{t_2\}) = 3,$

$t_3 = \{2, 3, 4\}, \kappa(\{t_3\}) = 2,$

$t_4 = \{3, 4\}, \kappa(\{t_4\}) = 2.$

(Für alle mehr als einelementigen Teilmengen von T, deren Kosten dadurch noch nicht definiert sind, seien die Kosten gemäß der Forderung für multicost die Summe der Kosten der Einzelteilmengen.)

In diesem Beispiel ist $T' = \{t_1, t_4\}$ eine Lösung, aber auch $T'' = \{t_1, t_3\}$, denn beide überdecken die gesamte Menge M und die Kosten $\kappa(T') = \kappa(T'') = 3$ werden von keiner anderen überdeckenden Kombination von Teilmengen unterschritten.

Wie in dem Beispiel, soll es im folgenden bei den Varianten unicost und multicost genügen, die Kosten nur für die einelementigen Teilmengen von T anzugeben. Für alle Teilmengen $T_{>1} \subseteq T$, die mehr als ein Element enthalten, gelte dann implizit: $T_{>1} = \{t_1, ..., t_n\}$ $(n > 1)$ $\Rightarrow \kappa(T_{>1}) = \kappa(t_1) + ... + \kappa(t_n)$.

Obwohl die in dieser Arbeit beschriebenen Programme prinzipiell generalcost-SCP bearbeiten können, wurden die Testläufe des evolutionären Algorithmus auf die Varianten unicost und multicost beschränkt. Die Gründe dafür sind, dass einerseits schon multicost-Probleme so schwierig sein können, dass sie den Algorithmus in seiner jetzigen Form an seine Grenzen bringen, und dass andererseits zum Ende hin die Zeit fehlte, noch weitere Tests durchzuführen. Ansätze zur Verbesserung des Algorithmus werden in Abschnitt 3.4 diskutiert.

Um verschiedene Instanzen des SCP besser miteinander vergleichen zu können, sollen einige typische Größen mit Bezeichnungen versehen werden.

Sei (M, T, κ) eine Instanz des SCP, dann gelten folgende Bezeichnungen:

Bezeichnungen

Die *(Problem-)Größe* ist die Anzahl der Index-Teilmengen $|T|$.

Die *Indexgröße* ist die Anzahl der Indizes $|M|$.

Die *(durchschnittliche) Überdeckung eines Index* ist gegeben durch:

$$\frac{\sum_{t \in T} |t|}{|M|}.$$

Die *(durchschnittliche) Größe der Index-Teilmengen* ist gegeben durch:

$$\frac{\sum_{t \in T} |t|}{|T|}.$$

Die *(durchschnittliche) Dichte* ist gegeben durch:

$$\frac{\sum_{t \in T} |t|}{|M| \cdot |T|}.$$

2.7 Ein vorhandenes Framework für das SCP

Der in dieser Arbeit verwendete evolutionäre Algorithmus ist in Java geschrieben und wurde grundsätzlich selbst entworfen. Der Teil des Algorithmus, der auf dem phänotypischen Suchraum Ω arbeitet, also auf den dekodierten Problemlösungsinstanzen (Itm-Familien), wurde aber der Funktionalität des Frameworks von Dietmar Lippold [Lippold2006] entnommen, das dieser im Zuge seines Promotionsvorhabens entworfen hat. Das Framework beinhaltet zahlreiche Funktionen für das Arbeiten mit SCP-Instanzen und einige Algorithmen für deren Lösung. Die folgenden Teile wurden verwendet:

- Die Startklasse implementiert *ein Interface des Frameworks*, sodass der evolutionäre Algorithmus innerhalb des Frameworks als ein Lösungsalgorithmus für das SCP betrachtet und verwendet werden kann. In diesem Sinn ist er eine Erweiterung des Frameworks.
- Vor dem Start des eigentlichen evolutionären Algorithmus wird eine Referenzlösung mit dem *Iterated-Enhenced-Greedy-Verfahren* des Frameworks erzeugt. Diese wird zurzeit nur für die Steuerung der Terminierungsbedingung eingesetzt (falls der evolutionäre Zyklus eine Lösung findet, die mindestens so gut ist wie die Referenzlösung, werden nur noch weitere k Generationen durchgeführt). Es ist aber durchaus denkbar, diese Referenzlösung auch bei der Erzeugung der Startpopulation zu verwenden, wodurch sich ein Hybridverfahren ergeben würde.
- Die Methoden zur *Erzeugung von zufälligen Probleminstanzen*, sowie zum *Einlesen von in Dateien gespeicherter Probleminstanzen* wurden für das Durchführen von Tests verwendet.
- Die *Datenstruktur für Probleminstanzen* wurde benutzt. Dazu gehört auch deren Erzeugung sowie das Hinzufügen und Entfernen von Indexteilmengen.
- Die Methode zur *Berechnung der Kosten κ einer Lösungsinstanz* wurde verwendet.
- Die Methoden zur *Abfrage, wie viele Indizes noch nicht überdeckt sind* sowie, *wie viele Teilmengen nur Elemente abdecken, die auch von anderen Teilmengen überdeckt werden*, wurden benutzt.

Die Verwendung des Frameworks brachte neben vielen Vorteilen auch einige Nachteile mit sich. Der bedeutendste war der, dass das Framework hinsichtlich der Datenstruktur für die bereits implementierten Algorithmen optimiert war; bei diesen wurde jeweils nur eine Lösungsinstanz erzeugt und dann so lange verändert, bis eine akzeptable Lösung herauskam, weshalb es kein Problem darstellte, dass das Erzeugen der Lösungsinstanz relativ lange dauerte. Ein evolutionärer Algorithmus erzeugt hingegen eine große Menge von Lösungsinstanzen. Eine anfangs allzu geradlienige Vorgehensweise führte daher zu schlechtem Laufzeitverhalten. Das Problem konnte dadurch vermindert werden, dass statt einer wiederholten Neuerzeugung von Instanzen die alten wiederverwendet wurden. Siehe zur Implementierung des evolutionären Algorithmus Abschnitt 3.

2.8 Graphvisualisierung

Zur Visualisierung der suchraumverändernden Wirkung des Baldwin-Effekts wird durch die Vorgehensweise in dieser Arbeit eine Visualisierung des Nachbarschaftsgraphen notwendig. Da ein Graph in der Ebene (oder im Raum) unendlich viele Darstellungen besitzt, steht man prinzipiell vor dem Problem, verschiedene Kriterien „ästhetischer“, d. h. die Verständlichkeit erhöhender Darstellungsrichtlinien gegeneinander abwägen zu müssen. Solche Richtlinien sind beispielsweise die Hervorhebung symmetrischer Eigenschaften, die Vermeidung von Kreuzungen der Kanten (im Raum kann jeder Graph völlig kreuzungsfrei dargestellt werden) oder die Darstellung von Kanten als gerade Linien. Viele dieser Kriterien widersprechen einander oder die Berechnung ihrer Darstellung ist NP-vollständig. Die Arbeit von Batista et al. [Batista1994] bietet eine umfangreiche Liste ästhetischer und anderer Kriterien zur Graphdarstellung, sowie zu jedem Kriterium eine Auflistung wichtiger Arbeiten; Lipton et al. behandeln in ihrer Arbeit [Lipton1985] die Wichtigkeit von Symmetrie bei der Darstellung von Graphen in der Ebene und geben einen mathematischen Hintergrund für ihre algorithmische Erzeugung.

Weil die umfassende Beschäftigung mit ästhetischer Darstellung von Graphen, die bis in die Bereiche der Ergonomie, aber auch die der Komplexitätstheorie reicht, den Rahmen einer Studienarbeit gesprengt hätte, wurde die vereinfachende Vereinbarung getroffen, dass Knoten mehrfach angezeigt werden dürfen. Während sie in der internen Graphstruktur nur einfach vorkommen können, werden sie u. U. mehrfach dargestellt, bspw. wenn ein Individuum sein eigener Nachbar ist. Auf diese Weise wurden viele der oben angesprochenen Probleme umgangen, während die Darstellung dennoch übersichtlich gehalten werden konnte.

Zusätzlich zur Fülle allgemeiner ästhetischer Aspekte kommen bei dieser speziellen Visualisierungsaufgabe einige problemspezifische hinzu. Ein besonders wichtiger Aspekt ist die übersichtliche und intuitive Darstellung der Fitness von Knoten (Individuen). Da das Lernen nach Baldwin nur auf die Fitness, nicht aber auf das Genom eines Individuums wirkt, verändert es die Topologie des Nachbarschaftsgraphen nicht, die Veränderung des Suchraums beschränkt sich auf eine Veränderung der Fitness von Knoten. Unabhängig von der geometrischen Darstellung (drei verschiedene Darstellungen wurden implementiert) wurde deshalb hierbei die Bedingung gestellt, dass die Fitness farblich an den Knoten gekennzeichnet sein muss. Ein weiteres erfolgversprechendes Konzept, das aber nicht weiter verfolgt wurde, ist die dreidimensionale Darstellung. Im Prinzip kann jede zweidimensionale Darstellung dazu erweitert werden, dass in der dritten Dimension die Fitness angezeigt wird, bspw. durch die „Höhe“ oder „Tiefe“ der Knoten. Durch Drehen der Ansicht kann dann die Fitness als Gebirge betrachtet werden. Bei zwei der implementierten Darstellungsarten wird die Fitness auch in der Ebene durch die Höhe der Knoten angezeigt, was eine Anzeige in der dritten Dimension überflüssig macht. Die dritte (radiale) Darstellung könnte aber durch solch eine Anzeige durchaus an Übersichtlichkeit gewinnen. Eine entsprechende Ergänzung sollte bei der aktuellen Implementierung ohne großen Aufwand möglich sein, wurde aber aus Gründen der Zeitknappheit nicht durchgeführt.

Ein weiterer Aspekt ist die Frage nach Größe und Art des Auschnitts des Nachbarschaftsgraphen, der angezeigt werden soll. Dass eine Darstellung des gesamten Graphen aller möglicher Individuen schon bei kleinen Individuenräumen nicht möglich ist, scheint klar. Auch eine Population, die in einem evolutionären Zyklus entstanden ist, mit den darin auftretenden Nachbarschaftsbeziehungen darzustellen ist fragwürdig, weil sich gewöhnlich nur wenige Individuen der Population in einer direkten Nachbarschaftsbeziehung befinden – solche In-

dividuen sind schließlich durch Mutation *und* Rekombination aus anderen entstanden. Hier wurde die Entscheidung gefällt, die Darstellung auf ein einziges Individuum einer Population und dessen gesamte Nachbarschaft zu beschränken. Der dargestellte Graph entspricht also dem Teil des Suchraums, den die Evolution, von einem bestimmten Individuum ausgehend, durch reine Mutation erreichen kann. Dabei wird auch dieser Teil des Suchraums nicht auf einmal komplett dargestellt, durch Klicken auf die Knoten kann aber (unabhängig von der Darstellungsart) ein neuer Knoten als Ausgangsknoten bestimmt werden; auf diese Weise kann prinzipiell der komplette (durch Mutation erreichbare) Suchraum betrachtet werden.

Die genaue Beschreibung der implementierten Darstellungsarten erfolgt in Abschnitt 5.3.

2.9 Statistik

Bei der Auswertung der Testläufe werden einige statistische Hilfsmittel benötigt. Die statistischen Formeln und Verfahren wurden dem Buch von Jürgen Bortz [Bortz1999] entnommen.

Standardabweichung: Für die Berechnung der Standardabweichung s eines Tupels von Messwerten $X = (x_1, ..., x_n)$ wird die folgende Formel verwendet:

$$s(X) =_{def} \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n}}.$$

Dabei ist $\overline{x}$ das arithmetische Mittel aller Werte in X.

Studentscher t-Test: Aufbauend auf der Standardabweichung wird der Studentsche t-Test verwendet, um eine vermutete sinifikante Unterscheidung zweier Messreihen zu bestätigen. Dabei wird das im o. a. Buch beschriebene Verfahren zur Berechnung eines *zweiseitigen t-Tests für unabhängige Intervalldaten* benutzt. Der Einfachheit halber wurde für die tatsächliche Berechnung des t-Tests die entsprechende Funktion von Microsoft Excel 2000 (Version 9.0.2182) verwendet. Als Signifikanzniveaus werden, wie für zweiseitige t-Tests empfohlen, 2,5% für signifikante Abweichung und 0,5% für sehr signifikante Abweichung angenommen.

3 Ein evolutionärer Algorithmus für das SCP

Der erarbeitete Algorithmus für das SCP hält sich im Wesentlichen an den in Abschnitt 2.2 beschriebenen Zyklus. Es folgt hier zunächst eine theoretische Beschreibung der Eigenschaften des Algorithmus und aller implementierten Operatoren und dann eine Beschreibung der konkreten Implementierung in Java.

3.1 Theoretische Beschreibung

3.1.1 Die Kodierung des Genotyps

Ein Genom, das eine Lösungsinstanz des SCP repräsentiert, sollte Mutation und Rekombination erlauben, die in gewisser Weise stetig sind, d. h. kleine Veränderungen am Genom sollten

auch zu einer „kleinen“ Verschiebung im Suchraum führen. Außerdem sollte eine effiziente Umwandlung zwischen Genotyp und Phänotyp möglich sein. Eine naheliegende Kodierung, die diese Bedingungen erfüllt, ist ein Bitstring, dessen Länge der Anzahl der Teilmengen des SCP entspricht. Das i-te Bit ist dabei genau dann 1, wenn die zu i gehörige Teilmenge $t_i \in T$ zur Lösung gehört. Zu diesem Zweck müssen die Teilmengen aus T vorher durchnummeriert werden.

Definition (Dekodierungsfunktion *dec* für das SCP)

Sei (M, T, κ) eine Instanz des SCP, wobei $T = \{t_1, t_2, ..., t_{|T|}\}$.

Dann ist die Dekodierungsfunktion $dec : \{0,1\}^{|T|} \to \Omega$ definiert durch

$$dec(G) = \{t_i \in T \mid i \in \{1, ..., |T|\} \wedge G[i] = 1\}$$

für ein $G \in \{0,1\}^{|T|}$.

($G[i]$ bezeichnet das i-te Bit des Bitstrings G.)

Umgekehrt sei die Kodierungsfunktion *cod* definiert:

Definition (Kodierungsfunktion *cod* für das SCP)

Sei (M, T, κ) eine Instanz des SCP, $T = \{t_1, t_2, ..., t_{|T|}\}$ und $T' \subseteq T$.

Dann ist die Kodierungsfunktion $cod : \Omega \to \{0,1\}^{|T|}$ definiert durch

$$cod(T') = G$$

mit

$$G[i] = 1 \Leftrightarrow t_i \in T' \; \forall i \in \{1, ..., |T|\}.$$

Auf diese Weise definierte Genome können zunächst eine beliebige Sequenz von Nullen und Einsen sein. Darunter sind aber viele, die keine gültige Lösung repräsentieren, weil sie nicht alle Indizes überdecken (insbesondere sind diese nach der Definition des Optimierungsproblems SCP auf Seite 24 außerhalb des Suchraums). Andererseits ergeben sich auch viele Lösungsinstanzen, die durch triviales Entfernen von Indexteilmengen weniger kosten, aber dennoch alle Elemente überdecken würden (diese befinden sich zwar innerhalb des Suchraums, aber stellen keine sinnvollen Lösungen dar). Zur Bewältigung dieser beiden Probleme ergeben sich zwei prinzipielle Vorgehensweisen. Die eine ist, alle solchen Lösungen zuzulassen und durch eine „Fitnessstrafe“ die Regulierung dem Algorithmus selbst zu überlassen; dafür wäre eine Abänderung der Funktion κ und des Suchraums Ω notwendig. Die andere ist, solche Lösungen als „ungültig“ zu deklarieren und im Algorithmus nach jeder Mutation und Rekombination eine Reparatur vorzunehmen, die eventuell ungültig gewordene Indviduen wieder gültig macht. Es wurde hier die zweite Variante gewählt, weil einige Tests mit der ersten Variante keine guten Ergebnisse brachten; es ist u. a. nicht trivial zu entscheiden, wie die Fitnessstrafe für Individuen, die nicht alle Indizes überdecken, aussehen soll. Auch die Arbeit von Beasley et al. kam in diesem Punkt zum gleichen Ergebnis [Beasley1994]. (Zu beachten ist, dass der auf Seite 24 definierte Suchraum dadurch um diejenigen Individuen beschnitten wird, die trivial entfernbare Indexteilmengen enthalten. Dennoch soll

die allgemeine Definition des Suchraums gültig bleiben, weil die für die Testläufe benutzte Reparaturmethode nicht die einzige denkbare ist und Raum für andere Implementierungen bleiben soll.)

Der (eingeschränkte) genotypische Suchraum ist also folgendermaßen definiert:

Definition (Genotypischer Suchraum)

Sei (M, T, κ) eine Instanz des SCP.

Der genotypische Suchraum Γ besteht aus allen Genomen G, die folgende Eigenschaften erfüllen:

- $G \in \{0,1\}^{|T|}$,
- $\bigcup_{t \in dec(G)} t = M$ und
- $\forall t' \in dec(G) : \bigcup_{t \in dec(G) \setminus t'} t \subsetneq M$.

Daraus ergibt sich, dass der phänotypische Suchram Ω eventuell nicht komplett abgedeckt wird; der Teil des Suchraums, der abgedeckt wird, soll Ω' heißen und alle Itm-Familien enthalten, die sich aus Genomen des genotypischen Suchraums durch die Funktion *dec* ergeben:

$$\Omega' \;=\; \{F \in \Omega \,|\, \exists\; G \;\in\; \Gamma \;:\; F \;=\; dec(G)\} \;\subseteq\; \Omega.$$

Beispiel (Genotyp)

Seien $M = \{1,2,3,4\}$ und $T = \{t_1, t_2, t_3, t_4, t_5\}$ die gegebenen Mengen einer SCP-Instanz mit

$t_1 = \{1,2\}$, $t_2 = \{2,3\}$, $t_3 = \{3,4\}$, $t_4 = \{1,2,3\}$, $t_5 = \{2,4\}$.

Es gelte für alle $t_i \in T : \kappa(\{t_i\}) = 1$, es handelt sich also um ein Unicost-SCP.

Man kann dieses Problem übersichtlich als Boolesche (5×4)-Matrix darstellen, die an der Stelle (i,j) genau dann *true* ist, wenn der Index j in der i-ten Teilmenge Enthalten ist (*true* soll durch $\times$, $false$ durch ein leeres Feld symbolisiert werden):

$$\begin{array}{cc} & M = \{ \quad 1, \quad 2, \quad 3, \quad 4 \quad \} \\ 1 & t_1 \quad \left(\begin{array}{cccc} \times & \times & & \\ & \times & \times & \\ & & \times & \times \\ \times & \times & \times & \\ & \times & & \times \end{array} \right) \\ 0 & t_2 \\ 1 & t_3 \\ 0 & t_4 \\ 0 & t_5 \end{array}$$

Die Lösungsinstanz $\{t_1, t_3\}$ wird nun bspw. durch den Bitstring 10100 kodiert. Dieser stellt auch ein gültiges Genom dar.

Sowohl Bitstrings wie 11100, 10110, usw. als auch solche wie 00001, 10000, usw. sind dagegen keine gültigen Genome, weil die einen unnötige Indexteilmengen enthalten, und die anderen nicht alle Indizes überdecken.

3.1.2 Operatoren und Rahmenbedingungen

Zu beachten ist, dass nach jeder Mutation und Rekombination eine zufällige Reparatur stattfindet, um gültige Individuen wie oben definiert zu erhalten (bei der Visualisierung ist diese Reparatur deterministisch). Diese nachträgliche Reparatur soll implizit Teil jedes Mutations- und Rekombinationsoperators sein. Der Nachbarschaftsbegriff wird dadurch etwas verwischt, weil nicht unmittelbar klar ist, welche Nachbarn sich durch die jeweilige Mutation ergeben können.

Selektion Es wurden zwei Selektionsmethoden implementiert, die beide probabilistisch und nicht duplikatfrei sind. Die eine wurde als Elternselektion, die andere als Umweltselektion verwendet. Sie wurden in den Testläufen nicht gegeneinander ausgetauscht. Beide wurden aus dem Buch von Karsten Weicker [Weicker2002] übernommen. Die genauen Implementierungen sind in Abschnitt 3.3 beschrieben.

Variante a: Als Elternselektion wurde eine *k-fache Trunierselektion verwendet.* Dabei werden zufällig gleichverteilt k Individuen ausgewählt, die ein Turnier austragen. Das Individuum mit der besten Fitness gewinnt und wird selektiert. Dieser Vorgang wird solange iteriert, bis die gewünschte Anzahl an Individuen selektiert wurde. Bei den Testläufen wurde die Turniergröße jeweils auf 5% der Populationsgröße festgesetzt.

Variante b: Als Umweltselektion wurde eine *lineare rangbasierte Selektion* verwendet. Dafür wird jedem Individuum eine Selektionswahrscheinlichkeit zugewiesen, die nur davon abhängt, welchen Rang das Individuum in der Population hat. Vom besten zum schlechtesten Individuum nimmt die Wahrscheinlichkeit dabei linear ab. Wenn $A^{(1)}, ..., A^{(r)}$ die Individuen der Population in sortierter Reihenfolge sind, d. h. $A^{(i)}$ bezeichnet das i-t beste Individuum und die Größe der Population ist $r > 1$, dann werden die Wahrscheinlichkeiten durch folgende Formel zugewiesen:

$$Pr[A^{(i)}] = \frac{2}{r}\left(1 - \frac{i-1}{r-1}\right).$$

Wie man leicht nachrechnen kann, gilt für alle r: $\sum_{\forall i} Pr[A^{(i)}] = 1$.

Rekombination Da im vorliegenden Fall die Bits im Genom der Individuen voneinander „relativ“ unabhängig sind (d. h. es können Bits beliebig geflippt werden, ohne eine unbrauchbare Lösung zu erhalten; möglicherweise muss die Lösung allerdings danach noch repariert werden; außerdem bedeutet ein Bitflip normalerweise keinen großen Sprung im Suchraum) wurden sie sowohl bei der Rekombination als auch bei der Mutation weitgehend einzeln betrachtet. Für die Rekombination wurden zwei einander sehr ähnliche Rekombinationsmethoden benutzt. Beide haben folgende Eigenschaften:

- Es wird immer ein Kindindividuum K aus zwei Elternindividuen E_1, E_2 erzeugt.
- Für jede Stelle im Genom, an der beide Eltern das gleiche Bit haben, erhält das Kind genau dieses Bit.

Der Unterschied besteht in dem Fall, dass sich die Genome von den Elternteilen an einer Bitstelle unterscheiden. Sei i eine solche Stelle. Bei Variante a) wird in diesem Fall gleichverteilt zufällig das Bit $K[i]$ des Kindindividuums auf 1 oder 0 gesetzt. Bei Variante b) wird

eine fitnessproportionale Wahrscheinlichkeitsverteilung für die beiden möglichen Wahrheitswerte benutzt, wie sie in der Arbeit von Beasley et al. [Beasley1994] vorgeschlagen wird. Das Kindindividuum wird dabei an der Stelle i mit der Wahrscheinlichkeit $w_1 = \frac{E_2.F}{E_1.F+E_2.F}$ auf das Bit $E_1[i]$ des ersten Elternindividuums gesetzt und mit der Wahrscheinlichkeit $1 - w_1$ auf das des zweiten Elternindividuums.

Variante a: Der gleichverteilte Rekombinationsoperator ist folgendermaßen definiert:

Definition (Gleichverteilter Rekombinationsoperator)

Der gleichverteilte Rekombinationsoperator R_a^ξ ist ein Rekombinationsoperator wie auf Seite 16 definiert mit $r = 2, s = 1$ und für zwei Genome $G_1, G_2 \in \Gamma$:

$$R_a^\xi(G_1, G_2) =_{def} G_k \ (G_k \in \Gamma),$$

wobei für alle $i \in \{1, ..., |G_k|\}$ gilt:

$$G_k[i] = \begin{cases} G_1[i], & \text{falls } G_1[i] = G_2[i] \\ G_{n^\xi}[i], & \text{falls } G_1[i] \neq G_2[i] \end{cases}$$

Dabei ist $n^\xi \in \{1, 2\}$, je nach Zustand des Zufallsgenerators ξ und die Wahrscheinlichkeit für 1 und 2 beträgt jeweils $0,5$.

Variante b: Der fitnessproportionale Rekombinationsoperator ist analog definiert:

Definition (Fitnessproportionaler Rekombinationsoperator)

Sei f die Fitnessfunktion des Optimierungsproblems.

Der fitnessproportionale Rekombinationsoperator R_b^ξ ist ein Rekombinationsoperator wie auf Seite 16 definiert mit $r = 2, s = 1$ und für zwei Genome $G_1, G_2 \in \Gamma$:

$$R_b^\xi(G_1, G_2) =_{def} G_k \ (G_k \in \Gamma),$$

wobei für alle $i \in \{1, ..., |G_k|\}$ gilt:

$$G_k[i] = \begin{cases} G_1[i], & \text{falls } G_1[i] = G_2[i] \\ G_{n^\xi}[i], & \text{falls } G_1[i] \neq G_2[i] \end{cases}$$

Dabei ist $n^\xi \in \{1, 2\}$, je nach Zustand des Zufallsgenerators ξ. Die Wahrscheinlichkeit für 1 beträgt $\frac{f(G_2)}{f(G_1)+f(G_2)}$, die Wahrscheinlichkeit für 2 beträgt $\frac{f(G_1)}{f(G_1)+f(G_2)}$.

Vor dem Start der endgültigen Testläufe, die später in dieser Arbeit beschrieben werden, wurden viele Testläufe gemacht, deren Gültigkeit für die aktuelle Programmversion nicht ganz klar ist, weil sie stattfanden, als noch nicht alle Änderungen am Programm abgeschlossen waren. Während dieser Tests zeichnete sich tendenziell ab, dass die fitnessproportionale Rekombination bessere Ergebnisse liefert als die gleichverteilte. Auf die Untersuchung dieser These musste schließlich aus Zeitgründen verzichtet werden. Obwohl dies also nicht exakt untersucht wurde, wurden die endgültigen Testläufe dennoch durchgehend mit der fitnessproportionalen Rekombination durchgeführt.

Die endgültigen Rekombinationsoperatoren ergeben sich aus den oben definierten mit hinterher durchgeführter Reparatur.

Mutation Für die Mutation wurden drei Varianten implementiert, die alle im Wesentlichen auf Bitflips basieren. Ob eine Mutation bei einem bestimmten Individuum stattfindet, wird im Algorithmus durch einen Wahrscheinlichkeitsparameter gesteuert. Bei den Testläufen wurde dieser Parameter auf 1 gesetzt, das heißt Mutation fand bei allen Individuen statt, außer bei denen, die als elitär markiert waren.

Variante a: Die erste Variante wurde (wie eine der Rekombinationen) der Arbeit von Beasley et al. [Beasley1994] entnommen. Dabei wird einfach jedes Bit des Genoms mit einer kleinen Wahrscheinlichkeit negiert. Durch Angabe einer Wahrscheinlichkeit, die von der Genomlänge abhängig ist, kann auf diese Weise eine Mutation erzielt werden, die im Mittel eine bestimmte Anzahl von Bits negiert. Da theoretisch alle Bits im Genom bei einer einzigen Mutation negiert werden können, sind jedoch große Sprünge im Suchraum denkbar. Obwohl das in einem evolutionären Zyklus möglicherweise sogar erwünscht ist, da sehr große Sprünge selten vorkommen, aber möglicherweise aus lokalen Optima herausführen, wurde diese Variante schließlich dennoch nicht verwendet. Der Grund dafür ist, dass eine solche Mutation bei einer Nachbarschaftsbeziehung, wie sie in Abschnitt 2.5 definiert wurde, einen Graphen erzeugen würde, bei dem jedes Individuum mit jedem benachbart ist. Sowohl in einem evolutionären Zyklus, bei dem Baldwinsches Lernen aktiviert ist, als auch insbesondere bei der Visualisierung des Nachbarschaftsgraphen wäre das nicht sinnvoll.

Variante b: Die zweite Variante wurde als eine „deterministischere“ Abwandlung der ersten implementiert. Hier werden für ein festes k immer genau k Bits des Genoms negiert. Diese Variante wurde bei allen Testläufen und bei der Visualisierung verwendet.

Definition (k-Bit-Flip-Mutationsoperator)

> Für ein $k \in \mathbb{N}$ ist der k-Bit-Flip-Mutationsoperator M_b^ξ ein Mutationsoperator wie auf Seite 16 definiert, wobei für ein Genom $G \in \Gamma$ gilt:
>
> $$M_b^\xi(G) =_{def} G_m^\xi$$
>
> mit $G_m^\xi \in \{G' \in \Gamma \mid \text{es existieren paarweise verschiedene } i_1, ..., i_k \in \{1, ..., |G'|\} : \forall\ j \in \{i_1, ..., i_k\} : G'[j] = \neg G[j] \text{ und } \forall j \in \{1, ..., |G|\} \backslash \{i_1, ..., i_k\} : G'[j] = G[j]\}$.
>
> Welches Element dieser Menge gewählt wird, hängt vom Zustand des Zufallsgenerators ξ ab und ist über alle Elemente gleichverteilt.

Variante c: Die dritte Variante wurde als eine einmalige Vertauschung von zwei Bits des Genoms implementiert. Diese Mutation erwies sich als zu klein und wurde daher nicht verwendet. Die Idee dahinter ist aber, dass (im Unterschied zu einem 2-Bit-Flip) die Anzahl der Indexteilmengen in der Lösungsinstanz garantiert gleich bleibt. Dadurch könnte eine Reparatur seltener erforderlich werden oder zumindest weniger nachträgliche Veränderungen am Genom durchführen müssen, was eine bessere Abschätzung der Sprungweite im Suchraum ermöglichen könnte. Dazu müsste die Mutation jedoch wohl auf eine k-fache Vertauschung erweitert werden. Eine Erforschung in dieser Richtung erscheint durchaus sinnvoll, deswegen soll die Mutation auch als k-fach-Vertauschung definiert werden, obwohl sie aus Zeitgründen nur für $k = 1$ implementiert wurde.

Definition (k-fach-Swap-Mutationsoperator)

> Für ein $k \in \mathbb{N}$ ist der k-fach-Swap-Mutationsoperator M_c^ξ ein Mutationsoperator wie auf Seite 16 definiert, wobei für ein Genom $G \in \Gamma$ gilt:
>
> $$M_c^\xi(G) =_{def} G_m^\xi$$
>
> mit $G_m^\xi \in \{G' \in \Gamma \mid$ es ex. paarweise verschiedene $i_1, ..., i_k, j_1, ..., j_k \in \{1, ..., |G|\} : \forall p \in \{1, ..., k\} : G[i_p] \neq G[j_p]$ sowie $G'[i_p] = G[j_p], G'[j_p] = G[i_p]$; außerdem $\forall P \in \{1, ..., |G|\} \backslash \{i_1, ..., i_k, j_1, ..., j_k\} : G'[P] = G[P]\}$.
>
> Welches Element dieser Menge gewählt wird, hängt vom Zustand des Zufallsgenerators ξ ab und ist über alle Elemente gleichverteilt.

Auf die Bedingung $G[i_p] \neq G[j_p]$ könnte eventuell verzichtet werden, der Gedanke ist jedoch, dass viele Probleme eine sehr geringe Dichte haben; das Genom enthält in diesem Fall viel mehr Nullen als Einsen, was bei einem Verzicht auf die Bedingung, dass sich die Bits unterscheiden müssen, sehr viele Vertauschungen wirkungslos machen würde.

Die endgültigen Mutationsoperatoren ergeben sich aus den oben definierten mit hinterher durchgeführter Reparatur.

Erzeugung der Startpopulation Für die Erzeugung der Startpopulation wurden drei Varianten entworfen. Die zwei ersten (Varianten a und b) wurden getrennt getestet (siehe Abschnitt 4). Die dritte (Variante c) konnte noch nicht getestet werden, weil die Idee dazu erst zum Ende der Studienarbeit aufkam; sie war in der Programmversion, die für die Testläufe eingefroren wurde, noch nicht implementiert.

Variante a: Die erste Variante erzeugt die Individuen durch Setzen aller Genome auf $0^{|T|}$, wenn T die Famile der Teilmengen der SCP-Instanz ist, und einer nachträglichen (probabilistischen) Reparatur. Auf diese Weise werden rein zufällig gültige Individuen erzeugt. Diese Variante soll *randomisierte Startpopulationserzeugung* heißen.

Variante b: Diese Variante wurde wieder aus der Arbeit von Beasley et al. [Beasley1994] übernommen – allerdings wurde statt dem dort beschriebenen letzten Schritt eine rein probabilistische Reparatur am Ende vorgenommen.

Bemerkung: Diese Variante ist nur für Multicost-SCP geeignet, da in Schritt 2a die billigsten Index-Teilmengen gebraucht werden. Bei Generalcost hat eine Index-Teilmenge aber keine eigenen Kosten, sie hängen von den anderen in der Lösung befindlichen Index-Teilmengen ab. Bei Multicost-SCP sei dafür eine Index-Teilmenge t_1 billiger als eine andere Index-Teilmenge t_2, wenn $\kappa(\{t_1\}) < \kappa(\{t_2\})$ gilt.

Folgender Algorithmus erzeugt in vier Schritten ein Individuum ($L \subseteq T$ ist dabei die zu erzeugende Lösung, das Genom ist also $cod(L)$):

Algorithmus (Erzeugung eines Individuums, Variante b)

Eingabe: Eine Instanz des SCP (M, T, κ), $k \in \mathbb{N}^+$.

Rückgabe: Gültiges Individuum $I \in Ind$.

Sei L eine Menge, implementiert als Hashtabelle (HashSet von Java).

1. Setze $L := \varnothing$.
2. Für jeden Index $i \in M$ führe folgende Schritte aus:
   (a) Sei $\alpha_{ik} = \{t_{j_1}, ..., t_{j_k}\}$ die Menge der k billigsten Index-Teilmengen aus T, die Index i überdecken (falls weniger als k solche Index-Teilmengen existieren, sei α_{ik} die Menge aller Index-Teilmengen, die den Index i überdecken). Wähle zufällig ein $l \in \{1, ..., k\}$.
   (b) Füge t_{j_l} zu L hinzu.
3. Wende die probabilistische Reparatur auf L an.
4. Erzeuge das Genom durch anwenden der Funktion *cod* und gib das zugehörige Individuum zurück.

Iteriert angewendet erhält man durch diesen Algorithmus eine Startpopulation, deren Individuen (durch Ausnutzung von Problemwissen in Schritt 2) erwartungsgemäß eine deutlich bessere Fitness haben als bei der ersten Variante. Die Anzahl der billigsten Index-Teilmengen k wurde in allen Testläufen auf 5 gesetzt, wie es in der Arbeit von Beasley et al. empfohlen wird.

Variante c: Die dritte Variante ist eine leichte Abwandlung der zweiten, bei der noch etwas mehr Problemwissen eingesetzt wird. Anstatt die Index-Teilmengen nur nach ihren Kosten zu sortieren wird auch die Anzahl der enthaltenen Elemente berücksichtigt. Eine Index-Teilmenge t_1 ist in diesem Fall also „besser“ als eine andere Index-Teilmenge t_2, wenn gilt: $\frac{\kappa(t_1)}{|t_1|} < \frac{\kappa(t_2)}{|t_2|}$. Die dritte Variante ergibt sich aus der zweiten, indem man in Schritt 2a die k billigsten Index-Teilmengen durch diese Definition der k besten Index-Teilmengen ersetzt.

Terminierung Es wurde eine Terminierungsbedingung implementiert, die von zwei Parametern bestimmt wird. Der erste Parameter ist die maximale Generationenzahl; falls der evolutionäre Zyklus mehr als diese Anzahl an Generationen durchführt, ist die Terminierungsbedingung auf jeden Fall erfüllt und der Algorithmus bricht ab. Der zweite Parameter ist abhängig von einer Referenzlösung, die bei den Testläufen durch den Iterated-Greedy-Algorithmus von Dietmar Lippold (siehe Abschnitt 2.7) vor dem eigentlichen Lauf des evolutionären Algorithmus ermittelt wurde. Falls diese Referenzlösung im Lauf des Zyklus erreicht oder verbessert wird, dann wird noch höchstens die Anzahl an Generationen durchgeführt, die der zweite Parameter festlegt, und dann abgebrochen. Bei den Testläufen wurden verschiedene Werte für die maximale Generationenzahl genommen: bei Unicost-Problemen waren es 200, bei den normalen Läufen mit Multicost-Problemen 800 und bei den Langzeit-Läufen mit Multicost-Problemen 8000 Generationen. Für die Anzahl an Generationen, die nach Erreichen des Referenzwerts durchgeführt werden, wurde für die normalen Läufe mit Unicost- und Multicost-Problemen 200 genommen, für die Langzeit-Läufe mit Multicost-Problemen 1000 (siehe zur genauen Durchführung der Testläufe Abschnitt 4 und Anhang).

Bewertung Die Bewertungsfunktion war ohne Baldwin-Lernen einfach die Funktion κ. Mit Baldwin-Lernen wurde die gelernte Fitness F_{N_M} verwendet (siehe Abschnitt 2.5), wobei N_M die Nachbarschaft ist, die sich durch die jeweils benutzte Mutation M ergibt; für die Testläufe und die Visualisierung war das immer die k-Bit-Flip-Mutation M_b^ξ. Für die Testläufe konnte allerdings aus Effizienzgründen nicht die gesamte Nachbarschaft berücksichtigt werden. Es wurden in diesem Fall nur 3 zufällig ausgewählte Nachbarn angesehen und die beste Fitness aus dieser kleinen Nachbarschaft berechnet.

Der Reparaturmechanismus Es wurden zwei Reparaturmechanismen implementiert, die beide aus einem beliebigen Genom ein gültiges Genom wie oben definiert erzeugen.

Der erste Mechanismus (Variante a) tut dies stochastisch, der zweite (Variante b) deterministisch. Beide arbeiten nach folgendem Algorithmus, wobei sie sich jeweils in der Ausführung der Wahl der Index-Teilmengen in den Schritten 2a und 3a unterscheiden:

Algorithmus (Reparaturmechanismus)

Eingabe: Eine Instanz des SCP (M, T, κ), $G \in \{0,1\}^{|T|}$.

Rückgabe: Gültiges Genom $G' \in \Gamma$.

1. Setze $G' := G$.
2. Solange ein Index in M existiert, der von $dec(G')$ nicht überdeckt wird, führe folgendes aus:
   (a) Wähle j mit $G'[j] = 0$ und die zu j gehörige Index-Teilmenge $t_j \in T$ enthält mindestens einen Index, der noch nicht überdeckt wurde.
   (b) Setze $G'[j] := 1$.
3. Solange eine Index-Teilmenge in $dec(G')$ existiert, die nur Indizes überdeckt, die schon durch andere Index-Teilmengen aus $dec(G)$ überdeckt werden führe folgendes aus:
   (a) Wähle eine solche Index-Teilmenge $t_j \in dec(G)$.
   (b) Setze das zugehörige Bit auf 0: $G'[j] := 0$
4. Gib G' zurück.

Variante a: In der ersten Variante werden j in Schritt 2a und die Index-Teilmenge t_j in Schritt 3a rein zufällig (gleichverteilt) gewählt. Diese Variante wurde für die Testläufe verwendet.

Variante b: In der zweiten Variante wird immer das kleinste j in Schritt 2a, das die Bedingung erfüllt, und das kleinste j in Schritt 3a, so dass t_j die Bedingung erfüllt, gewählt. Diese Variante wurde für die Visualisierung verwendet, um eine deterministische Darstellung der Nachbarschaft zu ermöglichen. Anderenfalls würden sich je nach Zustand des Zufallsgenerators unterschiedliche Nachbarschaften für dasselbe Individuum ergeben, da die Reparatur Teil jeder Mutation ist.

Diversitätsmessung In den evolutionären Zyklus wurde ein Mechanismus implementiert, der eine Population, die zu homogen geworden ist, aus möglicherweise erreichten lokalen Mi-

nima herausholen soll. Zu diesem Zweck kann statt der gewöhnlichen Mutation eine größere Mutation angestoßen werden, die im Fall der benutzten Mutation M_b^ξ eine größere Anzahl an Bits k flippt (ansonsten aber den gleichen Mutationsoperator verwendet); sie wird *Makromutation* genannt. Um zu bestimmen, wann eine Population zu homogen geworden ist, wird zu Beginn jedes Zyklus eine Messung durchgeführt, die die „Verschiedenheit“ der sich in der Population befindenden Genome angibt. Diese Zahl wird *Diversität* genannt und auf folgende Weise berechnet:

Algorithmus (Diversitätsmessung)

Eingabe: Population $P = (I_1, ..., I_n)$, $k \in \mathbb{N}$.

Rückgabe: Diversitätszahl d.

1. Setze $d' := 0$
2. Führe k mal folgendes aus:
   (a) Wähle zufällig $i, j \in \{1, ..., n\}$.
   (b) Setze $d' := d' + u(I_i.G, I_j.G)$.
3. Gib $d := d'/k$ zurück.

Dabei wird der Unterschied u zweier Genome (Schritt 2b) folgendermaßen definiert:

$$u : \Gamma \times \Gamma \to \mathbb{R} : u(G_1, G_2) = \frac{\text{Anzahl der Stellen } i \text{ mit } G_1[i] \neq G_2[i]}{\text{Anzahl der Stellen } i \text{ mit } (G_1[i] = 1 \vee G_2[i] = 1)}$$

Der Grund für die Division durch die Anzahl der Bitstellen, an denen mindestens eines der zu vergleichenden Genome 1 ist, liegt darin, dass die Diversitätszahl möglichst zwischen Problemen verschiedener Größe und Dichte vergleichbar sein sollte. Eine Division durch die Genomlänge hätte den Nachteil, dass Probleme verschiedener Dichte nicht bezüglich ihrer Diversität vergleichbar wären; bspw. kommen bei Problemen mit geringer Dichte viele Nullen in den Genomen vor, wodurch die Wahrscheinlichkeit steigt, dass zwei Genome an einer Bitstelle den gleichen Wert haben.

Die Eingabe k des Algorithmus wurde bei den Testläufen auf die Populationsgröße n gesetzt. Im Optimalfall müssten alle Individuen paarweise miteinander verglichen werden, was aber wegen der quadratischen Laufzeit nicht durchführbar war. Stattdessen sollte eine stochastische Messung genügen, bei der die Wahrscheinlichkeit für jedes Individuum „groß“ ist, mindestens einmal mit einem anderen verglichen zu werden.

Die Wahrscheinlichkeit mindestens einmal für einen Vergleich herangezogen zu werden, beträgt bei der beschriebenen Variante für jedes Individuum mindestens $1 - \mathrm{e}^{-2} \approx 0,865$.

Beweis:

Die Population habe die Größe $n > 0$ und die Eingabe k für den Algorithmus sei n.

Dann werden $2n$ Individuen unabhängig mit Zurücklegen aus der Population gezogen und für einen Vergleich herangezogen.

Die Wahrscheinlichkeit, gezogen zu werden, beträgt für ein bestimmtes Individuum der Population in Abhängigkeit der Populationsgröße:

$$\begin{aligned} p(n) &= 1 - \left(1 - \frac{1}{n}\right)^{2n} \\ &= 1 - \left(\left(1 - \frac{1}{n}\right)^{n}\right)^{2} \\ &\overset{n\to\infty}{\longrightarrow} 1 - \left(\mathrm{e}^{-1}\right)^{2} \\ &= 1 - \mathrm{e}^{-2} \approx 0,865. \end{aligned}$$

Weiter ist die Funktion $p(n)$ monoton fallend. Dafür ist es ausreichend zu zeigen, dass

$$p_1(n) \ =_{def} \ \left(1 - \frac{1}{n}\right)^{n}$$

monoton steigend ist. $p_1(n)$ sei dafür erweitert auf den Definitionsbereich $\mathbb{R}^+ \backslash [0,1]$. (Für den Speizalfall $n = 1$ gilt $p(n) = 1 > 1 - \mathrm{e}^{-2}$.) Die Ableitung ist:

$$p_1'(n) = \left(\ln\left(1 - \frac{1}{n}\right) + \frac{1}{n-1}\right) \cdot \underbrace{\left(1 - \frac{1}{n}\right)^{n}}_{\geq 0}.$$

Wie man sieht, ist der zweite Faktor für alle n positiv. Der erste ist auch positiv:

$$\frac{1}{n-1}$$

ist positiv. Mit

$$\ln(x) \leq x - 1 \ \forall x \in \mathbb{R}^+$$

gilt:

$$\ln\left(1 - \frac{1}{n}\right) = \ln\left(\frac{n-1}{n}\right) = -\ln\left(\frac{n}{n-1}\right) = -\ln\left(1 + \frac{1}{n-1}\right) \geq -\frac{1}{n-1}$$

und damit:

$$\begin{aligned} &\ln\left(1 - \frac{1}{n}\right) + \frac{1}{n-1} \geq 0 \\ &\Rightarrow p_1'(n) \geq 0 \ \text{ für alle } n \text{ im Definitionsbereich.} \end{aligned}$$

Damit ist gezeigt, dass $p_1(n)$ monoton steigend und somit $p(n)$ monoton fallend ist. Mit $\lim_{n\to\infty} p(n) = 1 - \mathrm{e}^{-2}$ ist gezeigt, dass die Wahrscheinlichkeit, für einen Vergleich herangezogen zu werden, für jedes Individuum mindestens $1 - \mathrm{e}^{-2}$ beträgt.

□

Für die Testläufe wurde die Populationsgröße $n = 400$ verwendet. Damit ergibt sich eine Wahrscheinlichkeit von ziemlich genau $0,865$.

Sperrung von Individuen Vor der Ausführung jeder Mutation, Rekombination und Selektion kann in der Population eine Sperrung bestimmter Individuen durchgeführt werden, die dafür sorgt, dass diese Individuen von dem darauffolgenden Operator nicht verändert oder ausselektiert werden können. Dadurch kann erzwungen werden, dass beispielsweise das beste Individuum immer in der Population erhalten bleibt. Die Sperrung vor der Selektion ist bei probabilistischer Selektion notwendig, damit die gesperrten Individuen garantiert nicht verlorengehen. In der einzigen bisher implementierten Variante werden die besten k Individuen der Population gesperrt. Das entspricht einer elitären Variante des evolutionären Algorithmus, wie sie in Abschnitt 2.2.3 beschrieben wurde. In den Testläufem wurde ein ausführlicher Vergleich zwischen elitären und nichtelitären Läufen durchgeführt.

3.2 Implementierung

Das implementierte System gliedert sich grob in die Pakete

1. evolutionaererAlgorithmus,
2. operatoren,
3. graphVis und
4. test.

Während das Paket 1 den Kern des evolutionären Algorithmus darstellt, in dem insbesondere der evolutionäre Zyklus gestartet und gesteuert wird, sind im Paket 2 alle in Abschnitt 3.1.2 beschriebenen Operatoren als Module verfügbar. Das Paket 3 ist für die Visualisierung des Nachbarschaftsgraphen und die Verwaltung von Graphstrukturen zuständig; es benutzt die Pakete 1 und 2, ist aber kein Teil des evolutionären Zyklus und wird daher erst in Abschnitt 5.1 beschrieben. Im Paket 4 befinden sich verschiedene Klassen zum Testen des evolutionären Algorithmus, die zum Teil auf dem Framework von Dietmar Lippold basieren, darunter zufällige SCP-Instanz-Erzeugung und das Einlesen von Beispielproblemen aus Dateien; außerdem gibt es einige Klassen zum Testen der Graphstruktur und eine Klasse zum Verarbeiten der Statistiken, die während eines evolutionären Laufs erzeugt werden.

3.2.1 Das Paket „evolutionaererAlgorithmus“

Das Paket evolutionaererAlgorithmus enthält die folgenden Klassen:

- *BewertungsVergleich*: Hier wird die für die Sortierung der Individuen notwendige Ordnungsrelation „besser als“ definiert. Diese gilt in diesem Fall zwischen I_1 und $I_2 \in Ind$ genau dann, wenn $I_1.F \leq I_2.F$.
- *EvolutionaereUeberdeckung*: Diese Klasse stößt den evolutionären Zyklus an. Ihrem Konstruktor wird ein Satz von Operatoren und Parametern übergeben (falls diese nicht übergeben werden, wird ein Satz von Standardoperatoren und Standardparametern erzeugt). Diese sind Objekte des Typs OperatorenAuswahl bzw. ParameterAuswahl (s. u.). Die Hauptmethode der Klasse ist ueberdeckung, die eine Instanz des SCP (das ist ein Objekt des Typs *ItmFamilie* aus dem Framework von Dietmar Lippold

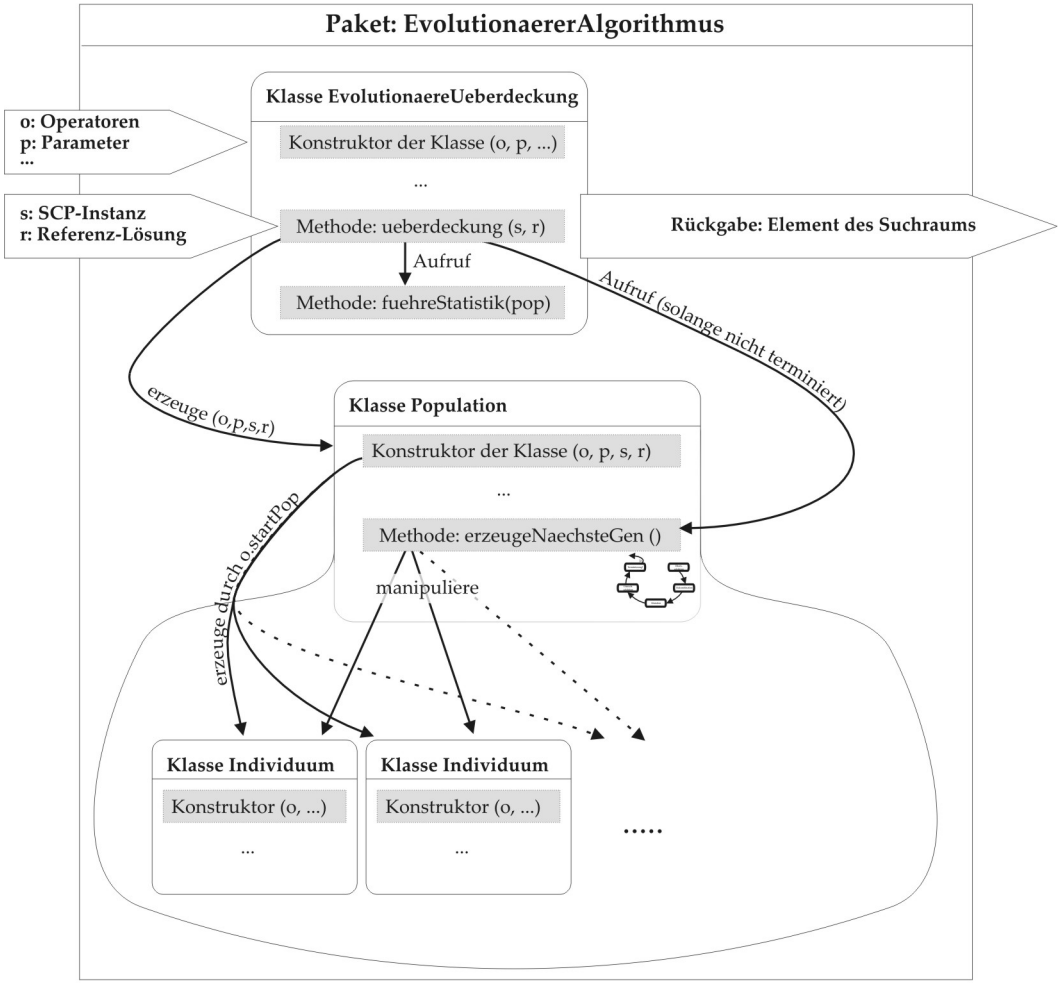

Abbildung 4: *Arbeitsweise der wesentlichen Klassen im Paket „evolutionaererAlgorithmus“*

– zu beachten ist der Unterschied zum Begriff „Itm-Familie“, der ansonsten im Text lediglich für eine Menge von Indexteilmengen steht) und optional eine bereits ermittelte Referenzlösung als Eingabeparameter erhält. Mit diesen Informationen und den bereits erhaltenen Operatoren und Parametern wird ein Objekt der Klasse Population (s. u.) erzeugt, das für den evolutionären Zyklus selbst verantwortlich ist. Die Methode ueberdeckung muss nur jeweils die Erzeugung einer neuen Generation anfordern (das ist ein einzelner Durchlauf des Zyklus) und am Rückgabewert erkennen, ob eine Terminierung bereits erfolgt ist. Zwischen den Generationen werden in einem Objekt des Typs Statistik die aktuellen statistischen Daten der Population gespeichert; nach einer Terminierung werden diese in eine Datei auf der Festplatte geschrieben.

- *Genom*: Objekte des Typs Genom sind der wichtigste Bestandteil der Klasse Individuum. Zu Beginn der Arbeit bestand ein Genom nur aus einem Bitstring, der die Bedeutung hatte, die im vorhergehenden Abschnitt durch die Funktion *cod* definiert worden war. Es stellte sich aber heraus, dass bei der Dekodierung dadurch jedesmal ein neues Objekt des Typs ItmFamilie erzeugt werden musste, was häufig vorkam und sehr ineffizient war. Um das zu vermeiden, wird bei der aktuellen Version mit dem Bitstring bei jedem Genom auch ein Objekt des Typs *ItmFamilie* gespeichert, das die Dekodierung des Bitstrings enthält und bei jeder Mutation oder sonstigen Veränderung ebenfalls verändert wird.

- *Individuum*: Die Klasse Individuum besteht aus einem Genom, einer (Float-) Bewertung, einem booleschen Flag, das anzeigt, ob das Genom sich seit der letzten Berechnung der Bewertung verändert hat (und somit bei einer Abfrage der Bewertung eine Neuberechnung erforderlich ist), und einem booleschen Flag, das anzeigt, ob das Individuum gesperrt oder für Mutationen freigegeben ist. Außerdem enthält es eine Methode zur Mutation des Genoms und eine Methode zur Reparatur des Genoms (jeweils in Abhängigkeit eines übergebenen Mutations- bzw. Reparaturoperators), sowie verschiedene Methoden zur Abfrage des Zustands des Individuums. Die Klasse ist für alle Veränderungen am Genom verantwortlich und insbesondere dafür, dass es sich stets in einem gültigen Zustand befindet.

- Konstanten: Diese Klasse enthält vor allem die Werte für die Standardparameter, die in der Klasse EvolutionaereUeberdeckung erzeugt werden, falls kein Parametersatz an den Konstruktor übergeben wurde. Außerdem enthält sie den relativen Pfad für das Speichern der Statistiken.

- *OperatorenAuswahl*: Objekte dieses Typs dienen zum Speichern eines Operatorensatzes. Sie haben Methoden zum Setzen und Abfragen aller Operatoren, die für den evolutionären Algorithmus zur Verfügung stehen. Operatoren sind alle Objekte, die durch die Klassen im Paket operatoren definiert werden.

- *OpUndParErzeugung*: Diese Klasse dient zum Erzeugen von Standardoperatoren und Standardparametern. Die Parameter werden aus der Klasse Konstanten gelesen. Die Operatoren sind fest in der Klasse selbst kodiert.

- *ParameterAuswahl*: Objekte dieses Typs speichern die Parameter in analoger Weise wie in OperatorenAuswahl die Operatoren gespeichert werden.

- *Population*: Beim Erzeugen eines Objekts dieses Typs werden Operatoren, Parameter, eine Instanz des SCP und eine Referenzlösung an den Konstruktor der Klasse

übergeben; dann wird zunächst die Erzeugung einer Startpopulation durchgeführt, die als Liste von Individuen in einem Attribut gespeichert wird. Der evolutionäre Zyklus muss durch wiederholtes Aufrufen der Methode erzeugeNaechsteGeneration am Laufen gehalten werden. In dieser Methode wird zunächst überprüft, ob die Terminierungsbedingung bereits erfüllt ist; ist dies der Fall wird $false$ zurückgegeben und die Methode beendet. Sonst wird eine Diversitätsmessung durchgeführt. Dann werden anhand der übergebenen Parameter und Operatoren und in Abhängigkeit der Diversität die Individuen der Elternselektion, Rekombination, Mutation und Umweltselektion unterworfen. Zuletzt wird *true* zurückgegeben. Neben dieser Methode enthält die Klasse noch Methoden zum Abfragen des k-t besten Individuums, dessen Genoms oder dessen Bewertung, der Durchschnittsbewertung der Population, der aktuellen Generation und des Werts der Referenzlösung.

- *Statistik*: Diese Klasse bietet Funktionen für die Verwaltung und Speicherung von Statistik-Verläufen an. Dabei können bspw. Durchschnittsverläufe gespeichert, Maxima, Minima, Mediane, Standardabweichungen, etc. berechnet werden, Statistiken in eine Datei gespeichert, sowie bereits gespeicherte Statistiken geladen werden, usw.

- *TeilmengenZuordnung*: Diese Klasse dient der Kodierung und Dekodierung von Genomen. Insbesondere werden hier den einzelnen Index-Teilmengen einer Itm-Familie Indizes zugewiesen, was für eine eindeutige Zuordnung einer Bitstelle im Genom zu einer Index-Teilmenge notwendig ist. Außerdem werden einige Methoden der Klasse ItmFamilie angeboten, um zu ermöglichen, dass ein Objekt des Typs ItmFamilie, das die SCP-Instanz repräsentiert, an keiner anderen Stelle im Algorithmus gespeichert werden muss.

Abbildung 4 zeigt die Arbeitsweise der wichtigsten Klassen im Paket evolutionaererAlgorithmus. Die Klasse EvolutionaereUeberdeckung dient als Steuerung. Sie erhält die Problemparameter und die Parameter und Operatoren für die Evolution. Daraus erzeugt sie ein Objekt der Klasse Population. Dann wird wiederholt abwechselnd ein neuer Zyklus in der Population angestoßen und das Erheben der statistischen Daten durchgeführt. Nach Terminierung wird eine Lösung zurückgegeben.

3.2.2 Das Paket „operatoren“

Dieses Paket hat auf der ersten Hierarchieebene keine Klassen, sondern besteht aus einer Reihe von Unterpaketen. Bis auf das Paket 10 sind die Pakete eine Implementierung der im Abschnitt 3.1.2 beschriebenen Operatoren. Die Unterpakete des Pakets operatoren sind:

1. selektion
2. rekombination
3. mutation
4. startpopulation
5. terminierung
6. bewertung

7. gueltigkeit (In diesem Paket ist der Reparaturmechanismus implementiert.)
8. diversitätsmessung
9. sperren
10. teilmengenSortierung

Alle Pakete außer 10 sind grundsätzlich in ähnlicher Weise aufgebaut. Sie enthalten ein abstraktes Interface, welches die Schnittstelle für den jeweiligen Operator festlegt, und eine Reihe von Implementierungen dieses Interfaces. Die Rekombination hat zusätzlich ein weiteres Unterpaket „einzelRek“, welches wieder ein Interface und mehrere Implementierungen hat; dort wird eine einzelne Rekombination von r Eltern zu s Nachkommen definiert. Im evolutionären Zyklus wird nur auf die Interfaces verwiesen, während die Angabe, welche konkrete Implementierung verwendet werden soll, als Parameterübergabe „von außen“ erfolgt. Der abstrakte evolutionäre Zyklus ist in Abbildung 5 dargestellt. Auch die potenzielle Abzweigung zu einer Makromutation ist dort eingezeichnet. Rechts im Bild sind die bereits implementierten Klassen als Module dargestellt, die in das jeweils passende Interface eingesetzt werden können.

Das Paket 10 stellt eine Ausnahme dar. Es enthält nur eine Klasse, die für die Zuordnung von Index-Teilmengen zu Bitstellen im Genom zuständig ist. Prinzipiell sollten der Einheitlichkeit halber alle Teile des Algorithmus unabhängig von der konkreten Sortierung der Index-Teilmengen im Genom sein. Die Implementierung des Startpopulations-Interfaces in den Varianten b) und c) (siehe Abschnitt 3.1.2) benutzt allerdings zurzeit noch diese Sortierung; sie weist den Index-Teilmengen Indexwerte in Abhängigkeit ihrer Kosten zu, wobei die Index-Teilmengen mit den geringsten Kosten die kleinsten Indexwerte erhalten. (Dieses Vorgehen ist eigentlich nur für Multicost- und Unicost-Probleme geeignet; allgemein wird einer Index-Teilmenge t hier aber abhängig von den Kosten $\kappa(\{t\})$ ein Index zugewiesen, was auch für Generalcost definiert ist. Da der Sinn der Sortierung bei Generalcost-Problemen verlorengeht, sollte ihr keine semantische Bedeutung zukommen. Im Fall der Varianten b) und c) der Startpopulationserzeugung ist es aber in Ordnung, da diese ohnehin nicht für Generalcost-Probleme geeignet sind.)

3.2.3 Das Paket „test“

Das Paket test besteht aus den folgenden Klassen:

- *GrapherzeugungZufall*: Erzeugt einen zufälligen gerichteten Graphen zu Testzwecken in der im Paket GraphVis definierten Graphstruktur.
- *Konstanten*: Enthält einige Konstanten des Pakets.
- *ResultStatisticCreation*: Aus dem Paket von Dietmar Lippold entnommene und abgewandelte Klasse zum Parsen von Dateien, in denen Beispiel-Probleme gespeichert sind, und zum Starten von Testläufen.
- *StatistikAusgeben*: Klasse zum Umwandeln der während des evolutionären Laufs gespeicherten Statistiken in „menschenlesbare“ Form.

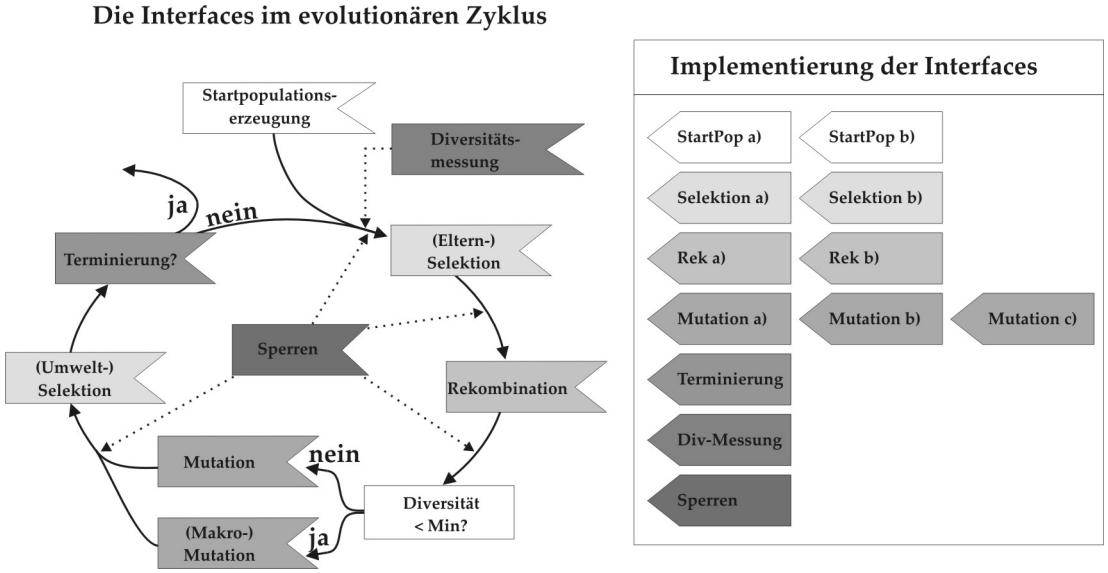

Abbildung 5: *Der evolutionäre Zyklus arbeitet auf einer Reihe von Interfaces, die konkret als unterschiedliche Klassen implementiert werden können. Jede konkrete Implementierung kann in das passende Interface eingesetzt werden. Rechts im Bild sind die bisher implementierten Klassen dargestellt. Sie entsprechen den in Abschnitt 3.1.2 beschriebenen Varianten. Die Schattierung deutet an, welche Interfaces zu welcher Implementierung passen.*

- *TestEvolUeberdeckBib*: Klasse zum Starten eines evolutionären Laufs mit Problemen die in Dateien gespeichert sind. Die Klasse baut im Wesentlichen auf der Klasse ResultStatisticCreation auf.
- *TestEvolUeberdeckZufall*: Klasse zum Starten eines evolutionären Laufs mit einem zufällig generierten Problem.
- *TestItGreedyBib*: Klasse zum Starten von Testläufen mit dem von Dietmar Lippold implementierten Iterated-Greedy-Algorithmus anhand von Problemen, die in Dateien gespeichert sind.

3.3 Komplexitätsbetrachtungen

Wie ausführliche Analysen mit Profiling-Programmen zeigten, wird der größte Teil der Rechenzeit bei evolutionären Läufen für die Dekodierung des Genotyps und allgemein für Operationen im phänotypischen Suchraum verbraucht, also an der Schnittstelle zwischen dem evolutionären Algorithmus und dem Framework von Dietmar Lippold. Es wurden sowohl am evolutionären Algorithmus als auch am Framework diesbezüglich ständig Verbesserungen vorgenommen; beispielsweise konnte durch die Speicherung einer Dekodierung des Bitstrings im Genom, die bei jeder Veränderung am Bitstring mitverändert wurde, eine Laufzeitverbesserung erzielt werden. Dennoch geht auch in der letzten Version der Hauptteil der Rechenzeit an dieser Schnittstelle verloren. Es scheint hier auch keine wesentliche Verbesserung durch naheliegende Veränderungen mehr möglich zu sein.

Im Folgenden sollen die einzelnen Elemente des evolutionären Algorithmus für eine SCP-Instanz (M, T, κ) im Bezug auf ihre Rechenzeit in Abhängigkeit von

- der Anzahl der Index-Teilmengen $m := |T|$,
- der Populationsgröße $|Pop| = n$ und
- eventuell sonstiger Eingaben

untersucht werden. Der Übersichtlichkeit halber werden die Zeitabschätzungen, die eigentlich Funktionen $t : \mathbb{N} \times \mathbb{N} \times ... \to \mathbb{R}$ darstellen, nur als t bzw. T geschrieben; der Definitionsbereich ergibt sich implizit durch die verwendeten Größen.

Zu beachten ist, dass für numerische Operationen eine lineare Zeitabschätzung in der Eingabelänge verwendet wurde.

Die Rechenzeiten der Elemente aus dem Framework seien allgemein durch folgende Funktionen als Platzhalter vertreten (sie sind im Unterschied zu den Elementen des evolutionären Algorithmus durch T, statt t, gekennzeichnet):

- $T_{erzeugen}$: Zeit, die zum Erzeugen eines Objekts vom Typ ItmFamilie benötigt wird.
- T_{klonen}: Zeit, die zum Klonen einer Itm-Familie benötigt wird.
- $T_{tmHinzufuegen}$: Zeit, die zum Hinzufügen einer Index-Teilmenge zu einer Itm-Familie benötigt wird.
- $T_{tmEntfernen}$: Zeit, die zum Entfernen einer Index-Teilmenge aus einer Itm-Familie benötigt wird.
- $T_{tmIstNotwendig}$: Zeit zum Überprüfen, ob eine Index-Teilmenge in einer Itm-Familie benötigt wird oder ob sie weggelassen werden kann, ohne die Überdeckung zu zerstören.
- $T_{\#nichtNotwendige}$: Zeit zum Ermitteln der Anzahl der nicht benötigten Index-Teilmengen in einer Itm-Familie.
- T_{kosten}: Zeit für die Berechnung der Kosten κ einer Itm-Famile. Diese kann als konstant angenommen werden, denn die Berechnung erfolgt im Framework durch einen einzigen Zugriff auf ein Attribut.

Die Elemente des evolutionären Algorithmus haben folgende Zeitabschätzungen:

Negation eines Bits im Genom (Bitflip): Die Negation eines Bits im Genom hat auch eine Veränderung der im Genom gespeicherten Itm-Familie zur Folge. Je nachdem ob von 0 zu 1 negiert wird oder umgekehrt, wird die Zeit $T_{tmHinzufuegen}$ oder $T_{tmEntfernen}$ gebraucht.

$$t_{bitFlip} = O(\max(\{T_{tmHinzufuegen}, T_{tmEntfernen}\}))$$

Erzeugen eines neuen Individuums: Es wird $O(m)$ Zeit für die Reservierung von m Bits im Speicher für den Bitstring benötigt und $O(1)$ für die übrigen Attribute. Dazu kommt die Zeit $T_{erzeugen}$ für das Erzeugen des Objekts vom Typ ItmFamilie, das Teil des Genoms ist, sowie die Zeit $t_{reparatur}$ für eine Reparatur des Genoms:

$$t_{erzeugenInd} \in O(m + T_{erzeugen} + t_{reparatur}).$$

Dekodierung eines Genoms: Die Dekodierung des Genoms erfolgt in konstanter Zeit, denn sie ist nur ein Zugriff auf ein ständig mitgeführtes und aktualisiertes Attribut:

$$t_{dec} \in O(1)$$

Bewertungsberechnung: Die Berechnung der Bewertung f eines Individuums erfolgt nach dem folgenden Algorithmus (insbesondere ist eine Bewertung ohne Baldwin-Lernen möglich, wenn $k = 0$ als Eingabe übergeben wird):

Algorithmus (Bewertungsberechnung)

Eingabe: $I \in Ind$, $k \geq 0$ (Anzahl der Nachbarn), *ops* (ein Operatorensatz).

Rückgabe: $f \in \mathbb{R}$.

Sei L eine Liste.

1. Falls die gespeicherte Bewertung $I.F$ noch aktuell ist, gib $I.F$ zurück. Sonst:
2. $f := \kappa(dec(I.G))$
3. $L := ops.mutation.mutListe(G, k)$ (Hier wird eine Liste erzeugt, die k Elemente enthält. Jedes Element i bezeichnet die Bitstellen, die in G geflippt werden müssen, um den i-ten Nachbarn für den entsprechenden Mutationsoperator zu erhalten.)
4. Für alle $e \in L$ führe folgendes aus:
   (a) $G' := klone(I.G)$
   (b) Wende die in e beschriebenen Bitflips auf G' an.
   (c) Repariere G': $G' := ops.gueltigkeit.macheValide(G')$.
   (d) falls $\kappa(dec(G')) \leq f$, setze $f := \kappa(dec(G'))$.
5. Gib f zurück.

Der erste Schritt benötigt konstante Zeit. Die Zeit für Schritt 2 beträgt $t_{dec} + T_{kosten} \in O(1)$. Schritt 3 benötigt für jeden Nachbarn die Zeit für das Erzeugen der Liste der zu flippenden Bits, also $k \cdot O(t_{mutation})$ für den in *ops* festgelegten Mutationsoperator *mutation*. Schritt 4 benötigt für jeden Nachbarn die Zeit zum Klonen des Genoms (da im Genom zum Bitstring auch eine Itm-Familie gespeichert ist, wird $O(T_{klonen})$ Zeit benötigt), die Zeit zum Anwenden der Bitflips, also $O(t_{mutation}) \cdot t_{bitFlip}$, die Zeit für die Reparatur, also $O(t_{reparatur})$ und die Zeit zum Berechnen der Kosten, also $O(1)$. Zusammen ergibt sich:

$$\begin{aligned} t_{bewertung} \in\ & O(1) + O(1) + k \cdot O(t_{mutation}) \\ & + k \cdot (O(T_{klonen}) + O(t_{mutation} \cdot t_{bitFlip}) + O(t_{reparatur}) + O(1)) \\ & = O(k \cdot (T_{klonen} + t_{mutation} \cdot t_{bitFlip} + t_{reparatur})). \end{aligned}$$

Reparatur: Die Reparatur eines Genoms wird nach dem Algorihmus auf Seite 36 vorgenommen. Schritt 1 benötigt die Zeit $O(1)$. Die Schleife ab Schritt 2 wird höchstens $|M|$ mal

durchlaufen, falls zu Beginn kein Index überdeckt wird und jede hinzugenommene Index-Teilmenge nur einen neuen Index überdeckt. Die Zeit für die Wahl eines Index in Schritt 2a beträgt $O(m \cdot T_{tmIstNotwendig})$. Die Zeit für die Zuweisung in Schritt 2b ist $t_{bitFlip}$. Die Schleife ab Schritt 3 wird höchstens $m-1$ mal durchlaufen, falls alle Indizes aus M durch eine einzige Index-Teilmenge überdeckt werden und alle anderen Index-Teilmengen vorher entfernt werden. Analog zur vorherigen Schleife kann man für Schritt 3a und 3b argumentieren: Die Wahl der Index-Teilmenge kostet $O(m \cdot T_{tmIstNotwendig})$, die Zuweisung $t_{bitFlip}$ Zeit. Insgesamt ergibt sich:

$$\begin{aligned} t_{reparatur} &\in |M| \cdot O(m \cdot T_{tmIstNotwendig} + t_{bitFlip}) + (m-1) \cdot O(m \cdot T_{tmIstNotwendig} + t_{bitFlip}) \\ &= O((|M| + m) \cdot (m \cdot T_{tmIstNotwendig} + t_{bitFlip})). \end{aligned}$$

Diese Abschätzung gilt für beide implementierten Reparaturvarianten, trotz der unterschiedlichen Art ihrer Index-Wahl. Bei der zufälligen Variante werden die Indizes, die potenziell zu 1 bzw. 0 geflippt werden können, in Listen gehalten und entfernt, falls sie wegen der jeweiligen Eigenschaft der zugehörigen Index-Teilmenge (nicht notwendig bzw. notwendig) nicht geeignet sind. Dadurch wird bei der Wahl in den Schritten 2a und 3a jeder Index höchstens einmal betrachtet. Bei der linearen Variante werden die Indizes der Reihe nach durchlaufen und somit ebenfalls jeweils nur einmal betrachtet.

Die Zeitkomplexität der Reparatur ist quadratisch in Abhängigkeit von $m = |T|$ und der es kommt der (gewöhnlich kleinere) Term $|M| \cdot m$ in der Abschätzung vor. Das ist unbefriedigend, weil die Anzahl der Indizes, sowie die der Index-Teilmengen vom Problem abhängig ist. Obwohl man davon ausgehen kann, dass der schlimmste Fall fast nie vorkommt, weil die Probleme gewöhnlich eine geringe Dichte haben und wohl nur wenig nach einer kleinen Veränderung am Genom repariert werden muss, sollte möglicherweise nach einer Alternative für diese Reparatur gesucht werden. Auch in den Profiling-Läufen kam heraus, dass ein Großteil der Rechenzeit bei der Reparatur verbraucht wird.

Diversitätsmessung: Der Zeitverbrauch der Diversitätsmessung ist nach dem Algorithmus auf Seite 37:

$$t_{div} \in O(m \cdot k).$$

Bei den Testläufen wurde immer $k = n$ gesetzt.

Mutation: Es soll hier nur die benutzte Variante der Mutation FM_b^ξ betrachtet werden. (Die erste Variante, bei der jedes Bit mit einer kleinen Wahrscheinlichkeit geflippt wird, benötigt trivialerweise die Zeit $O(m \cdot t_{bitFlip})$; die dritte Variante, wo k mal eine Vertauschung von zwei Bits stattfindet wurde in dieser Form noch nicht implementiert, die implementierte Variante benutzt implizit $k = 1$ und benötigt $O(m \cdot t_{bitFlip})$ Zeit).

Im Folgenden soll $rand(M)$ für eine Menge M ein zufälliges Element dieser Menge zurückgeben. Die Wahrscheinlichkeit ist für jedes Element gleichverteilt.

Die Mutation FM_b^ξ arbeitet nach dem folgenden Algorithmus:

Algorithmus (Mutation FM_b^ξ)

Eingabe: $G \in \Gamma$, $0 \leq k \leq |G|$.

Rückgabe: $G' \in \{0,1\}^m$.

1. Sei *nichtmut* eine Menge mit $nichtmut := \{1, ..., |G|\}$.
2. Führe k mal folgendes aus:
   (a) $i := rand(nichtmut)$.
   (b) $nichtmut := nichtmut \backslash \{i\}$.
   (c) $G := bitFlip(G, i)$. (Dabei sei $bitFlip(G, i)$ der Bitstring, der entsteht, wenn man in G das i-te Bit negiert.)
3. Gib $G' := G$ zurück.

Dieser Algorithmus benötigt für Schritt 1 $O(m)$ Zeit für die Reservierung des Speichers. Die Schleife wird k mal ausgeführt und benötigt $O(1)$ Zeit für Schritt 2a, für Schritt 2b die Zeit zum Entfernen eines Elements aus einem *HashSet* von Java ($T_{hashEntfernen}$) und die Zeit $t_{bitFlip}$ für Schritt 2c. Insgesamt ergibt sich:

$$t_{mut} \in O(m) + O(k \cdot (T_{hashEntfernen} + t_{bitFlip})).$$

Die Mutation der gesamten Population hat dann die Zeitabschätzung:

$$O(n \cdot (t_{mut} + t_{reparatur})).$$

Rekombination: Die bisher implementierten Rekombinationsvarianten arbeiten nach dem folgenden allgemeinen Algorithmus (die Liste von Kindindividuen hat dabei immer die Länge 1, ist jedoch aus Gründen der Erweiterbarkeit dennoch eine Liste):

Algorithmus (Rekombination)

Eingabe: Liste von Elternindividuen $(E_1, ..., E_s)$.

Rückgabe: Liste von Kindindividuen (K_1).

1. Weise allen Elternindividuen $E_1, ..., E_s$ Wahrscheinlichkeiten $w_1, ..., w_s \in [0, 1]$ zu, mit $\sum_{i \in \{1,...,s\}} w_i = 1$.
2. Sei G ein Bitstring mit $G := 0^{|E_1.G|}$
3. Für alle $i \in \{1, ..., |G|\}$ tue folgendes:
   (a) Falls $\forall k, l \in \{1, ..., s\} : E_k.G[i] = E_l.G[i]$, setze $G[i] := E_1.G[i]$.
   Sonst tue folgendes:
      i. $r := rand(]0, 1])$.
      ii. Setze $G[i] := E_k.G[i]$ mit $w_1 + ... + w_{k-1} < r$ und $w_1 + ... + w_k \geq r$.
4. Erzeuge ein (gültiges) Individuum K_i mit dem Bitstring G als (evtl. zunächst ungültigem) Genom und gibt (K_i) zurück.

(Die Anzahl der Eltern s wurde bei allen Versuchen konstant auf 2 gehalten, weil die obige allgemeinere Variante bei dem für die Testläufe eingefrorenen Programmzustand noch nicht implementiert war. In Schritt 3(a)i wurde in der Praxis die Zufallszahl aus dem Intervall $[0, 1]$ gezogen und im sehr unwahrscheinlichen Fall, dass genau 0 herauskommt, E_1 für dieses Bit gewählt.)

Schritt 1 benötigt (bei den beiden implementierten Varianten) $O(s)$ Zeit. Schritt 2 benötigt $O(m)$ Zeit für die Speicherreservierung. Die Schleife ab Schritt 3 wird m mal ausgeführt. Die Überprüfung und Zuweisung in Schritt 3a benötigen $O(s)$ Zeit. Schritt 3(a)i benötigt konstante Zeit. Das finden eines k mit der geforderten Eigenschaft benötigt in Schritt 3(a)ii $O(s)$ Zeit, denn es muss nur Summand für Summand aufsummiert und geprüft werden, ob die bisherige Summe die Eigenschaft erfüllt. Schritt 4 benötigt $O(t_{erzeugenInd})$ Zeit; die Zeit für die Reparatur der Individuen ist hier schon inbegriffen. Zusammen ergibt sich:

$$\begin{aligned} t_{rek} &\in O(s) + O(m) + m \cdot (O(s) + O(1) + O(s)) + O(t_{erzeugenInd}) \\ &= O(m \cdot (s + t_{erzeugenInd})). \end{aligned}$$

Die Rekombination einer gesamten Elternpopulation zu einer Kindpopulation der Größe n hat dann die Zeitabschätzung:

$$O(n \cdot t_{rek}).$$

Selektion: Die lineare rangbasierte Selektion ist der nach der Definition von Karsten Weicker [Weicker2002] implementiert und arbeitet nach dem folgenden Algorithmus:

Algorithmus (Lineare rangbasierte Selektion)

Eingabe: Population $P = (I_1, ..., I_n)$, Selektionsgröße $k \in \mathbb{N}$.

Rückgabe: Population $P_{sel} = (I_{j_1}, ..., I_{j_k})$ mit $j_i \in \{1, ..., n\}$ $\forall i \in \{1, ..., k\}$.

Sei S eine Menge, implementiert als Array der Länge k.

1. $S := \varnothing$.
2. *Übernahme gesperrter Individuen zu S.*
3. Sortiere P absteigend bezüglich $\succeq$ (das beste Indiviuum ist also vorne). Sei die sortierte Population: $P' := (I'_1, ..., I'_n)$.
4. Für alle $i \in \{1, ..., n\}$ sei $w_i := \frac{2}{r}\left(1 - \frac{i-1}{r-1}\right)$ die Wahrscheinlichkeit für I'_i, selektiert zu werden.
5. Solange $|S| < k$ tue folgendes:
   (a) $r := rand(]0, 1])$.
   (b) Setze $S := S \cup \{I'_k\}$ mit $w_1 + ... + w_{k-1} < r$ und $w_1 + ... + w_k \geq r$.
6. Sei o. B. d. A. $S = \{I_{S_1}, ..., I_{S_k}\}$.
   Gib $P_{sel} := (I_{S_1}, ..., I_{S_k})$ zurück.

(In Schritt 2 werden die gesperrten, also die als „elitär“ gekennzeichneten Individuen der

Population übernommen. Dieser Schritt benötigt $O(n)$ Zeit. Die Anzahl der gesperrten Individuen darf k nicht überschreiten. In Schritt 5a wurde in der Praxis die Zufallszahl aus dem Intervall $[0, 1]$ gezogen und im unwahrscheinlichen Fall $r = 0$ in diesem Schleifendurchlauf kein Individuum hinzugefügt. Theoretisch ergibt sich daher eine Laufzeit von $O(\infty)$.)

Schritt 1 benötigt $O(k)$ Zeit zur Speicherreservierung. Die Sortierung in Schritt 3 benötigt $O(n \cdot \log(n) + n \cdot t_{bewertung})$ Zeit. Die Berechnung der w_i in Schritt 4 benötigt $O(n)$. Die Schleife ab Schritt 5 wird höchstens k mal durchlaufen. Schritt 5a benötigt konstante Zeit. Schritt 5b benötigt (analog zu Schritt 3(a)ii bei der Rekombination) $O(n)$ Zeit. Schritt 6 benötigt konstante Zeit, da die Menge Sel keine komplizierten Mengenoperationen braucht und deswegen schon als Liste implementiert ist. Zusammen ergibt sich:

$$\begin{aligned} t_{selR} &\in O(k) + O(n) + O(n \cdot \log(n) + n \cdot t_{bewertung}) + O(n) + O(k) \cdot (O(1) + O(n)) + O(1) \\ &= O(n \cdot (\log(n) + t_{bewertung})) + O(k \cdot n). \end{aligned}$$

Da gewöhnlich $k = n$ gilt, ergibt sich eine quadratische Laufzeit in Abhängigkeit von n. Diese fällt bei konstantem $n \approx 400$ jedoch nicht weiter ins Gewicht. Wird diese Selektion für eine Plusstrategie als Umweltselektion verwendet, gilt $n = 2k$, da die Population doppelt so groß ist wie zu Beginn des Zyklus und wieder auf die ursprüngliche Größe gebracht werden soll. Auch dabei ergibt sich erfahrungsgemäß keine bemerkbare Laufzeitverschlechterung; die im Anhang bei den Testläufen notierten Laufzeiten dienen dafür als Beleg.

Die Turnierselektion ist ebenfalls nach der Definition von Karsten Weicker [Weicker2002] implementiert und arbeitet nach dem folgenden Algorithmus:

Algorithmus (Turnierselektion)

Eingabe: Population $P = (I_1, ..., I_n)$, Selektionsgröße $k \in \mathbb{N}$, Turniergröße $t \in \mathbb{N}$.

Rückgabe: Population $P_{sel} = (I_{j_1}, ..., I_{j_k})$ mit $j_i \in \{1, ..., n\}$ $\forall i \in \{1, ..., k\}$.

Seien S, T Mengen, beide als Arrays der Länge k bzw. t implementiert.

1. $S := \varnothing$.
2. *Übernahme gesperrter Individuen zu S.*
3. Solange $|S| < k$ tue folgendes:
   (a) $T := \varnothing$.
   (b) Solange $|T| < t$ tue folgendes:
      i. $r := rand(\{1, ..., n\})$.
      ii. $T := T \cup \{I_r\}$.
   (c) $S := S \cup \{I | I$ ist das Individuum mit der besten Bewertung in $T\}$.
4. Sei o. B. d. A. $S = \{I_{S_1}, ..., I_{S_k}\}$.
   Gib $P_{sel} := (I_{S_1}, ..., I_{S_k})$ zurück.

(Wie zuvor benötigt Schritt 2 $O(n)$ Laufzeit und die Anzahl gesperrter Individuen darf k nicht überschreiten.)

Schritt1 benötigt $O(k)$ Zeit für die Reservierung des Speichers. Die Schleife ab Schritt 3 wird höchstens k mal durchlaufen. Schritt 3a benötigt $O(t)$ Zeit zur Reservierung des Speichers bzw. zum Löschen des Arrays ab dem zweiten Schleifendurchlauf. Die Schleife ab Schritt 3b wird genau t mal durchlaufen. Schritt 3(b)i und 3(b)ii benötigen konstante Zeit. Schritt 3c benötigt $O(t) \cdot t_{bewertung}$ Zeit; dabei ist jedoch zu beachten, dass die Bewertung insgesamt höchstens n mal berechnet werden muss, da sie sich bei erneuter Abfrage nicht verändert haben kann; spätestens nach n Berechnungen benötigt sie also konstante Zeit. Schritt 4 ist wie bei der rangbasierten Selektion in konstanter Zeit möglich. Es ergibt sich insgesamt die Laufzeit:

$$\begin{aligned} t_{selTurnier} &\in O(k) + O(n) + O(k) \cdot (O(1) + t \cdot (O(1) + O(1)) + O(t) \cdot t_{bewertung}) + O(1) \\ &= O(n) + O(k \cdot t \cdot t_{bewertung}). \end{aligned}$$

Weil $t_{bewertung}$ höchstens n mal berechnet werden muss, ergibt sich auch die (bei $k = n$ bessere) Abschätzung:

$$t_{selTurnier} \in O(n) + O(n \cdot t_{bewertung} + k \cdot t) = O(n \cdot t_{bewertung} + k \cdot t).$$

Da t in den Versuchen immer auf $0,05n$ gesetzt wurde und meist $k = n$ galt, ergibt sich auch hier eine quadratische Laufzeit in Abhängigkeit von n, was bei konstantem $n \approx 400$ aber ohne Bedeutung ist.

Sperren: Das Sperren der Individuen benötigt in der implementierten Variante, wo die besten k Individuen gesperrt werden,

$$t_{sperren} \in O(n \cdot t_{bewertung}) + O(n \cdot \log(n)) + O(n) = O(n \cdot (t_{bewertung} + \log(n)))$$

Zeit. Dies ergibt sich aus einer (möglicherweise notwendigen) Sortierung der Individuen der Population nach ihrer Fitness und dem Durchlauf durch alle Individuen, um das *sperren*-Flag zu setzen. (Individuen, die nicht gesperrt werden, werden entsperrt, daher muss dabei jedes Individuum betrachtet werden.) Falls das Sperren nach einer Operation stattfindet, bei der die Bewertung des Individuums berechnet wurde, muss die Bewertung nicht erneut berechnet werden. In diesem Fall ergibt sich die Komplexität $O(n \cdot \log(n))$. Das trifft bei den Testläufen auf alle Sperr-Operationen außer der nach der Mutation zu.

Startpopulationserzeugung: Die randomisierte Startpopulationserzeugung (Variante a) benötigt die Zeit

$$t_{startpopRand} \in O(n \cdot t_{erzeugenInd}).$$

Es werden nämlich n Individuen erzeugt, indem als initialer Bitstring 0^m genommen und durch Reparieren zum Genom des Individuums gemacht wird. Alle dafür erforderlichen Schritte sind in $t_{erzeugenInd}$ berücksichtigt.

Die Startpopulationserzeugung nach Beasley et al. (Variante b) arbeitet nach dem auf Seite 34 beschriebenen Algorithmus.

Der gesamte Algorithmus wird n mal ausgeführt. Schritt 1 benötigt konstante Zeit. Die Schleife ab Schritt 2 wird $|M|$ mal durchlaufen. Schritt 2a benötigt für die Erzeugung der Menge α_{ik} und das Ziehen eines zufälligen Elements aus dieser Menge $O(k)$ Zeit, denn die Index-Teilmengen sind schon durch ihre Zuordnung zu den Bitstellen im Genom implizit

nach ihren Kosten sortiert. Schritt 2b benötigt die Zeit $T_{hashHinzu}$ für das Hinzufügen eines Elements zu einem HashSet von Java. Die Schritte 3 und 4 benötigen zusammen die Zeit für $t_{erzeugenInd}$ – und tatsächlich werden sie im Programm auch durch einen entsprechenden Aufruf des Konstruktor der Klasse Individuum ersetzt. Zusammen ergibt sich:

$$\begin{aligned} t_{startpopBeasley} &\in n \cdot (O(1) + |M| \cdot (O(k) + T_{hashHinzu}) + t_{erzeugenInd}) \\ &= n \cdot |M| \cdot (O(k) + T_{hashHinzu}) + n \cdot t_{erzeugenInd}. \end{aligned}$$

Der Parameter k wurde in allen Versuchen auf 5 gesetzt, wie es auch in der Arbeit von Beasley et al. empfohlen wird.

Terminierung: Die Terminierungsbedingung kann in konstanter Zeit überprüft werden:

$$t_{terminierung} \in O(1).$$

Gesamtkomplexität: Die Gesamtkomplexität eines Laufs des evolutionären Algorithmus mit $g \in \mathbb{N}$ Generationen ergibt sich aus:

$$t_{ges} = t_{startpop[X]} + g \cdot \left(t_{div} + t_{mut[Y]} + t_{rek} + t_{sel[Z_E]} + t_{sel[Z_U]} + t_{terminierung}\right).$$

Dabei muss für $[X], [Y], [Z_E]$ und $[Z_U]$ jeweils die Variante des entsprechenden Operators eingesetzt werden.

Wegen der Vielzahl an Unbekannten, die sich vor allem aus der Benutzung des Frameworks ergeben, soll hier darauf verzichtet werden, eine genaue Zeitabschätzung für die Läufe anzugeben, die als Testläufe durchgeführt wurden. Von den Unbekannten absehend kann jedoch festgestellt werden, dass die Reparatur mit einer quadratischen Abschätzung in der Problemgröße $|T| = m$ asymptotisch an der Spitze des Zeitverbrauchs der Testläufe liegt. Das wurde auch durch Profiler-Läufe bestätigt. Dabei ist berücksichtigt, dass die Populationsgröße konstant gehalten wurde – sonst müsste auch die rangbasierte Selektion angegeben werden, die quadratische Laufzeit in der Populationsgröße n hat.

3.4 Ausstehende Verbesserungen und Weiterentwicklungen

Die vorgeschlagenen Verbesserungen unterteilen sich in die Bereiche Funktionalität und Zeitkomplexität.

3.4.1 Weiterentwicklung der Funktionalität

Beim evolutionären Algorithmus wurden folgende mögliche Erweiterungen überlegt:

- Die Speicherung des besten Genoms, welches irgendwann im Lauf des evolutionären Zyklus existiert hat. Bisher wird nur das beste von den Genomen gespeichert, die irgendwann *in der Population* existiert haben (dieses wird als Lösung zurückgegeben). Wird Baldwin-Lernen angewandt, können aber Genome als Nachbarn entstehen, die in der Population niemals auftauchen. Von diesen wird bisher nur die beste Bewertung gespeichert, was für Testläufe und Untersuchungen reicht, in der Praxis aber unbefriedigend ist, weil die Möglichkeit besteht, dass irgendwann eine bessere Lösung existiert

hat, aber verworfen wurde, weil sie nicht zur Population gehörte. In diesem Zusammenhang wären auch Untersuchungen interessant, die überprüfen, wie oft es vorkommt, dass Individuen der Population miteinander in Nachbarschaftsbeziehung stehen.

- Lamarcksche Evolution. Der Frage, in welcher Hinsicht sich im Fall des SCP Lamarcksche Evolution und Evolution mit Baldwin-Lernen unterscheiden, könnte durch wenig Implementierungs-Aufwand nachgegangen werden.

- Altersabhängige Lern-Rechenzeit für verschiedene Individuen. Da das Baldwinsche Lernen sehr viel Rechenzeit kostet, wäre es denkbar, die Lernzeit für Individuen von ihrem evolutionären Erfolg abhängig zu machen. Je länger ein Individuum „überlebt“, also nicht von der Selektion ausselektiert wird, desto mehr Rechenzeit erhält es zum Lernen. Die Begründung ist, dass ein Individuum, das mehrfach nicht ausselektiert wird, ein relativ gutes und möglicherweise noch verbesserbares Genom zu besitzen scheint.

- Die dritte Variante der Startpopulationserzeugung (siehe Abschnitt 3.1.2). Diese Variante wurde zwar schon implementiert, sollte aber ausführlich getestet und vor allem mit der zweiten Variante verglichen werden. Erwartungsgemäß sollte die Startpopulation mit der neuen Variante zwar eine i. A. bessere Fitness haben, möglicherweise jedoch leichter in lokalen Minima hängenbleiben.

3.4.2 Verbesserungen der Zeitkomplexität

Im Lauf der Studienarbeit wurde an der Schnittstelle zwischen dem evolutionären Algorithmus und dem Framework von Dietmar Lippold viel verbessert und die an dieser Stelle verbrauchte Zeit erscheint inzwischen etwa „dem Problem angemessen“, bzw. nicht mehr ohne erheblichen Aufwand optimierbar. Bei den Elementen des evolutionären Algorithmus gibt es jedoch einige, die quadratischen Aufwand in einer der Eingangsgrößen haben.

Während die Selektionen, die eine quadratische Zeitabschätzung in $n = |Pop|$ haben, nicht gravierend sind, solange n ungefähr konstant gehalten wird, ist die stark von $m = |T|$ und $|M|$ abhängige Reparatur möglicherweise problematisch. Als Alternative könnte beispielsweise der zweite Teil der Reparatur weggelassen werden, bei dem unnötige Index-Teilmengen entfernt werden. Dadurch ist die Zeitabschätzung zumindest nicht mehr quadratisch in m. Kleinere Tests mit dieser einfacheren Reparatur brachten aber sehr schlechte Ergebnisse, was darauf zurückzuführen ist, dass sich der Suchraum dabei je nach Dichte des Problems extrem vergrößern kann. Bei alternativen Reparaturverfahren werden also vermutlich auch die anderen Operatoren und Parameter angepasst werden müssen. Eine weitere Möglichkeit wäre, die Reparatur ganz wegzulassen und auch Individuen zuzulassen, die nicht alle Indizes überdecken. Dann müsste man auf jeden Fall eine neue Bewertung entwerfen.

4 Testläufe und Auswertungen

Die Testläufe gliedern sich in vier Gruppen, deren exakte Durchführung und detaillierte Ergebnisse in Anhang A dargestellt sind. Dort findet sich auch eine Beschreibung der Dichte, Größe, etc der verwendeten Testprobleme. Im aktuellen Abschnitt soll eine Bewertung der Ergebnisse vorgenommen und ihre Konsequenzen für die Parameter- und Operatorenwahl diskutiert werden.

Es wurden folgende Gütewerte zu jedem Lauf ausgewertet (die bei Läufen ohne Baldwin-Lernen gleich sind):

a) Zum Ergebnis jedes Laufs die Bewertung des besten Genoms, das irgendwann in der Population existiert hat, also der beste Wert $\kappa(dec(I.G))$, der innerhalb des gesamten Laufs vorgekommen ist.

   Bei Läufen mit aktiviertem Baldwin-Lernen kann dieser Wert schlechter sein als die beste „echte“ Bewertung $I.F$, die in der Population vorgekommen ist, weil sie von Genomen stammen kann, die nicht selbst in der Population, sondern nur Nachbarn eines Genoms der Population waren.

b) Bei der Auswertung von Läufen, in denen Baldwin-Lernen aktiviert war, zusätzlich die beste „echte“ Bewertung $I.F$, die sich irgendwann in der Population befand.

Jeder beschriebene Lauf besteht aus 10 Einzeldurchläufen des evolutionären Algorithmus mit unterschiedlichen Zufallsgenerator-Initialisierungen. Vor jedem Lauf wurde ein Referenzwert durch den Iterated-Greedy-Algorithmus von Dietmar Lippold ermittelt; falls dieser Wert im evolutionären Zyklus erreicht wurde, wurde spätestens 200 Generationen nach dem Zeitpunkt des Erreichens die Terminierung eingeleitet (eine Ausnahme stellt der Langzeittestlauf dar, wo dieser Wert auf 1000 festgelegt wurde).

4.1 Die ersten 32 Testläufe

Aufgrund von Erfahrungswerten bei früheren kleineren Testläufen wurden bei der ersten Gruppe von Testläufen die meisten Parameter und Operatoren konstant gehalten, um die Wirkung einiger weniger auf die Güte der Ergebnisse zu untersuchen. Die zu untersuchenden variablen Parameter und Operatoren wurden in fünf Gruppen eingeteilt, von denen jede genau zwei Einstellmöglichkeiten bot. Die Stellgruppen waren:

1. Lösen von Unicost-Problemen mit maximal 200 Generationen vs. Lösen von Multicost-Problemen mit maximal 800 Generationen,
2. Komma-Strategie vs. Plus-Strategie,
3. Nichtelitäre Strategie vs. elitäre Strategie durch Sperren der besten 4% der Individuen,
4. Beasley-Startpopulationserzeugung (Parameter $k = 5$) vs. randomisierte Startpopulationserzeugung und
5. Baldwin-Lernen mit Berücksichtigung von 3 Individuen aus der Mutationsnachbarschaft vs. kein Baldwin-Lernen.

Alle möglichen Kombinationen aus diesen Gruppen ergaben die 32 Testläufe.

4.1.1 Auswertung der Stellgruppe 1 (Unicost vs. Multicost)

Die genauen Beschreibungen der untersuchten Unicost- bzw. Multicost-Probleme finden sich in Anhang A.

Die Tabellen 1 und 2 (die Läufe entsprechen den Tabellen 40 und 41 in Anhang A) zeigen die Lösungsbewertung der 10 durchgeführten Läufe ohne Balwin-Lernen, mit elitärer Kommastrategie und Beasley-Startpopulationserzeugung, einmal für Unicost-Probleme und einmal für Multicost-Probleme. Diese beiden Läufe sollen exemplarisch betrachtet werden.

In der Spalte „Ref.“ steht der Referenzwert. Die Ergebnisse der einzelnen Läufe sind in den Spalten „L1“ bis „L10“ eingetragen. Der beste der 10 Läufe ist jeweils fettgedruckt, der schlechteste kursiv (falls es mehrere beste oder schlechteste gibt, ist nur einer markiert). In der vorletzten Spalte steht der Durchschnitt ∅ über die 10 Läufe, in der letzten die Standardabweichung s.

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	∅	s
scpclr10	25	27	**25**	25	25	27	25	25	25	27	*27*	25,8	1,0
scpclr11	27	27	25	27	**23**	27	28	*28*	27	23	25	26	1,8
scpclr12	23	**23**	26	29	29	27	28	*30*	28	29	29	27,8	1,9
scpclr13	26	30	29	*31*	28	29	29	29	29	29	**26**	28,9	1,2
scpcyc06	62	61	62	62	62	*63*	61	61	62	61	**60**	61,5	0,8
scpcyc07	144	**148**	156	*156*	154	156	153	155	152	153	155	153,8	2,4
scpcyc08	346	*380*	**370**	375	374	376	376	377	373	372	373	374,6	2,7
scpe1	5	**5**	5	5	5	5	5	5	5	5	*5*	5	0

Tabelle 1: Testlauf – Komma-Strategie, elitär, ohne Baldwin, Beasley-Startpop., Unicost

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	∅	s
scp401	430	459	**447**	459	466	520	*520*	471	466	460	493	476,1	24,6
scp501	253	306	**284**	317	*332*	298	294	301	323	309	311	307,5	13,5
scp61	138	164	158	168	*184*	159	162	**146**	157	162	142	160,2	10,9
scpa1	255	327	332	315	**292**	317	295	334	*337*	325	299	317,3	15,9
scpb1	69	87	94	96	84	80	**77**	80	84	*96*	90	86,8	6,6
scpc1	227	320	336	327	335	346	352	360	328	*378*	**311**	339,3	19,0
scpd1	60	77	76	**67**	67	82	82	81	81	*82*	74	76,9	5,6
scpnre1	29	34	**33**	34	33	35	34	34	*36*	34	35	34,2	0,9
scpnrf1	14	15	15	15	**14**	15	14	14	14	15	*15*	14,6	0,5
scpnrg1	176	268	267	257	271	268	273	270	**256**	*277*	269	267,6	6,2

Tabelle 2: Testlauf – Komma-Strategie, elitär, ohne Baldwin, Beasley-Startpop., Multicost

Wie man sieht, konnte für alle Unicost-Probleme, außer zwei, der Referenzwert bei mindestens einem Lauf erreicht oder verbessert werden. Bei den Multicost-Problemen konnte der Referenzwert dagegen nur für ein einziges Problem (scpnrf1) erreicht werden.

Dieser Effekt zieht sich durch alle Auswertungen dieser 32 Testläufe und wird noch deutlicher bei Läufen, wo statt der Beasley-Startpopulationserzeugung die randomisierte verwendet wurde (siehe auch Abschnitt 4.1.4). Auch frühere Tests zeigten schon, dass die Unicost-Probleme im Allgemeinen gut abschnitten, bei Multicost-Problemen aber keine guten Ergebnisse erzielt werden konnten. Die Ursache dafür wurde zunächst darin vermutet, dass die Multicost-Probleme einerseits größer sind (in der Anzahl der Index-Teilmengen $|T|$) und andererseits per se ein schwierigeres Problem darstellen als die Unicost-Problemen.

Während der Testauswertungen wurde jedoch eine weitere Ursache gefunden, die aus einem Denkfehler bei der Implementierung der Mutation entstanden ist: die Mutationsrate wurde linear steigend mit der Genomlänge implementiert. Während es logisch erscheint, dass die Mutationsrate mit zunehmender Genomlänge ansteigen soll, weil bei zunehmender Länge

des Genoms ein einzelner Bitflip weniger ins Gewicht fällt, muss zusätzlich auch die Dichte des Problems berücksichtigt werden. Von der Dichte hängt die Anzahl der Bits ab, die in den Genomen durchschnittlich auf 1 gesetzt sind. Da sie bei den getesteten Problemen nicht konstant ist, kann sie nicht unberücksichtigt bleiben.

Wenn beispielsweise bei einem Genom der Länge 400 etwa 50 Bits auf 1 gesetzt sind und eine Mutation mit einer Rate von $0,2\%$ der Genomlänge durchgeführt wird, dann werden 8 Bits geflippt, was keine zu große Mutation darstellt. (Dabei muss man berücksichtigen, dass durch die Reparatur evtl. einige dieser Änderungen rückgängig gemacht werden, weil vorrangig Nullen zu Einsen geflippt wurden.) Würde aber dieselbe Mutation auf ein Genom der Länge 10000 angewendet (so groß sind Genome des größten Problems der Testläufe: scpnrg1), welches dennoch nur 50 Bits auf 1 gesetzt hat, dann würden 200 Bits mutiert; mit großer Wahrscheinlichkeit würden fast ausschließlich Nullen zu Einsen flippen. Zumindest intuitiv ist es einleuchtend, dass daraufhin oft viele Index-Teilmengen der Lösung nicht mehr notwendig wären, was es der Reparatur erlauben würde, fast völlig wahllos Einsen zu Nullen zu machen. Das entstehende Genom hätte also kaum noch Ähnlichkeit mit dem ursprünglichen.

Der genaue Einfluss der Dichte auf die Anzahl der Einsen im Genom, sowie die Art der Konsequenzen, die man für die Mutationsrate daraus ziehen sollte, konnte aus Zeitgründen nicht genauer untersucht werden. Es kann aber in den 32 Testläufen beobachtet werden, dass die linear steigende Mutationsrate tendenziell für Probleme mit geringer Dichte nicht geeignet ist. So haben die Probleme *scpcyc07* und *scpcyc08* die geringste Dichte aller Unicost-Probleme; das sind aber gerade die Unicost-Probleme, bei denen fast nie der Referenzwert erreicht werden konnte (tatsächlich wurde er innerhalb der 32 Läufe nur ein einziges Mal für scpcyc07 erreicht). Bei den Multicost-Problemen haben die Probleme *scpnre1* und *scpnrf1* die höchste Dichte; ihre Referenzwerte wurden von allen Multicost-Problemen am häufigsten erreicht. Zu beachten ist dabei, das *scpnre1* und *scpnrf1* mit $|T| = 5000$ zu den größten der Multicost-Probleme zählen. Eine Ausnahme stellt *scp401* dar: bei ihm konnte, trotz der geringen Dichte von $2,0\%$, bei dem in Tabelle 43 dargestellten Lauf der Referenzwert sogar verbessert werden.

Weitere Testläufe mit kleinerer Mutationsrate konnten den Verdacht bestätigen, dass die ursprüngliche Mutationsrate für die meisten getesteten Multicost-Probleme zu groß war (siehe Abschnitte 4.3 und 4.4).

Bei den folgenden Auswertungen (der ersten 32 Testläufe) muss beachtet werden, dass sich diese zu große Mutationsrate vor allem negativ auf die Läufe mit aktiviertem Baldwin-Lernen ausgewirkt haben müsste. Die Mutationsnachbarschaft wurde hier mit derselben Mutationsrate berechnet. Die Läufe mit Baldwin-Lernen entsprechen den Tabellen 44, 45, 46, 47, 52, 53, 54, 55, 60, 61, 62, 63, 68, 69, 70 und 71.

4.1.2 Auswertung der Stellgruppe 2 (Komma vs. Plus)

Zwischen den Testläufen mit Komma-Strategie und den Testläufenn mit Plus-Strategie wurden t-Tests durchgeführt. Dabei wurde von je zwei Läufen, die sich nur in der Wahl der Plus- bzw. Komma-Strategie unterschieden, alle 10 Ergebnisse von jedem getesteten Problem des einen Laufs mit den 10 Ergebnissen des gleichen Problems des anderen Laufs verglichen. Dabei wurden Unicost- und Multicost-Läufe natürlich separat betrachtet. Wenn ein Signifikanzniveau von $2,5\%$ unterschritten wurde, wurden die Auswertungen dieses einen Problems

in den entsprechenden beiden Läufen als signifikant unterschiedlich angesehen. Die Vergleiche fanden jeweils zwischen den Gütewerten des oben beschriebenen Typs a) statt, also der besten Bewertung eines Genoms der Population ($\kappa(dec(I.G))$ für das beste Genom G, das in der Population während des Laufs existiert hat).

Die Vergleiche werden in Tabellen dargestellt, die wie folgt zu interpretieren sind:

- In der ersten Zeile stehen die Nummern der jew. zwei Tabellen aus Anhang A, deren Werte für den Vergleich der jeweiligen Spalte verwendet wurden.
- In der zweiten Zeile steht die Anzahl der Probleme, die von der ersten Variante signifikant besser gelöst wurde, als positive Zahl und die Anzahl der Probleme, die von der zweiten Variante signifikant besser gelöst wurde als negative Zahl. Falls es innerhalb eines Laufs Probleme gab, die besser gelöst wurden, und solche, die schlechter gelöst wurden, dann werden die Zahlen nicht addiert, sondern beide getrennt in der Tabelle verzeichnet. Falls keine signifikanten Unterschiede aufgetreten sind, wird in der betreffenden Zelle 0 verzeichnet.
- In der letzten Spalte werden alle signifikanten Unterschiede addiert (wieder positive und negative Werte separat).

In der folgenden Tabelle bedeuten negative Werte signifikant bessere Ergebnisse der Komma-Strategie und positive Werte signifikant bessere Ergebnisse der Plus-Strategie.

Unicost-Läufe Bei allen Vergleichen, wo mindestens eine signifikante Abweichung gefunden wurde, wurde bei dem Problem *scpcyc08* eine signifikante Abweichung gefunden. Beim dritten Vergleich wurde zusätzlich eine signifikante Abweichung beim Problem *scpclr11* gefunden.

Vergleich (Tab)	40,56	42,58	44,60	46,62	48,64	50,66	52,68	54,70	alle
Sign. Untersch.	0	0	-2	0	-1	-1	-1	-1	-6

Tabelle 3: t-Test – Komma vs. Plus, Unicost

Von den insgesamt 64 einzelnen Läufen (8 Unicost-Probleme mit jeweils 8 Parameter- und Operatoren-Einstellungen) wurden 6 Läufe von der Komma-Strategie besser gelöst und keines von der Plus-Strategie. Im Allgemeinen scheint zwischen der Plus- und der Komma-Strategie bei Unicost-Problemen kein wesentlicher Unterschied zu bestehen. Die Komma-Strategie ist wohl ein wenig besser.

Multicost-Läufe Bei Multicost-Problemen scheint die Komma-Strategie etwas besser zu sein als die Plus-Strategie. Es wurden von insgesamt 80 einzelnen Läufen (10 Multicost-Probleme mit 8 verschiedenen Einstellungen) 22 von der Komma-Strategie signifikant besser gelöst, 8 von der Plus-Strategie und bei den übrigen 50 Läufen konnte keine Signifikanzunterscheidung festgestellt werden.

Vergleich (Tab)	41,57	43,59	45,61	47,63	49,65	51,67	53,69	55,71	alle
Sign. Untersch.	-6	-7	-3	0	+1	+6	+1	-6	-22+8

Tabelle 4: t-Test – Komma vs. Plus, Multicost

Bei Problemen mit aktiviertem Baldwin-Lernen wurden auch die Werte, die dem besten Wert $I.F$ der Population entsprachen (die oben beschriebenen Gütewerte vom Typ b) auf Signifikanz-Unterschiede untersucht. Es ergab sich dabei ein ähnliches Bild wie mit den Gütewerten vom Typ a), die aufgelistet sind. Vor allem wich kein Problem bei einem der beiden Typen von Gütewerten in die andere Richtung ab als beim anderen. Auf die Darstellung der Vergleiche des Typs b) wird hier aus Gründen der Übersichtlichkeit verzichtet.

4.1.3 Auswertung der Stellgruppe 3 (nichtelitär vs. elitär)

Für diese Auswertung wurden t-Tests wie im vorigen Abschnitt durchgeführt, nur diesmal zwischen Testläufen mit elitärer und Testläufen mit nichtelitärer Strategie.

Negative Werte in der Tabelle bedeuten, dass die elitäre Strategie besser war, positive, dass die nichtelitäre Strategie besser war.

Unicost-Läufe Alle Vergleiche fielen bei den Unicost-Läufen zugunsten der nichtelitären Strategie aus.

Vergleich (Tab)	40,48	42,50	44,52	46,54	56,64	58,66	60,68	62,70	alle
Sign. Untersch.	+1	+1	+5	+5	+1	+1	+6	+7	+27

Tabelle 5: t-Test – Nichtelitär vs. Elitär, Unicost

Insgesamt wurden von 64 Problemen 27 von der nichtelitären Strategie besser gelöst und keines von der elitären Strategie. Jedesmal war das Problem *scpcyc08* unter den signifikant besser gelösten.

Multicost-Läufe Die Multicost-Läufe ergeben ein anderes Bild.

Vergleich (Tab)	41,49	43,51	45,53	47,55	57,65	59,67	61,69	63,71	alle
Sign. Untersch.	-10	-10	-4	+10	-10	-10	-1	+10	-45+20

Tabelle 6: t-Test – Nichtelitär vs. Elitär, Multicost

Insgesamt wurden von 80 Problemen 45 von der elitären Strategie besser gelöst und 20 von der nichtelitären Strategie.

Man kann beobachten, dass die Läufe mit aktiviertem Baldwin-Effekt hier stark von den übrigen Werten abweichen. Bei den Unicost-Problemen wurde bei allen 4 Läufen ohne Baldwin-Lernen jeweils nur ein Ergebnis als signifikant unterschiedlich eingestuft, während es bei den Läufen mit Baldwin-Lernen 5, 5, 6 und 7 Ergebnisse waren. Bei den Multicost-Problemen wurden bei allen 4 Läufen ohne Baldwin-Lernen alle 10 Probleme von der elitären Strategie signifikant besser gelöst als von der nichtelitären. Bei den Läufen mit Baldwin wurden dagegen nur 1 bzw. 4 Probleme von der elitären Strategie und zweimal sogar alle 10 Probleme von der nichtelitären Strategie besser gelöst.

Unter Aufschiebung der Interpretation der Ergebnisse mit Baldwin-Lernen kann für Läufe ohne Baldwin-Lernen bei Multicost-Problemen der elitären Strategie der Vorzug gegeben werden. Bei Unicost-Problemen scheinen die beiden Strategien etwa gleich gute Ergebnisse zu liefern.

4.1.4 Auswertung der Stellgruppe 4 (Beasley-Startpop. vs. randomisierte Startpop.)

Auch hier wurden t-Tests wie zuvor ausgeführt, diesmal zwischen den Problemen mit Beasley-Startpopulationserzeugung und denen mit randomisierter Startpopulationserzeugung.

Negative Werte in der Tabelle bedeuten, dass die Beasley-Variante signifikant besser war, positive, dass die randomisierte Variante besser war.

Unicost-Läufe Bei fast allen Versuchen wurde kein einziges Problem durch eines der Verfahren signifikant besser gelöst.

Vergleich (Tab)	40,42	44,46	48,50	52,54	56,58	60,62	64,66	68,70	alle
Sign. Untersch.	0	-1	0	0	0	-1	0	0	-2

Tabelle 7: t-Test – Beasley-Startpop. vs. rand. Startpop., Unicost

Von 64 Problemen wurden 2 von der Beasley-Variante besser gelöst und keines von der randomisierten Variante.

Multicost-Läufe Hier war bei allen Versuchen die Beasley-Variante bei fast allen Problemen besser.

Vergleich (Tab)	41,43	45,47	49,51	53,55	57,59	61,63	65,67	69,71	alle
Sign. Untersch.	-9	-10	-10	-10	-9	-10	-10	-10	-78

Tabelle 8: t-Test – Beasley-Startpop. vs. rand. Startpop., Multicost

Insgesamt wurden von 80 Problemen 78 von der Beasley-Variante signifikant besser gelöst und keines von der randomisierten Variante.

Dass die Beasley-Variante der Startpopulationserzeugung bei den Multicost-Problemen besser war, lässt sich teilweise dadurch erklären, dass die Individuen von Beginn an deutlich bessere Fitness-Werte haben. Die Chance der randomisierten Startpopulation ist, dort aufzuholen, wo die andere Population möglicherweise durch zu spezifische Gene in lokalen Optima stecken geblieben ist. Da bei diesen Versuchen die Mutationsrate für die Multicost-Probleme jedoch offenbar zu hoch eingestellt war, konnte dieser Vorteil wohl nicht zur Geltung kommen. Möglicherweise war auch die Anzahl an Generationen zu gering. Weitere Tests in dieser Richtung waren aus Zeitgründen nicht mehr möglich.

4.1.5 Auswertung der Stellgruppe 5 (Lernen vs. kein Lernen)

Bei der Untersuchung der Unterschiede zwischen Läufen mit Baldwin-Lernen und denen ohne wurden wieder t-Tests durchgeführt. Diesmal wurden die Gütewerte des Typs a) bei den Läufen *ohne* Baldwin-Lernen mit den Gütewerten des Typs a) und mit denen des Typs b) *mit* Baldwin-Lernen verglichen. Die Gütewerte des Typs a) repräsentieren dabei die Veränderung des Genoms durch Lernen der Individuen, wie sie Baldwin zufolge stattfinden sollte. Die Gütewerte des Typs b) sind eher von praktischem Nutzen, sie repräsentieren die beste Lösung, die im Lauf der Evolution überhaupt existiert hat.

Positive Einträge in der Tabelle bedeuten hier, dass die Variante mit Baldwin-Lernen besser war, negative, dass die ohne Baldwin-Lernen besser war.

Vergleiche des Typs a) mit dem Typ a) – Unicost-Läufe Hier schnitt die Berechnung ohne Baldwin-Lernen besser ab als die Berechnung mit Baldwin-Lernen.

Vergleich (Tab)	40,44	42,46	48,52	50,54	56,60	58,62	64,68	66,70	alle
Sign. Untersch.	-3	-2	0	0	-4	-4	0	0	-13

Tabelle 9: t-Test – Lernen (Typ a) vs. kein Lernen, Unicost

Es wurden von 64 Problemen 13 von der Variante ohne Baldwin-Lernen besser gelöst und keines von der Variante mit Baldwin-Lernen.

Vergleiche des Typs a) mit dem Typ a) – Multicost-Läufe Hier schnitt die Variante ohne Baldwin-Lernen sogar noch deutlicher besser ab als die Variante mit Baldwin-Lernen.

Vergleich (Tab)	41,45	43,47	49,53	51,55	57,61	59,63	65,69	67,71	alle
Sign. Untersch.	-4	-10	-1	-8	-1	-10	0	-10	-44

Tabelle 10: t-Test – Lernen (Typ a) vs. kein Lernen, Multicost

Insgesamt wurden von 80 Problemen 44 von der Variante ohne Baldwin-Lernen signifikant besser gelöst und keines von der Variante mit Baldwin-Lernen.

Dieses kontra-intuitive Ergebnis kann teilweise darauf zurückgeführt werden, dass die Gütewerte des Typs a) nicht dem Fitnesswert entsprechen, den die Evolution bei der Selektion (und Rekombination) zu berücksichtigen hatte. Der Fitnesswert war für die Evolution eventuell besser durch einen besseren Nachbarn. Das Ergebnis sieht für den Gütewert b) (s. u.) etwas besser aus, jedoch ist es immer noch, in Anbetracht der um ein Vielfaches höheren Rechenzeit, nicht zufriedenstellend. Das schlechte Abschneiden der Variante mit Baldwin-Lernen muss wohl ebenfalls (bei Multicost-Problemen, bzw. Problemen mit geringer Dichte) auf die zu hohe Mutationsrate zurückgeführt werden (siehe Diskussion unten).

Vergleiche des Typs a) mit dem Typ b) – Unicost-Läufe Hier ergaben fast alle Tests keinen signifikanten Unterschied.

Vergleich (Tab)	40,44	42,46	48,52	50,54	56,60	58,62	64,68	66,70	alle
Sign. Untersch.	0	0	0	0	-1	0	0	0	-1

Tabelle 11: t-Test – Lernen (Typ b) vs. kein Lernen, Unicost

Ein einziges Problem wurde von der Variante ohne Baldwin-Lernen besser gelöst. Dieses Problem war *scpcyc08*, welches die geringste Dichte aller Unicost-Probleme hat.

Vergleiche des Typs a) mit dem Typ b) – Multicost-Läufe Bei diesen Läufen wurden von 80 Problemen 11 von der Variante ohne Baldwin-Lernen besser gelöst und 29 von der Variante mit Baldwin-Lernen.

Vergleich (Tab)	41,45	43,47	49,53	51,55	57,61	59,63	65,69	67,71	alle
Sign. Untersch.	-2	-5+4	+1	+9	0	-3+6	0	-1+8	-11+29

Tabelle 12: t-Test – Lernen (Typ b) vs. kein Lernen, Multicost

Dieses Ergebnis rechtfertigt zumindest tendentiell den Einsatz von mehr Rechenzeit, da die Probleme prinzipiell etwas besser gelöst wurden. Dennoch ist die erwartete 4-fache Rechenzeit (bei 3 berücksichtigten Nachbarn) sehr hoch, verglichen damit, dass zwei Läufe sogar schlechter gelöst wurden (erster und zweiter Vergleich). Weiterhin konnte bei diesen Versuchen die erwartete Verbesserung am Genom selbst, also an den Gütewerten des Typs a), nicht beobachtet werden. Dass die Betrachtung einer größeren Anzahl von Individuen (der Nachbarn) innerhalb des Laufs zu einem besseren Ergebnis des Typs b) führt, ist nicht verwunderlich. Der Baldwin-Effekt sollte jedoch auch in den Genomen der Individuen Verbesserungen auslösen.

Der Grund für diese nicht zufriedenstellenden Ergebnisse liegt vermutlich wieder an der zu großen Mutationsrate. Vor allem bei Multicost-Problemen konnte festgestellt werden, dass sich bei ein und demselben Lauf der Gütewert a) mitunter um ein Vielfaches vom Gütewert b) unterscheiden konnte. Siehe als Extremfall bspw. Tabelle 47, wo bei Problem *scpnrf1* der Gütewert a) bei 750 liegt und der Gütewert b) bei 16. Offensichtlich ist hier der Nachbarschaftsbegriff ohne Nutzen für die Evolution. Die Vermutung liegt sogar nahe, dass sich das Baldwin-Lernen hier negativ auf die Evolution auswirken könnte, weil sich der Selektionsdruck auf „falsche“ Individuen auswirkt: es werden Individuen bevorzugt, deren eigene Position im Suchraum schlecht ist, weil sie einen (weit entfernten) Nachbarn haben, der eine gute Bewertung hat. Die Evolution findet gewissermaßen, über den Umweg der eigentlichen Population, nur noch auf den Nachbarn statt.

4.2 Langzeittestlauf

Der Langzeittestlauf wurde mit den gleichen Parametern durchgeführt wie der zur Tabelle 41 im Anhang gehörige Lauf, nur dass die maximale Generationenzahl auf 8000 festgelegt wurde und die Anzahl der Generationen nach Erreichen des Referenzwerts auf 1000.

Der Lauf folgte also im Allgemeinen folgenden Einstellungen des letzten Abschnitts:

1. Lösen von Multicost-Problemen (allerdings mit 8000 statt 800 Generationen),
2. Komma-Strategie,
3. elitäre Strategie,
4. Beasley-Startpopulation und
5. Kein Baldwin-Lernen.

Diese Einstellungen wurden ausgewählt, weil sie bei den 32 Läufen des vorigen Abschnitts die besten Werte der Multicost-Läufe ohne Baldwin-Lernen lieferten. Die Unicost-Probleme wurden nicht weiter untersucht, weil sie bereits zufriedenstellend gelöst wurden.

Es ergaben sich folgende Werte:

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	∅	s
scp401	430	440	435	*452*	**431**	449	447	436	432	443	434	439,9	7,1
scp501	253	294	281	297	273	295	*311*	273	285	**269**	278	285,6	12,7
scp61	138	154	159	*167*	146	148	151	**138**	162	162	153	154	8,3
scpa1	255	289	273	288	294	291	**271**	274	297	*315*	282	287,4	12,6
scpb1	69	90	**80**	85	93	82	83	87	*96*	93	83	87,2	5,2
scpc1	227	291	311	**269**	272	320	330	*338*	280	310	299	302	22,7
scpd1	60	76	74	73	**67**	*82*	75	69	76	67	70	72,9	4,5
scpnre1	29	**31**	*36*	33	34	33	35	35	33	32	35	33,7	1,5
scpnrf1	14	15	15	15	**14**	14	15	15	15	15	*15*	14,8	0,4
scpnrg1	176	243	253	243	243	247	*284*	247	237	245	**235**	247,7	13,0

Tabelle 13: Testlauf (Lang) – Komma-Strategie, elitär, ohne Baldwin, Beasley-Startpop., Multicost

Anmerkung: Für Problem *scp61* ist das der einzige Lauf, bei dem der Referenzwert einmal erreicht wurde. Das geschah in Generation 4572 des 6. Laufs. Entsprechend wurde der Lauf nach 5572 Generationen abgebrochen. (Siehe auch Abbildung 6).

Zum Vergleich die Werte desselben Versuchs mit der normalen Generationenzahl:

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	∅	s
scp401	430	459	**447**	459	466	520	*520*	471	466	460	493	476,1	24,6
scp501	253	306	**284**	317	*332*	298	294	301	323	309	311	307,5	13,5
scp61	138	164	158	168	*184*	159	162	**146**	157	162	142	160,2	10,9
scpa1	255	327	332	315	**292**	317	295	334	*337*	325	299	317,3	15,9
scpb1	69	87	94	96	84	80	**77**	80	84	*96*	90	86,8	6,6
scpc1	227	320	336	327	335	346	352	360	328	*378*	**311**	339,3	19,0
scpd1	60	77	76	**67**	67	82	82	81	81	*82*	74	76,9	5,6
scpnre1	29	34	**33**	34	33	35	34	34	*36*	34	35	34,2	0,9
scpnrf1	14	15	15	15	**14**	15	14	14	14	15	*15*	14,6	0,5
scpnrg1	176	268	267	257	271	268	273	270	**256**	*277*	269	267,6	6,2

Tabelle 14: Testlauf – Komma-Strategie, elitär, ohne Baldwin, Beasley-Startpop., Multicost

Auch hier wurde ein t-Test zwischen den beiden Tabellen vorgenommen. Die Angabe „Vergleich (Prob)“ bezieht sich diesmal auf die einzelnen Probleme in den Tabellen in der Reihenfolge wie sie aufgelistet sind; 1 steht für scp401, ..., 10 steht für scpnrg1.

Es bedeutet -1, dass der normale Lauf signifikant besser war, +1, dass der Langzeitlauf signifikant besser war, 0, dass keiner der Läufe besser war.

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	+1	0	0	0	+1	0	-1	+1	-1+3

Tabelle 15: t-Test – Langzeit vs. Normal, Multicost

Es wurden 3 Probleme vom Langzeitlauf signifikant besser gelöst und 1 Problem vom normalen Lauf. Das Problem *scpnrg1* (10) ist das größte der getesteten Probleme ($|T| = 10000$) und hat mit $2,0\%$ eine geringe Dichte; es wurde vom Langzeitlauf besser gelöst als vom normalen Lauf. Das einzige Problem, für das der normale Lauf besser war, war *scpnrf1* (9); das ist ein großes Problem ($|T| = 5000$), hat aber eine hohe Dichte ($20,0\%$) und ist erfahrungsgemäß ein eher leichtes Problem für den evolutionären Algorithmus. Bei beiden Läufen

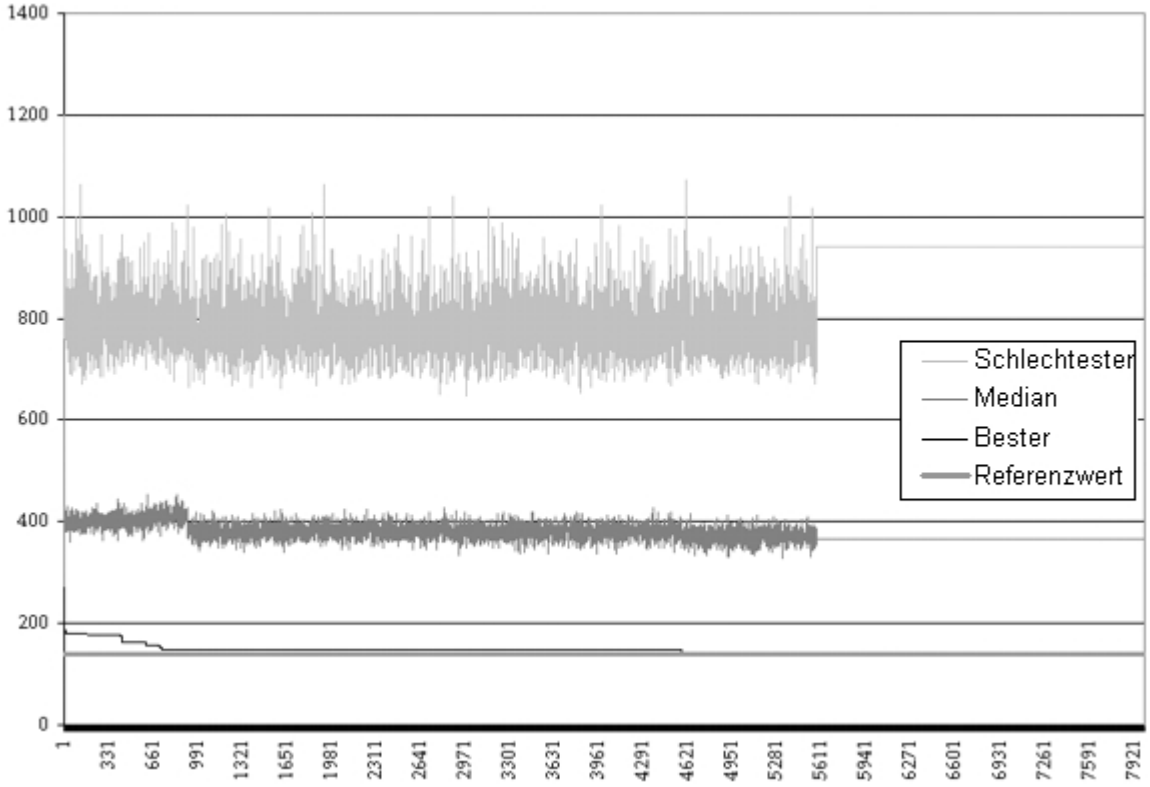

Abbildung 6: *Populationsentwicklung beim besten Langzeittest mit Problem scp61*

wurde bei diesem Problem stets 14 oder 15 als Ergebnis gefunden. Langzeitläufe scheinen also vor allem bei schwierigen Problemen bessere Ergebnisse zu liefern als kürzere Läufe. Die Population ist nach 800 Generationen nicht zu homogen, um noch weitere Verbesserungen erzielen zu können.

An dieser Stelle sollte zum Vergleich noch ein Testlauf mit randomisierter Startpopulationserzeugung durchgeführt werden.

4.3 Testläufe mit kleinerer linear steigender Mutationsrate

Für die hier betrachteten Testläufe wurde die Mutationsrate schrittweise von den bisher verwendeten 2% der Genomlänge verringert. Bei allen Läufen dieses Abschnitts wurde die randomisierte Startpopulationserzeugung verwendet, weil untersucht werden sollte, wie gute Lösungen der evolutionäre Zyklus selbst mit verschiedenen Parametern erzeugt. Die problemspezifischere Beasley-Variante der Startpopulationserzeugung hat einen großen Einfluss auf die Ergebnisgüte und würde vermutlich weniger signifikante Unterschiede zwischen verschiedenen Einstellungen bewirken.

4.3.1 Testläufe ohne Baldwin-Lernen

Abgesehen von der Mutationsrate haben die zu vergleichenden Läufe dieselben Einstellungen wie der zu Tabelle 43 gehörige Lauf. Nach den in Abschnitt 4.1 definierten Stellmöglichkeiten sind das folgende Einstellungen:

- Lösen von Multicost-Problemen,
- Komma-Strategie,
- elitäre Strategie,
- randomisierte Startpopulation,
- kein Baldwin-Lernen.

Es wurden fünf Testläufe gemacht, die untereinander und zusätzlich mit dem bereits vorhandenen Lauf mit Mutationsrate $0,2\%$ verglichen wurden.

Die Mutationsrate unterscheidet sich als einziger Parameter von dem o.a. Lauf. Sie wurde auf folgende Werte gesetzt:

1. 2% (das ist der bereits beschriebene Lauf),
2. 1%,
3. $0,75\%$,
4. $0,5\%$,
5. $0,25\%$,
6. $0,1\%$.

Die Ergebnisse sind im Anhang als Tabellen 43 und 73, 74, 75, 76, 77 bezeichnet.

Es wurden t-Tests zu allen Kombinationen der sechs durchgeführten Läufe gemacht.

Es stellte sich heraus, dass mit kleiner werdender Mutationsrate die Probleme (häufig schon im nächsten Schritt) signifikant besser gelöst wurden. Diese deutliche Tendenz zeigt sich bis Lauf 5, wo ein Optimum zu liegen scheint. Bei Lauf 6 werden die meisten Probleme wieder etwas schlechter gelöst. Beim größten Problem *scpnrg1* ist die Verbesserung am deutlichsten und bei diesem setzt sie sich auch bei einer Rate von $0,1\%$ leicht fort, ist hier aber nicht mehr signifikant. Diese Entwicklung bestätigt die Vermutung, dass die Mutationsrate vor allem bei großen Problemen mit niedriger Dichte bei den früheren Tests zu hoch war.

Da die Darstellung aller t-Tests umfangreich und unüberichtlich ist, werden hier nur t-Tests für die Vergleiche zwischen benachbarten Mutationsraten aufgeführt. „Vergleich (Prob.)“ bezieht sich wieder auf das entsprechende Multicost-Problem. +1 bedeutet, dass der Lauf mit kleinerer Mutation besser war, -1 bedeutet, dass der Lauf mit größerer Mutation besser war, 0 bedeutet, dass kein signifikanter Unterschied gefunden wurde.

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	+1	+1	+1	+1	0	+1	+5

Tabelle 16: t-Test – Mutationsrate: 2% vs. 1%

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	0	0	+1	+1

Tabelle 17: t-Test – Mutationsrate: 1% vs. $0,75\%$

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	+1	0	0	0	0	+1	+2

Tabelle 18: t-Test – Mutationsrate: $0,75\%$ vs. $0,5\%$

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	0	0	+1	+1

Tabelle 19: t-Test – Mutationsrate: $0,5\%$ vs. $0,25\%$

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	-1	0	0	0	-1

Tabelle 20: t-Test – Mutationsrate: $0,25\%$ vs. $0,1\%$

Abbildung 7 zeigt die durchschnittlich erreichten Ergebnisse für alle Multicost-Probleme und verschiedenen Mutationsraten.

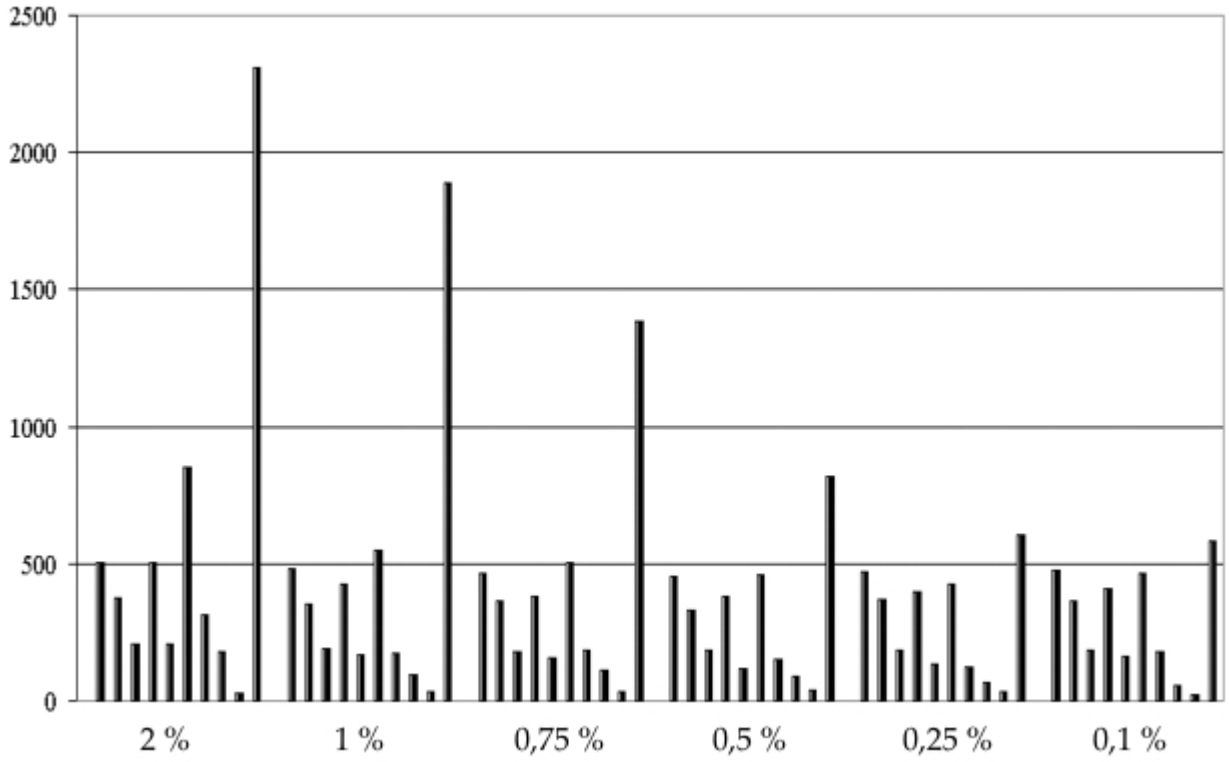

Abbildung 7: *Entwicklung der durchschnittlichen Ergebnis-Güte mit fallender Mutationsrate bei Läufen ohne Baldwin-Lernen. Jeweils 10 gruppierte Säulen stellen die Durchschnitts-Ergebnisse der 10 Multicost-Probleme dar – bei jew. 10 Läufen. Innerhalb der Gruppe sind die Probleme v. l. n. r. wie in der Tabelle v. o. n. u. sortiert. Die Mutationsrate fällt von links nach rechts von* 2% *bis* $0,1\%$ *der Genomlänge. Auf der Y-Achse ist die Fitness aufgetragen.*

4.3.2 Testläufe mit Baldwin-Lernen

Die zu vergleichenden Testläufe haben dieselben Einstellungen (auch Mutationsraten) wie die im vorigen Abschnitt beschriebenen Läufe, nur dass jeweils Baldwin-Lernen aktiviert war. Abgesehen von der Mutationsrate haben die zu vergleichenden Läufe dieselben Einstellungen wie der zu Tabelle 47 gehörige Lauf. Nach den in Abschnitt 4.1 definierten Stellmöglichkeiten sind das folgende Einstellungen:

- Lösen von Multicost-Problemen,
- Komma-Strategie,
- elitäre Strategie,
- randomisierte Startpopulation,
- Baldwin-Lernen mit Berücksichtigung von 3 Nachbarn.

Die Ergebnisse sind im Anhang als Tabellen 47 und 78, 79, 80, 81, 82 bezeichnet.

Es wurden t-Tests zwischen den Ergebnissen der Läufe mit unterschiedlichen Mutationen durchgeführt. Es sollen auch hier nicht alle Kombinationen von Vergleichen, sondern nur die Vergleiche von benachbarten Mutationsraten, dargestellt werden. Die Bedeutung der Tabelle ist dieselbe wie im vorigen Abschnitt.

Bei Vergleichen von Gütewerten des Typs a) wurde eine deutliche Signifikanz der Verbesserung mit fallender Mutationsrate festgestellt, oftmals wurde auch das „sehr signifikante“ Niveau von $0,5\%$ erreicht:

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	+1	+1	+1	+1	+1	+1	+1	+1	0	+1	+9

Tabelle 21: t-Test – Mutationsrate: 2% vs. 1% (Typ a)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	+1	0	0	0	+1

Tabelle 22: t-Test – Mutationsrate: 1% vs. $0,75\%$ (Typ a)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	+1	+1	+1	+1	+1	0	0	+1	+1	0	+7

Tabelle 23: t-Test – Mutationsrate: $0,75\%$ vs. $0,5\%$ (Typ a)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	+1	0	+1	+1	+1	+1	+1	+1	+1	+8

Tabelle 24: t-Test – Mutationsrate: $0,5\%$ vs. $0,25\%$ (Typ a)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+10

Tabelle 25: t-Test – Mutationsrate: $0,25\%$ vs. $0,1\%$ (Typ a)

Wie man sieht, steigt die Qualität der Ergebnisse deutlich mit fallender Mutationsrate an (siehe auch Abbildung 8). Bei diesen Gütewerten des Typs a) wirkt sich eine zu hohe Mutationsrate doppelt verheerend aus, denn die Selektion wurde nicht auf diese Gütewerte, sondern auf die Fitness des besten Nachbarn angewandt.

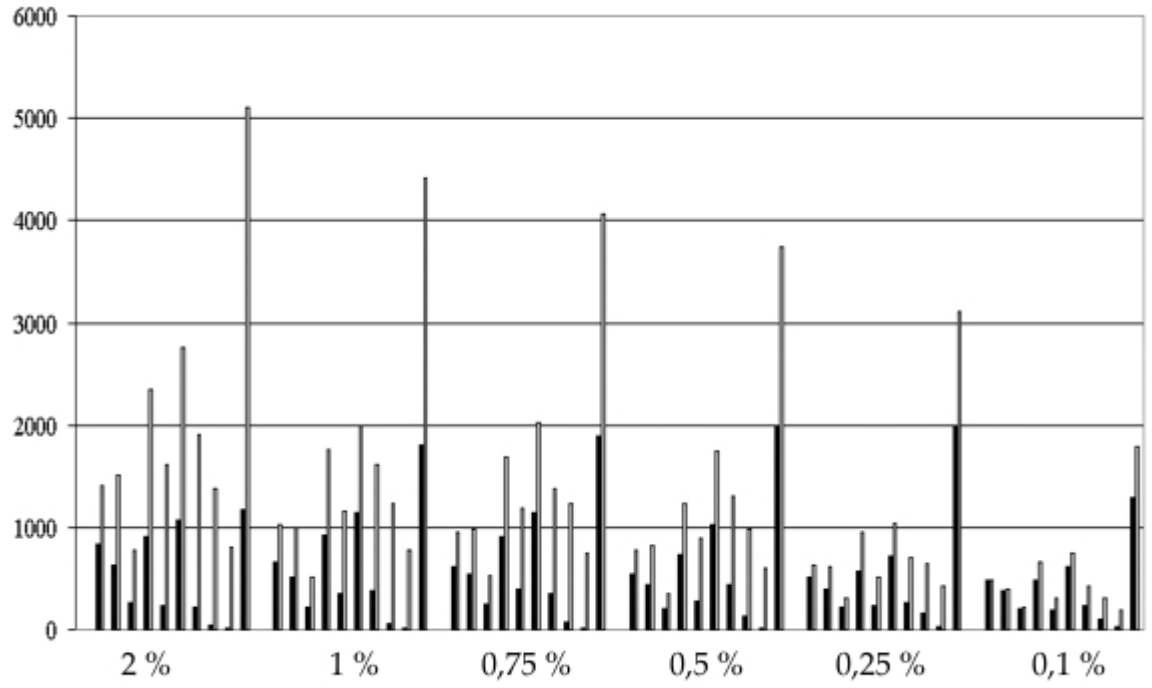

Abbildung 8: *Entwicklung der durchschnittlichen Ergebnis-Güte mit fallender Mutationsrate bei Läufen mit Baldwin-Lernen. Jeweils 10 gruppierte Doppel-Säulen stellen die Durchschnitts-Ergebnisse der 10 Multicost-Probleme dar – bei jew. 10 Einzelläufen. Die helle Säule stellt jeweils den Gütewert des Typs a) dar, die dunkle den Gütewert des Typs b).*

Es folgen die t-Test-Vergleiche der Gütewerte des Typs b):

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	+1	0	0	0	0	0	-1	-1	1	-1	-3+2

Tabelle 26: t-Test – Mutationsrate: 2% vs. 1% (Typ b)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	0	0	0	0

Tabelle 27: t-Test – Mutationsrate: 1% vs. 0,75% (Typ b)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	-1	0	0	-1

Tabelle 28: t-Test – Mutationsrate: 0,75% vs. 0,5% (Typ b)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	0	0	0	0

Tabelle 29: t-Test – Mutationsrate: 0,5% vs. 0,25% (Typ b)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	0	0	0	0

Tabelle 30: t-Test – Mutationsrate: 0,25% vs. 0,1% (Typ b)

Bei den Gütewerten des Typs b) konnte praktisch keine Verbesserung festgestellt werden. Im Allgemeinen scheint sich die Mutationsrate nicht signifikant auf diese Werte auszuwirken.

Diese so unterschiedlichen Ergebnisse für die beiden Typen von Gütewerten legen den Schluss nahe, dass für die Mutation im evolutionären Zyklus und für diejenige, die die Nachbarschaft festlegt, unterschiedliche Raten verwendet werden sollten.

4.4 Testläufe mit konstanter Mutationsrate

Für diese Testläufe wurden dieselben Einstellungen genommen wie im vorherigen Abschnitt, nur dass die Mutationsrate die folgenden konstanten Werte annahm:

1. 16 Bits,
2. 12 Bits,
3. 8 Bits und
4. 4 Bits.

4.4.1 Läufe ohne Baldwin-Lernen

Die Ergebnisse sind im Anhang als Tabellen 83, 84, 85 und 86 bezeichnet.

Es wurden wieder t-Tests zwischen den Läufen mit benachbarter Mutationsrate durchgeführt. Dabei ergaben sich folgende Ergebnisse:

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	0	+1	0	+1

Tabelle 31: t-Test – Mutationsrate: 16 Bit vs. 12 Bit

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	0	0	-1	-1

Tabelle 32: t-Test – Mutationsrate: 12 Bit vs. 8 Bit

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	0	0	0	0

Tabelle 33: t-Test – Mutationsrate: 8 Bit vs. 4 Bit

Die Unterschiede sind größtenteils nicht signifikant und weisen auf keine allgemeine Verbesserung in irgendeiner Richtung hin. Man kann aber aufgrund der Verschlechterung von Problem *scpnrg1* bei Herabsetzung der Mutationsrate von 12 auf 8 Bits vermuten, dass die verwendeten Raten diesmal zu klein für dieses Problem waren. Zum Vergleich: bei den Tests mit der kleinsten linearen Mutationsrate wurden bei diesem Problem 10 Bits / Genom mutiert. Etwa in dieser Größenordnung sollte vermutlich die optimale Mutationsrate für dieses Problem liegen.

Abbildung 9 fasst die Ergebnisse dieser Läufe zusammen.

4.4.2 Läufe mit Baldwin-Lernen

Die Ergebnisse sind im Anhang als Tabellen 87, 88, 89 und 90 bezeichnet.

Es wurden wieder t-Tests zwischen den Läufen mit benachbarter Mutationsrate durchgeführt. Dabei ergaben sich bei Vergleichen des Typs a) folgende Ergebnisse:

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	+1	0	0	+1	0	0	+1	0	+1	+1	+5

Tabelle 34: t-Test – Mutationsrate: 16 Bit vs. 12 Bit (Typ a)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	+1	0	0	+1	0	0	+1	+1	+1	+1	+6

Tabelle 35: t-Test – Mutationsrate: 12 Bit vs. 8 Bit (Typ a)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	+1	0	+1	0	+1	+1	+1	0	0	0	+5

Tabelle 36: t-Test – Mutationsrate: 8 Bit vs. 4 Bit (Typ a)

Die Ergebnisgüte des Typs a) verbesserte sich mit fallender Mutationsrate bei über der Hälfte der Probleme signifikant. Es ist außerdem zu beachten, dass sich die t-Test-Tabellen recht

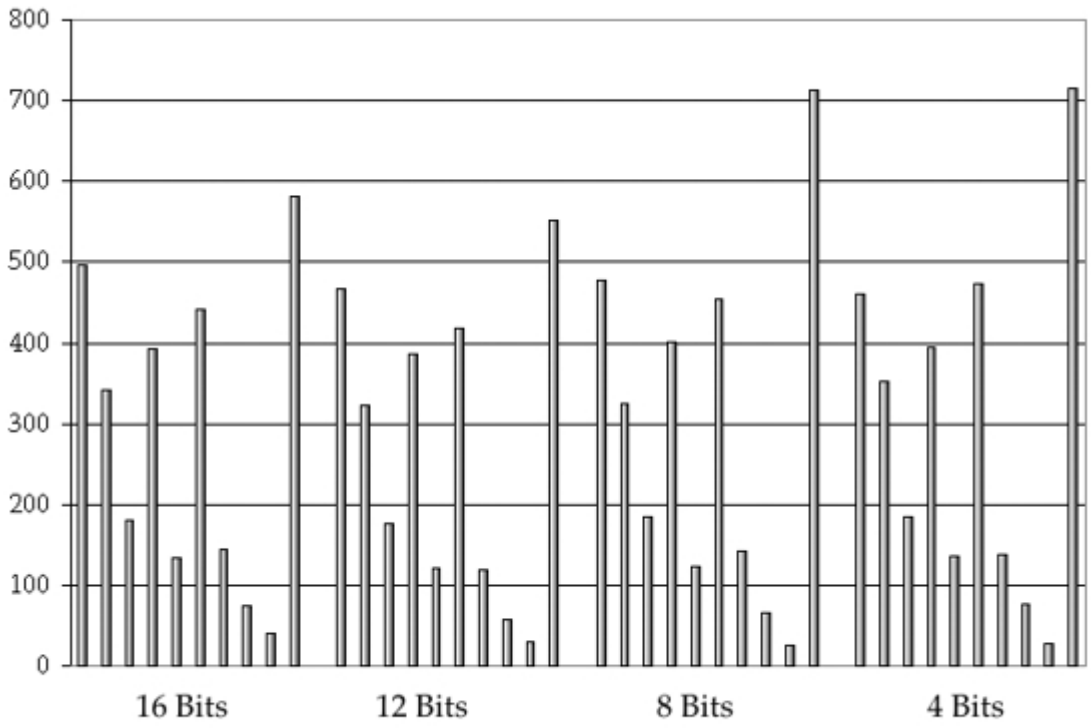

Abbildung 9: *Entwicklung der durchschnittlichen Ergebnis-Güte mit fallender konstanter Mutationsrate von 16, 12, 8 und 4 Bits bei Läufen ohne Baldwin-Lernen. Jeweils 10 gruppierte Säulen stellen die Durchschnitts-Ergebnisse der 10 Multicost-Probleme dar – bei jew. 10 Einzelläufen.*

stark ähneln (vor allem die ersten beiden). Bei manchen Problemen scheint eine Verkleinerung der Mutationsrate eine Verbesserung zu bewirken, bei anderen nicht. Möglicherweise hängt diese Tatsache wieder mit der Dichte der Probleme zusammen.

Es folgen die t-Test-Ergebnisse des Typs b):

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	0	0	+1	+1

Tabelle 37: t-Test – Mutationsrate: 16 Bit vs. 12 Bit (Typ b)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	+1	0	0	+1

Tabelle 38: t-Test – Mutationsrate: 12 Bit vs. 8 Bit (Typ b)

Vergleich (Prob.)	1	2	3	4	5	6	7	8	9	10	alle
Sign. Untersch.	0	0	0	0	0	0	0	0	0	0	0

Tabelle 39: t-Test – Mutationsrate: 8 Bit vs. 4 Bit (Typ b)

Bei der Ergebnisgüte des Typs b) gab es kaum signifikante Verbesserungen. Weiterhin kann immer noch ein großer absoluter Unterschied zwischen Ergebnissen des Typs a) und denen des Typs b) festgestellt werden. (Es wurden auch t-Tests zwischen den beiden Ergebnis-Typen mit derselben Mutationsrate durchgeführt, die fast alle signifikant waren.)

Diese Ergebnisse deuten darauf hin, dass die Mutationsrate für Baldwin-Lernen und die für den evolutionären Zyklus getrennt untersucht werden müssen. Beide sollten zu diesem Zweck mit unterschiedlichen Werten besetzt oder sogar mit unterschiedlichen Mutationsoperatoren durchgeführt werden. Die Mutationsrate für das Baldwin-Lernen sollte dabei vermutlich „kleiner“ sein.

Abbildung 10 fasst die Ergebnisse der Läufe mit Baldwin-Lernen zusammen.

4.5 Fazit und Ausblick

Bei den Testläufen wurden zwei Vermutungen aufgestellt, deren weitere Erforschung wichtig erscheint.

Die erste besagt, dass die Mutationsrate von der Dichte der Probleme abhängig gemacht werden sollte. Sowohl eine proportionale Steigerung der Mutationsrate mit der Genomlänge als auch eine konstante Mutationsrate neigten dazu, einige Probleme recht gut zu lösen, aber bei anderen mangelhafte Ergebnisse zu liefern. Es wurde tendentiell beobachtet, dass mit sinkender Problemdichte d die Mutationsrate sinken sollte, während sie mit steigender Problemgröße (Genomlänge $|T|$) zunehmen sollte. Als nächste Untersuchung wird vorgeschlagen, die Mutationsrate r (die Anzahl der zu flippenden Bits pro Genom) in folgender Weise zu berechnen:

$$r : \mathbb{R} \times \mathbb{N} \to \mathbb{N} : r(d, |T|) = \lceil k \cdot d \cdot |T| \rceil$$

für eine zu ermittelnde Konstante $k \in \mathbb{R}$, die Problemdichte d und die Problemgröße $|T|$.

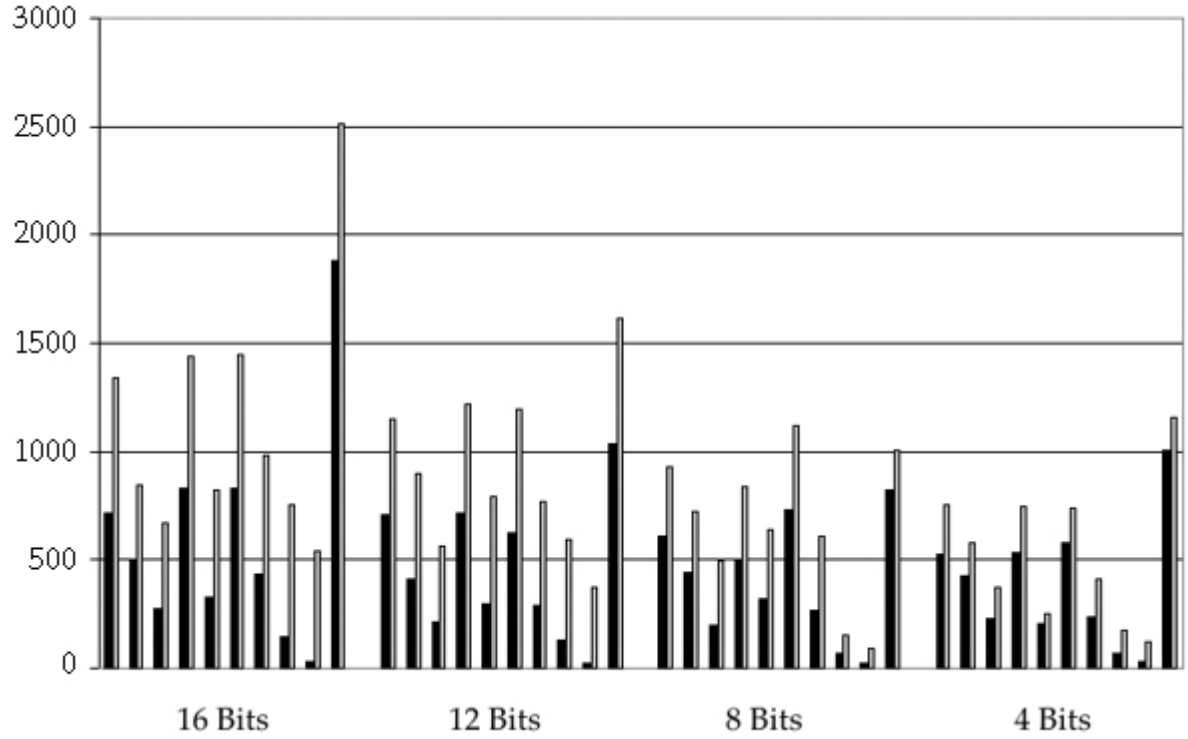

Abbildung 10: *Entwicklung der durchschnittlichen Ergebnis-Güte mit fallender konstanter Mutationsrate von 16, 12, 8 und 4 Bits mit Baldwin-Lernen. Jeweils 10 gruppierte Doppel-Säulen stellen die Durchschnitts-Ergebnisse der 10 Multicost-Probleme dar – bei jew. 10 Einzelläufen. Die hellen Säulen stehen für Ergebnisse des Typs a), die dunklen für die des Typs b).*

Die zweite Vermutung ist, dass die Mutation innerhalb des Zyklus von der für das Baldwin-Lernen getrennt werden sollte. Bei den Versuchen zeigte sich, dass sogar bei relativ niedriger Mutationsrate die bezüglich dieser Mutation berechneten Nachbarn sich signifikant in ihrer Bewertung unterscheiden konnten. Dennoch konnte bei Versuchen ohne Baldwin-Lernen umgekehrt gezeigt werden, dass eine noch niedrigere Mutationsrate im evolutionären Zyklus wieder zu Verschlechterungen führen würde. Aus diesen Tatsachen wird der Schluss gezogen, dass die Mutation, die das Baldwin-Lernen definiert, kleinere Veränderungen am Genom verursachen sollte als die tatsächliche Mutation der Individuen während des Zyklus.

Die Gesamtbilanz der Testläufe ist bei den Unicost-Problemen gut, es konnten sogar einige der Referenzwerte unterschritten werden. Bei den Multicost-Problemen war die Güte stark von der benutzten Startpopulationserzeugung abhängig. Die Beasley-Variante brachte gute, die randomisierte Variante schlechtere Ergebnisse. Diese Unterschiede können zum einen auf die noch nicht zufriedenstellend eingestellte Mutationsrate zurückgeführt werden, zum anderen möglicherweise auf eine „zu frühe“ Terminierung. Es ist einleuchtend, dass die Beasley-Variante gewöhnlich zunächst bessere Individuen in der Population hat und höchstens später von der randomisierten Variante überholt werden kann. Zu diesem Zweck sind noch weitere Langzeittests mit der randomisierten Variante notwendig.

5 Visualisierung Baldwinschen Lernens

In diesem Abschnitt wird die Implementierung und Visualisierung des Nachbarschaftsgraphen beschrieben.

5.1 Implementierung

Die Visualisierung ist als Java-Applet entworfen. Alle erforderlichen Komponenten befinden sich im Paket „graphVis“. Die Visualisierung baut jedoch auf dem evolutionären Algorithmus auf und benutzt über Schnittstellen einige seiner Komponenten.

Das Paket „graphVis“ enthält folgende Klassen:

- *DargestellterGraph*: Diese Klasse dient zum Speichern der konkreten Darstellung eines Graphen und zur Ausgabe einiger Informationen über diese Darstellung. Dazu enthält die Klasse Attribute für die Koordinaten von Knoten und Pfeilanfangs- und Endpunkten sowie Beschriftungen und deren Koordinaten. Über entsprechende Methoden können diese Attribute gelesen und geschrieben werden. Außerdem gibt es Methoden zur Rückgabe aller Knoten, die sich in einem bestimmten Rechteck befinden bzw. zur Rückgabe aller Knoten in diesem Rechteck, die zusätzlich minimalen Abstand zum Mittelpunkt haben. Diese Methode wird für die Zuordnung von Knoten zur Mausposition auf dem Bildschirm benötigt.

  Die Graphdarstellung wird im Paket *darstellungsModi* erzeugt und im Paket *zeichenModi* in eine konkrete Darstellung aus Linienzügen, Kreisen, etc. übertragen, die direkt gezeichnet werden können (s. u.).

- *Grapherzeugung*: Diese Klasse besitzt Methoden zur Erzeugung von Nachbarschaftsgraphen. Sie hat eine Schnittstelle zum Mutations-Interface des evolutionären Algorithmus, von dem eine Liste aller Mutationsnachbarn eines Individuums abgefragt werden

kann. Die Klasse bietet von einem Individuum ausgehend Möglichkeiten zur Erzeugung seines Nachbarschaftsgraphen bis zu einer bestimmten Tiefe oder zur Erzeugung des gesamten Nachbarschaftsgraphen. Letztere Methode kann nur bei Individuen mit kleinen Genomen sinnvoll eingesetzt werden, weil sonst die Nachbarschaft sehr groß wird. Außerdem gibt es eine Methode zur Erzeugung der Nachbarschaft einer gesamten Population. Bei der bisherigen Implementierung wird nur die Methoder zur Erzeugung der Nachbarschaft bis zu einer bestimmten Tiefe verwendet.

- *Konstanten*: Enthält die Konstanten des Pakets.
- *OpUndParErzeugungBaldwin*: Diese Klasse erzeugt einen Operatoren- und einen Parametersatz für die Berechnung der Nachbarschaft mit Baldwin-Lernen. Der Operatorensatz ist ein Objekt des Typs OperatorenAuswahl, der Parametersatz ein Objekt des Typs ParameterAuswahl, die beide im Paket evolutionaererAlgorithmus definiert sind.
- *OpUndParErzeugungNormal*: Diese Klasse erzeugt analog einen Operatoren- und einen Parametersatz für die Berechnung der Nachbarschaft ohne Baldwin-Lernen.
- *Vis*: Diese Klasse ist die Startklasse des Applets. Sie steuert die Erzeugung der Nachbarschaftsgraphen, den Bildschirmaufbau und den Umgang mit Ereignismeldungen wie Mausbewegungen und Mausklicks.

Das Paket „graphVis“ enthält die Pakete „graph“, „darstellungsModi“ und „zeichenModi“, die nachfolgend beschrieben werden.

5.1.1 Das Paket „graph“

Das Paket „graph“ implementiert eine gerichtete Graphstruktur, die an die Bedürfnisse für eine Visualisierung von Nachbarschaftsgraphen angepasst ist. Es enthält die Klassen *Knoten* und *Graph*.

Die Klasse *Knoten* dient im Wesentlichen der Speicherung eines Knotens und hat folgende Attribute:

- name: ein Name, der ein beliebiges Objekt sein kann. In der Graphstruktur werden Knoten mit gleichem Namen (das sind Knoten, deren Methode equals den Wert *true* liefert) nicht doppelt eingefügt und als ein einziger Knoten behandelt. Bei der Visualisierung wurde das Attribut name mit einem kompletten Individuum belegt.
- vorgaenger, nachfolger: Listen von Vorgängern und Nachfolgern (jeweils als Hashtabelle implementiert). Diese beiden Attribute werden von der Klasse Graph gesteuert und über entsprechende Methoden gesetzt und gelesen.
- zusatzinformation: ein beliebiges Objekt. Knoten mit gleichem Namen, aber unterschiedlicher Zusatzinformation, werden dennoch als ein und derselbe Knoten behandelt. Die Zusatzinformation kann beispielsweise eine Beschriftung oder eine Gewichtung sein. Bei der Implementierung als Nachbarschaftsgraph bietet sich an, ein Flag als Zusatzinformation zu speichern, das anzeigt, welcher Knoten der Ausgangsknoten der Nachbarschaft ist.

Außerdem hat die Klasse Methoden zum Hinzufügen und Entfernen von Nachfolgern und Vorgängern, Setzen und Lesen von Name und Zusatzinformation, sowie für eine Ausgabe der eigenen Attribute im Textformat.

Die Klasse *Graph* baut auf der Klasse *Knoten* auf und erzeugt und verwaltet die Graphstruktur. Die Implementierung basiert auf einer Adjazenzliste, in der alle Knoten mitsamt ihrer Nachfolger gespeichert sind. Die Klasse enthält Methoden zum Einfügen oder Entfernen von Knoten und Kanten und zum Einfügen einer Clique oder eines Sterngraphen sowie eines bereits vorhandenen Graphen in einen Graphen. Ein Sterngraph ist dabei ein Mittelknoten und eine Menge von weiteren Knoten, die in den Graphen so eingefügt werden, dass vom Mittelknoten aus zu jedem der übrigen Knoten eine Kante verläuft. Diese Struktur dient bei der Visualisierung der Beschreibung der Beziehung eines Individuums zu seinen Nachbarn.

5.1.2 Das Paket „darstellungsModi“

Dieses Paket dient der Erzeugung einer abstrakten Darstellung eines Graphen. Zu allen Knoten werden Koordinaten berechnet, an denen diese platziert werden sollen und ebenso werden die Koordinaten der zu zeichnenden Kanten angegeben. Wie die tatsächliche Darstellung der Knoten und Kanten auf dem Bildschirm auszusehen hat, wird hier offengelassen und an das Paket „zeichenModi“ übertragen.

Im Paket „darstellungsModi“ befindet sich das Interface *DarstellungsKit*, welches einige Methoden definiert, die ein Darstellungsmodus haben muss. Die wichtigste Methode heißt erzeuge. Sie erhält als Eingabe ein Objekt des Typs *Graph* mit einem ausgezeichneten Ausgangsknoten und gibt ein Objekt des Typs *DargestellterGraph* zurück. Außerdem gibt es eine Methode, die den Hintergrund für den jeweiligen Darstellungsmodus erzeugt und eine Methode, die die Anzahl der vom Ausgangsknoten aus darzustellenden Ebenen zurückgibt.

Weiterhin enthält das Paket die Klassen *DarstellungsModus1* (bewertungsorientierte Darstellung), *DarstellungsModus2* (radiale Darstellung) und *DarstellungsModus3* (richtungsorientierte Darstellung), die jeweils eine Implementierung des Interfaces sind. Wie die konkrete Platzierung der Knoten in den jeweiligen Darstellungsmodi erfolgt, wird in Abschnitt 5.3 beschrieben.

Die Klasse *Konstanten* definiert die Konstanten des Pakets.

5.1.3 Das Paket „zeichenModi“

Das Paket *zeichenModi* ist dafür zuständig, ein Objekt des Typs *DargestellterGraph* auf dem Bildschirm auszugeben.

Um die Erzeugung einer konkreten Darstellung eines Graphen von ihrer Ausgabe auf dem Bildschirm trennen zu können, wurde ein Datentyp (also eine Klasse) benötigt, der alle Informationen über die zu zeichnenden Elemente speichern kann. Ein Objekt dieses Typs kann in einer Erzeuger-Methode erstellt und an eine Ausgabe-Methode übergeben werden. Zu diesem Zweck wurde die Klasse *List* von Java benutzt und mit einigen Regeln versehen. Die Objekte, die eine solche Liste zu zeichnender Elemente enthalten darf, müssen einen der folgenden Typen haben:

- *AusgabeMerkmale* (in diesem Paket definiert),

- *Polygon* (interne Klasse von Java),
- *KreisKlasse* (in diesem Paket definiert, speichert Position und Radius eines Kreises),
- *Integer* oder
- *String*.

Beim Zeichnen wird die Liste von vorne nach hinten durchlaufen und je nach gefundenem Objekttyp wird verschiedenes Verhalten ausgelöst:

- Objekte des Typs AusgabeMerkmale erzeugen selbst keine Ausgabe, sondern haben nur eine Wirkung auf alle nachfolgenden Objekte. Sie definieren, ob und in welcher Farbe Füllung bzw. Rahmen von Polygonen und Kreisen gezeichnet werden. Für Strings definieren sie nur die Schriftfarbe. Die Wirkung bleibt erhalten, bis ein weiteres Objekt dieses Typs gefunden wird.
- Objekte der Typen Polygon bzw. KreisKlasse werden als Linienzug bzw. Kreis direkt mit den aktuellen Farb-Parametern für Rahmen und Füllung auf dem Bildschirm gezeichnet.
- Objekte der Typen Integer und String dienen zusammen der Positionierung und Ausgabe eines Textes auf dem Bildschirm. Sie dürfen nur in einer bestimmten Reihenfolge in der Liste vorkommen. Diese Reihenfolge ist (Integer, Integer, String). Das erste Integer-Objekt wird dabei als X-Koordinate, das zweite als Y-Koordinate der linken, unteren Ecke des zu zeichnenden Strings interpretiert. Ausgegeben wird nur der String selber an der bezeichneten Position.

Das Erzeugen und Ausgaben der Bildschirmdarstellung erfolgt prinzipiell durch Klassen, die das Interface *ZeichenKit* implementieren. Es wurde bisher nur eine einzige Implementierung des Interfaces, mit Namen *ZeichenModus1*, geschrieben. Zeichenmodi enthalten Methoden zum Zeichnen von Pfeilen, Knoten und, darauf aufbauend, von ganzen Graphen, also Objekten des Typs *DargestellterGraph*. Diese Methoden erzeugen zunächst noch keine Bildschirmausgabe, sondern geben nur eine Liste mit den oben beschriebenen Eigenschaften zurück. Zusätzlich enthält ein Zeichenmodus deshalb eine Methode zur Ausgabe dieser Listen-Objekte.

Auch dieses Paket enthält eine Klasse *Konstanten* für die Speicherung konstanter Attribute.

5.2 Konkrete Arbeitsweise und Bedienung

Die Klasse *Vis* des Pakets „graphVis“ setzt die oben beschriebenen Elemente zusammen, erzeugt entsprechende Darstellungen in Fenstern und steuert die Interaktion mit dem Benutzer. Die Klasse hat zurzeit noch eine eher temporäre und recht statische Form. Einige Parameter, die im Augenblick noch im Programm festgelegt werden, sollten zukünftig innerhalb der graphischen Benutzerschnittstelle einstellbar sein; dazu gehört beispielsweise die Anzahl der darzustellenden Ebenen des Nachbarschaftsgraphen oder andere Eigenschaften der verschiedenen Darstellungen, die in der Klasse *Konstanten* des Pakets *darstellungsModi* festgelegt werden. Außerdem sollte die Auswahl des Ausgangsindividuums flexibler erfolgen

können, beispielsweise durch Angabe eines Genoms durch den Benutzer. Auch die Angabe einer SCP-Instanz (M, T, κ) durch den Benutzer ist noch nicht möglich.

In ihrer jetzigen Form arbeitet die Klasse folgendermaßen:

Zu Beginn wird eine zufällige Instanz (M, T, κ) des SCP erzeugt; im Programm könnnen dafür $|M|$, $|T|$, die minimale und maximale Anzahl der Indizes pro Index-Teilmenge und die minimalen und maximalen Kosten κ angegeben werden. Auf Basis dieses Problems und der aktuellen Einstellungen für Operatoren und Parameter wird eine Population erzeugt. Dasjenige Individuum, dessen Bewertung dem Median der Population entspricht wird zum Ausgangsknoten erklärt. (Die Idee ist, dass von diesem Individuum aus wahrscheinlich sowohl Verbesserungen als auch Verschlechterungen in der näheren Nachbarschaft zu sehen sind.) Vom Ausgangsindividuum aus wird dann ein Nachbarschaftsgraph erzeugt (zunächst ohne Baldwin-Lernen), dessen Tiefe der Ebenenanzahl des aktuellen Darstellungsmodus entspricht. In diesem Darstellungsmodus wird der Graph gezeichnet.

Wenn der Graph gezeichnet wurde, kann zum ersten Mal der Benutzer eingreifen. Wenn die Maus über einen Knoten bewegt wird, wird dessen Fitness und Genom angezeigt. Beim Klick auf einen Knoten wird dieser Knoten zum Ausgangsknoten und der Graph neu gezeichnet. Beim Druck auf eine beliebige Taste der Tastatur wird, wieder vom aktuellen Ausgangsknoten ausgehend, der Graph neu berechnet, wobei diesmal die Berechnung der Fitness mit eingeschaltetem Baldwin-Lernen erfolgt; die aktuellen Einstellungen lassen dabei eine Berücksichtigung der gesamten Nachbarschaft eines Individuums zu. Der Baldwin-Modus bleibt solange aktiviert, bis wieder eine beliebige Taste gedrückt wird, was ein Zurückschalten in den Modus ohne Baldwin-Lernen bewirkt. Ob Baldwin-Lernen aktiviert ist, wird oben am Bildschirm angezeigt.

Jederzeit kann der Benutzer durch eine Auswahlbox zwischen den verschiedenen implementierten Darstellungsmodi wählen.

Die Einstellungen der Operatoren und Parameter, die der aktuellen Visualisierung zugrundeliegen, werden unten am Bildschirm aufgelistet.

Bild 11 zeigt eine Beispielansicht mit eingeschaltetem Baldwin-Lernen.

5.3 Visualisierung des Nachbarschaftsgraphen

In diesem Abschnitt werden die drei Darstellungsmodi beschrieben, die bisher implementiert wurden. Sie haben folgende Gemeinsamkeiten:

- Die Bewertung der Individuen wird durch die Farbe des entsprechenden Knotens veranschaulicht. Der beste von allen aktuell am Bildschirm angezeigten Knoten wird mit grüner Füllung gezeichnet, der schlechteste mit roter. Alle dazwischenliegenden Knoten werden in Schattierungen zwischen grün und rot gezeichnet.

- Knoten, die für dasselbe Individuum stehen, können mehrfach vorkommen. Der Grund, warum (bisher) keine Verfahren ohne Mehrfachdarstellung von Knoten implementiert wurden, liegt u. a. darin, dass es durch die dabei unvermeidbaren Kreuzungen von Kanten erheblich schwieriger wird, übersichtliche Darstellungen zu finden. Außerdem ist bei der gegebenen Problemstellung eine Darstellung mit Doppelten durchaus zufriedenstellend: Es soll vor allem eine Vorstellung von der Dichte und Güte des Suchraums vermittelt werden, sowie von der Erreichbarkeit verschiedener Bereiche untereinander durch

Abbildung 11: *Beispielansicht der Visualisierung eines Nachbarschaftsgraphen. Manche Operatoren werden für die aktuelle Form der Visualisierung nicht verwendet, könnten aber für künftige Erweiterungen benutzt werden.*

die Nachbarschaftsbeziehung. Diese Ziele werden durch das Vorkommen doppelter Individuen kaum beeinträchtigt. In gewissem Sinne kann man sogar argumentieren, dass die Darstellung doppelter Individuen näher an der Realität des evolutionären Algorithmus liegt als es eine Darstellung ohne Doppelte täte: Diese Darstellung entspricht nämlich genau der Mutationsnachbarschaft, die auch der evolutionäre Algorithmus „sieht“; bei der Mutation werden ebenfalls keine doppelten Individuen herausgefiltert. Dennoch muss bei der Interpretation berücksichtigt werden, dass Doppelte auch gehäuft vorkommen können. Beispielsweise ist die Nachbarschaft normalerweise reflexiv, was zur Folge hat, dass jedes Individuum als sein eigener Nachbar ein zweites Mal vorkommen kann.

- Die Interaktion des Benutzers mit der graphischen Oberfläche ist von der Darstellung unabhängig.

Die Darstellungsmodi wurden mit unterschiedlichen ästhetischen Absichten entworfen, die hauptsächlich den Zielen einer intuitiven Darstellung der Suchraumstruktur und einer übersichtlichen Darstellung der Individuen-Bewertung unterworfen sind.

5.3.1 Bewertungsorientierte Darstellung

Die bewertungsorientierte Darstellung ist in der Klasse *DarstellungsModus1* des Pakets „darstellungsModi“ implementiert. In diesem Modus wird das Augenmerk auf eine klare und leicht ersichtliche Darstellung der Bewertung der angezeigten Individuen gerichtet. Zu diesem Zweck werden die Knoten je nach ihrer Bewertung in vertikaler Richtung höher oder tiefer auf dem Bildschirm platziert. Eine beschriftete Skalenachse in der Mitte zeigt die Bewertung der jeweiligen Höhe an. In horizontaler Richtung gibt es (zunächst) drei feste Positionen, an denen die Knoten platziert werden: Das Ausgangsindividuum liegt genau auf der Skalenachse; Nachbarindividuen, die eine bessere Bewertung als das Ausgangsindividuum haben, werden in festem Abstand rechts von der Skalenachse gesetzt, solche, die eine schlechtere oder dieselbe Bewertung haben, werden in festem Abstand links von der Skalenachse gesetzt. Wenn es mehrere Nachbarindividuen gibt, die dieselbe Bewertung haben, werden diese jeweils nach außen, von der Skalenachse weg, nebeneinander gesetzt. Diese Darstellung ist in dieser Form nur für die Darstellung der unmittelbaren Nachbarschaft des Ausgangsindividuums geeignet, die Nachbarn der Nachbarn können also nicht angezeigt werden.

Abbildung 12 zeigt eine durch die bewertungsorientierte Darstellung erzeugte Beispielansicht.

5.3.2 Radiale Darstellung

Die radiale Darstellung ist in der Klasse *DarstellungsModus2* des Pakets „darstellungsModi“ implementiert. Bei dieser Darstellung soll die Struktur des Suchraums mit möglichst vielen angezeigten Nachbarschaftsebenen intuitiv verdeutlicht werden. Insbesondere sollen dichtere und dünnere Bereiche erkennbar werden und die Verbundenheit von „guten“ mit „schlechten“ Bereichen angezeigt werden (letzteres mithilfe der farblichen Kennzeichnung der Bewertung).

In diesem Modus wird der Knoten des Ausgangsindividuums in der Mitte einer Reihe von konzentrischen Kreisen platziert. Die Nachbarschaft ersten Grades wird auf den ersten (von innen) der Kreise platziert, die Nachbarschaft zweiten Grades auf dem zweiten usw. Verall-

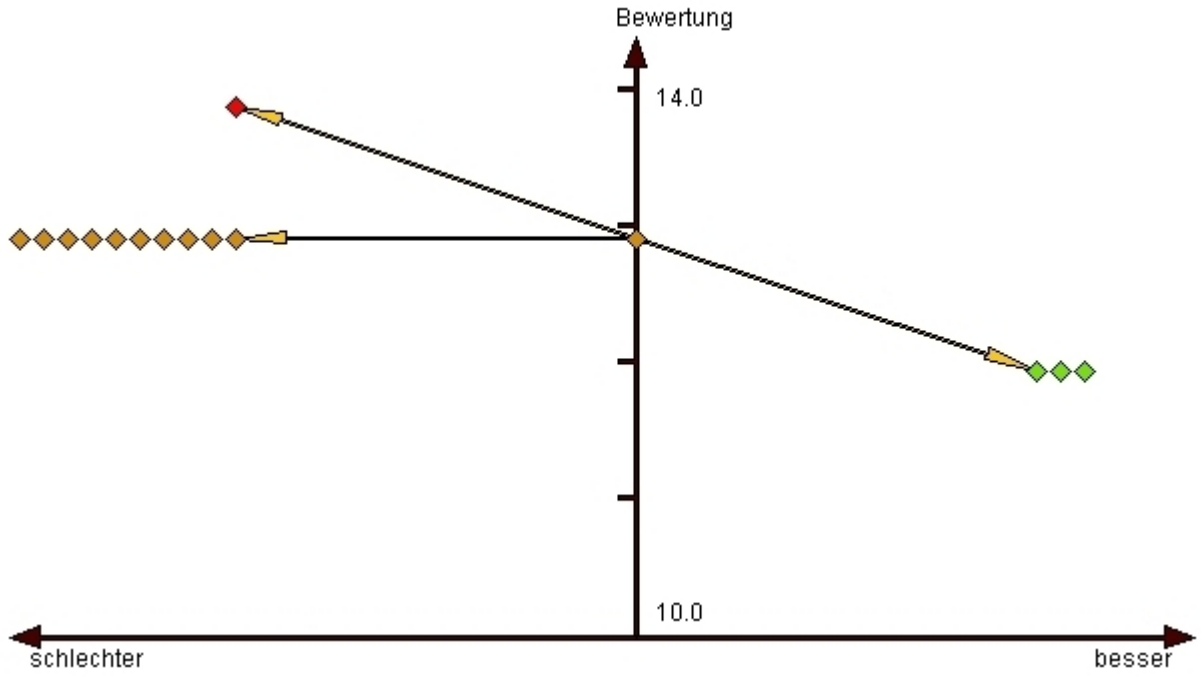

Abbildung 12: *Beispielansicht der Nachbarschaft eines Individuums (Mitte) mit bewertungsorientierter Darstellung.*

gemeinernd soll die Position des Ausgangsknotens als nullter Kreis mit Radius 0 bezeichnet werden.

Die Verteilung der Knoten auf den Kreisbahnen wird wie folgt berechnet (siehe auch Abbildung 13):

Der Ausgangsknoten K wird auf dem 0. Kreis platziert.

Dann wird folgende Vorgehensweise iteriert, bis alle Knoten des Graphen platziert sind:

- Weise jedem Knoten K', der bereits platziert ist, einen Freiheitswinkel α' vom Mittelpunkt aus zu. (Ein Freiheitswinkel bestehe eigentlich aus zwei Winkeln, die von einem Null-Winkel aus gemessen den Beginn und das Ende des Freiheitswinkels bezeichnen.)
- Verteile für jeden solchen Knoten K', der sich auf einer Kreisbahn k befindet, dessen n' Nachbarn gleichmäßig innerhalb des Freiheitswinkels α' auf der k+1-ten Kreisbahn. Die Knoten sind dann jeweils um den Winkel $\frac{\alpha'}{n'}$ voneinander entfernt. Von den Rändern des Freiheitswinkels wird jeweils ein Abstand von $\frac{\alpha'}{2n'}$ eingehalten.

Die Zuweisung der Freiheitswinkel wird wie folgt vorgenommen:

Für jeden Knoten K' werden dessen „linke und rechte Kreisbahn-Nachbarn“ K'_l und K'_r ermittelt; gemeint sind hier nicht Nachbarn bezüglich der Nachbarschaftsrelation, sondern Knoten, die gegen bzw. im Uhrzeigersinn am nächsten am betreffenden Knoten platziert sind. Die Winkelhalbierende des Winkels, den K' und K'_l bzw. K' und K'_r, mit dem Mittelpunkt als Scheitel, definieren, wird als linke bzw. rechte Grenze eines vorläufigen Freiheitswinkels

genommen. Ist K' der einzige Knoten auf seiner Kreisbahn wird ihm ein vorläufiger Freiheitswinkel von 0° bis 360° zugewiesen (insbesondere ist das beim Ausgangsknoten der Fall).

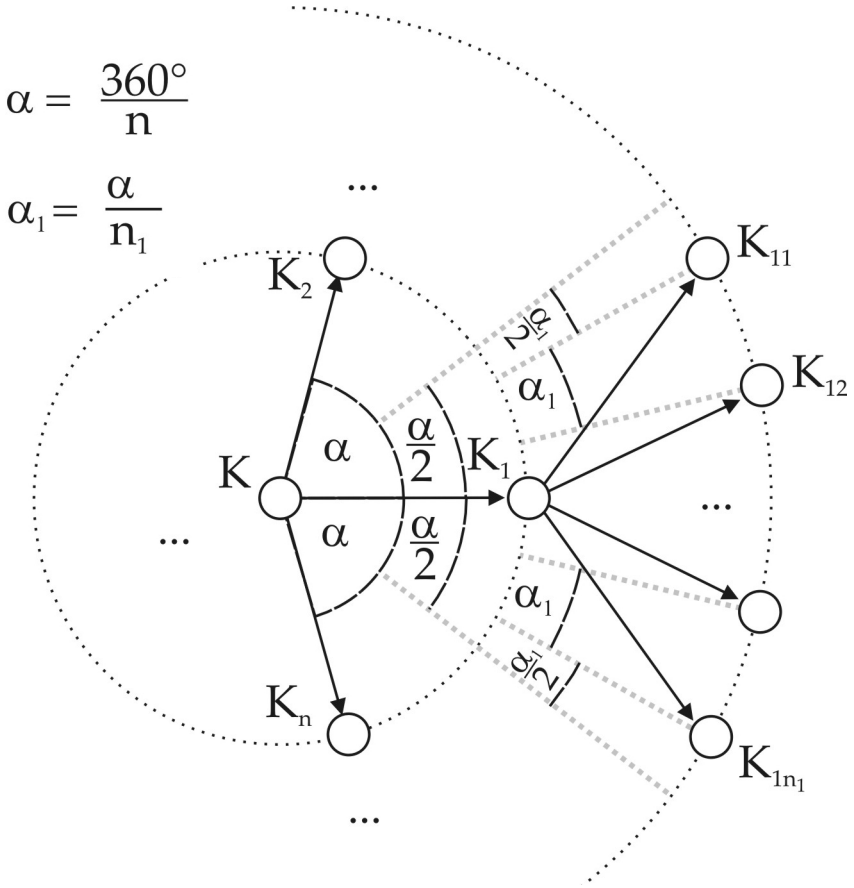

Abbildung 13: *Positionierung der Knoten auf den Kreisbahnen bei der radialen Darstellung. Die n Nachbarknoten von Knoten K werden in gleichen Abständen auf der ersten Kreisbahn verteilt. Jeder Knoten auf einer k-ten Kreisbahn erhält von der Mitte aus einen Freiheitswinkel, innerhalb dessen seine Nachbarknoten auf der $k+1$-ten Kreisbahn verteilt werden.*

Nun muss noch verhindert werden, dass es Knoten gibt, die Freiheitswinkel haben, die es erlauben würden, Nachbarknoten so zu platzieren, dass die zugehörigen Kanten eine der Kreisbahnen schneiden würden: Befinde sich ein Knoten auf der k-ten Kreisbahn. Wenn von ihm aus überhaupt eine Kreisbahn geschnitten wird, dann wird auch die k-te Kreisbahn geschnitten. Nun wird für jeden Knoten ein maximaler Freiheitswinkel so festgesetzt, dass die äußersten beiden Punkte, wo ein Nachbarn des Knotens platziert werden dürfte, jeweils mit dem Knoten selbst eine Tangente zum k-ten Kreis bilden.

Der tatsächliche Freiheitswinkel ist dann der Schnittbereich zwischen dem vorläufigen und dem maximalen Freiheitswinkel. Beim Ausgangsknoten verläuft auch der maximale Freiheitswinkel von 0° bis 360°.

Eine Beschneidung des vorläufigen Freiheitswinkels findet in der Praxis allerdings nur selten bei Graphen mit sehr wenigen Knoten statt.

Jede Gruppe von Nachbarknoten wird auf den Kreisbahnen im Gegenuhrzeigersinn nach fallender Bewertung sortiert.

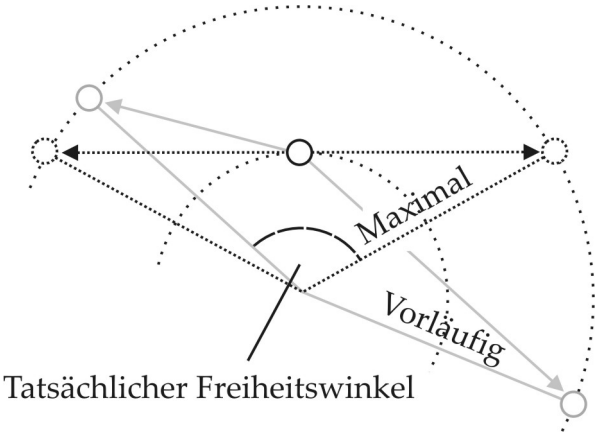

Abbildung 14: *Die Überschneidung des maximalen und des vorläufigen Freiheitswinkels bildet den tatsächlichen Freiheitswinkel.*

Abbildung 15 zeigt die Ansicht einer Beispielnachbarschaft mit radialer Darstellung.

Es wurde auch überlegt, bei der radialen Darstellung den Bewertungsunterschied zwischen zwei Nachbarknoten durch die Entfernung der Knoten voneinander zu kodieren; zu diesem Zweck würde jede Kreisbahn durch einen ringförmigen Bereich ersetzt, auf dem die Knoten je nach Bewertung weiter innen oder weiter außen platziert werden könnten. Diese Idee wurde fallengelassen zugunsten der deutlicher erkennbaren Suchraumstruktur bei der Darstellung mit einheitlichen Entfernungen. Außerdem würde bei der Variante mit dem ringförmigen Bereich ein Problem auftreten, wenn man mehr als eine Nachbarschaftsebene darstellen wollte; die Ringe der zweiten Ebene müssten nicht vom Mittelpunkt, sondern von den Knoten der ersten Ebene einen bestimmten Abstand einhalten. Ebenso verhält es sich mit jeder weiteren Ebene. Dabei ergibt sich das Problem, dass die Freiheitswinkel unter Umständen negativ werden, also keine weiteren Knotenplatzierungen zulassen. Falls eine solche radiale Darstellung noch implementiert werden sollte, müsste vorher diese Problematik gelöst werden.

5.3.3 Richtungsorientierte Darstellung

Die richtungsorientierte Darstellung ist in der Klasse *DarstellungsModus3* des Pakets „darstellungsModi“ implementiert. Sie ist der bewertungsorientierten Darstellung ähnlich, kann aber beliebig viele Nachbarschaftsebenen anzeigen. In diesem Sinne vereinigt sie Eigenschaften der bewertungsorientierten und der radialen Darstellung.

Am linken Fensterrand wird eine vertikale Skalenachse dargestellt, die mit Bewertungen beschriftet ist. Jeder Knoten wird der Bewertung des zugehörigen Individuums entsprechend auf einer bestimmten Höhe platziert. Rechts von der Skalenachse befinden sich in gleichmäßigen festen Abständen die Positionen der verschiedenen Nachbarschaftsgrade. Knoten gleichen Nachbarschaftsgrades mit gleicher Bewertung werden bei dieser Darstellung am gleichen Ort platziert.

Abbildung 16 zeigt die Ansicht einer Beispielnachbarschaft mit richtungsorientierter Darstellung.

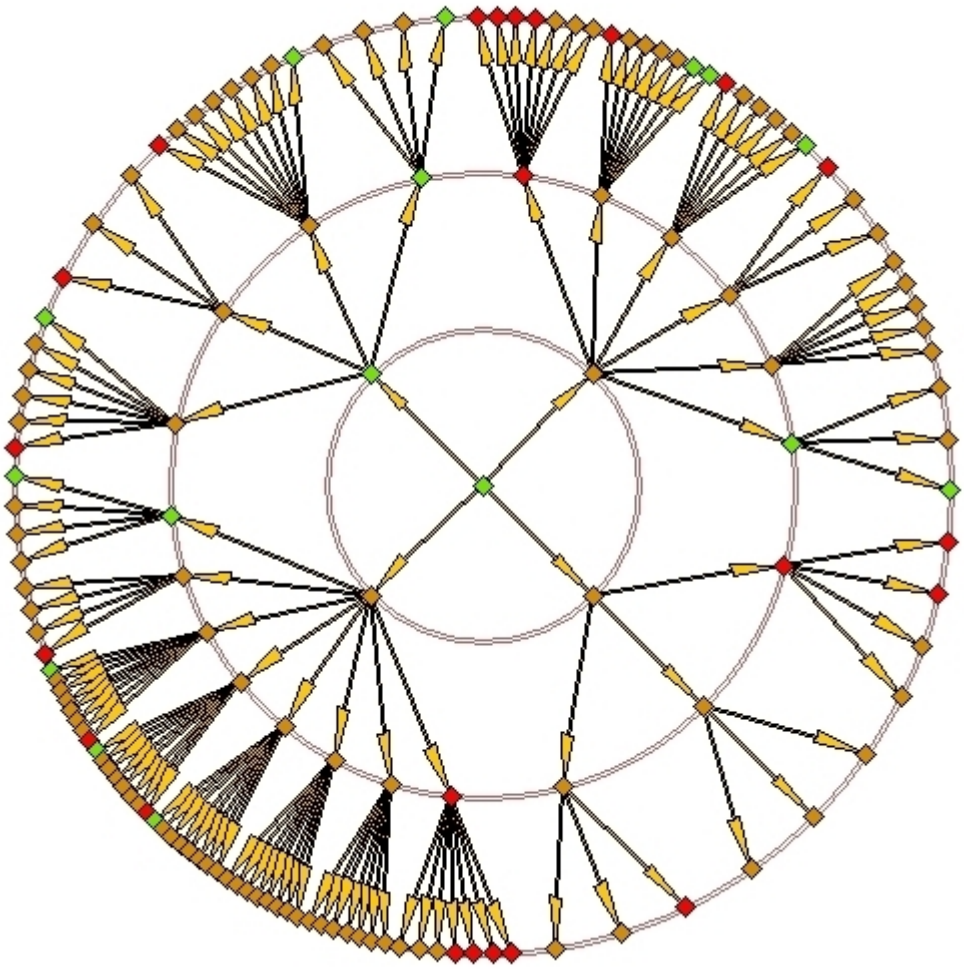

Abbildung 15: *Beispielansicht der Nachbarschaft eines Individuums (Mitte) mit radialer Darstellung bis auf 3 Ebenen.*

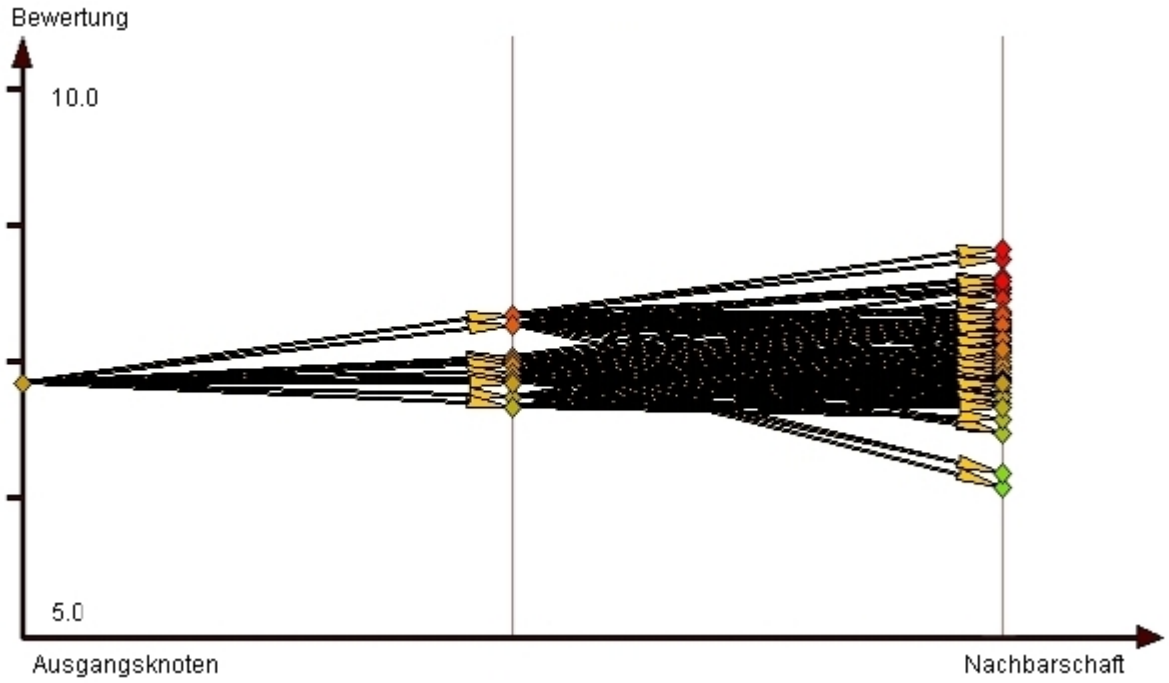

Abbildung 16: *Beispielansicht der Nachbarschaft eines Individuums (links) mit richtungsorientierter Darstellung bis auf 2 Ebenen.*

6 Ausblick

Hier sei zunächst auf die in den Abschnitten 3.4 und 4.5 beschriebenen möglichen Verbesserungen, Weiterentwicklungen und Vorschläge für künftige Forschung am evolutionären Algorithmus verwiesen.

Weiterhin ist es mit dem implementierten System ohne weiteres möglich, ein hybrides Verfahren zu entwerfen, bei dem beispielsweise durch den Iterated-Greedy-Algorithmus von Dietmar Lippold oder andere Ansätze eine vorläufige Lösung vor einem Lauf des evolutionären Algorithmus generiert und bei der Erzeugung der Startpopulation berücksichtigt wird. In Kombination mit dem gesamten SCP-Framework sind für hybride Verfahren sehr viele Möglichkeiten denkbar.

Im Bereich der Visualisierung diskreter Suchräume sowie der Visualisierung der Wirkung von evolutionären Operatoren gibt es eine Vielzahl von offenbar weitgehend unerforschten Richtungen, in die Weiterentwicklung und Forschung gehen können:

- Der Entwurf neuer Darstellungsformen der Mutationsnachbarschaft eines Individuums. Das ist direkt mit dem Entworfenen System möglich, indem neue Klassen programmiert werden, die das Interface *DarstellungsKit* des Pakets „darstellungsModi“ implementieren.
- Die Visualisierung einer ganzen Population mit Nachbarschaftsbeziehungen untereinander und eventuell Nachbarn, die sich nicht in der Population befinden. Dies wäre mit dem entworfenen System prinzipiell möglich, würde jedoch etwas tiefergehende Neuimplementierungen erfordern.

- Eine Visualisierung der Rekombinationsnachbarschaft. Dafür wären neuartige Überlegungen und Neuimplementierungen erforderlich. Es stellt sich die Frage, wie ein solcher Graph aussehen soll, da jeweils zwei Individuen zu einer bestimmten Anzahl von Nachkommen rekombiniert werden. Würde man hier ebenfalls alle möglichen Kinder zu Nachbarn von einem Eltern-Paar erklären, wären die bisher implementierten Rekombinationsformen möglicherweise ungeeignet, da bei ihnen die Anzahl möglicher verschiedener Kinder im schlimmsten Fall nur durch $2^{Genomlaenge}$ nach oben begrenzt ist; das hängt von der „Unterschiedlichkeit“ der Eltern ab.
- Eine Visualisierung der Populationsentwicklung über einen Zeitraum von mehreren Generationen. Dafür bietet das entworfene System sicher eine gute Grundlage, es müsste aber weiterentwickelt werden. Mit einer zeitabhängigen Visualisierung könnte die Wirkung von verschiedenen Arten der Eingliederung von Lernen in die Evolution untersucht werden. Das betrifft insbesondere den Unterschied zwischen dem Baldwin-Effekt und Lamarckscher Evolution. Bei der aktuell implementierten Visualisierung kann nicht entschieden werden, welche Wirkung das Gelernte auf spätere Suchräume haben wird, die sich aus dem aktuellen entwickeln werden. Baldwin-Lernen und Lamarck-Lernen sind innerhalb nur einer einzigen Generation in diesem Sinn dasselbe.

Eine übersichtliche und anschauliche Visualisierung der Suchräume von diskreten Optimierungsproblemen wie dem SCP kann, ebenso wie die Visualisierung der Wirkung von evolutionären Operatoren, sicher dabei helfen, das Verständnis für die Arbeitsweise evolutionärer Algorithmen in der Forschung und vor allem auch in der Lehre zu verbessern.

A Ergebnisse der Testläufe

Alle hier ausgewerteten Testläufe wurden auf dem Cluster *Mozart* des Instituts für Parallele und Verteilte Systeme (IPVS) der Universität Stuttgart durchgeführt. Das Cluster enthält 64 Knoten, von denen jeder zwei Prozessoren des Typs *Intel Xeon 3,066 FSB533 (1MB CACHE)* und 4 Gigabyte Ram hat (weitere Informationen dazu können auf der Homepage des IPVS [IPVS] abgerufen werden). Auf jedem Knoten wurden zwei Testläufe gleichzeitig durchgeführt. Die Läufe wurden vorher angemeldet, so dass davon ausgegangen werden kann, dass keine anderen (großen) Prozesse gleichzeitig auf den jeweiligen Knoten liefen. Zu jedem Lauf wurde die benötigte Prozessorzeit durch das Linux-Kommando *time* abgenommen.

Die Testdaten bestanden aus einem Satz von 8 Unicost-Problemen und 10 Multicost-Problemen. In jedem Test wurden entweder alle Unicost-Probleme oder alle Multicost-Probleme bearbeitet, nicht jedoch beide auf einmal. (Anmerkungen: bei den Unicost-Problemen galt für die Gesamtkosten aller Index-Teilmengen immer: $\kappa(T) = |T|$; alle Kosten bei beiden Problemsätzen waren ganzzahlig.)

Folgende Unicost-Probleme wurden für die Tests verwendet:

1. Name: scpclr10;

   Anzahl der Index-Teilmengen: $|T| = 210$;

   Anzahl der Indizes: $|M| = 511$;

   Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 25,9$;

   Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 63,0$;

   Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 12,3\%$.

2. Name: scpclr11;

   Anzahl der Index-Teilmengen: $|T| = 330$;

   Anzahl der Indizes: $|M| = 1023$;

   Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 41,0$;

   Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 127,0$;

   Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 12,4\%$.

3. Name: scpclr12;

   Anzahl der Index-Teilmengen: $|T| = 495$;

Anzahl der Indizes: $|M| = 2047$;

Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 61,7$;

Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 255,0$;

Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 12,5\%$.

4. Name: scpclr13;

   Anzahl der Index-Teilmengen: $|T| = 715$;

   Anzahl der Indizes: $|M| = 4095$;

   Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 89,2$;

   Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 511,0$;

   Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 12,5\%$.

5. Name: scpcyc06;

   Anzahl der Index-Teilmengen: $|T| = 192$;

   Anzahl der Indizes: $|M| = 240$;

   Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 4,0$;

   Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 5,0$;

   Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 2,1\%$.

6. Name: scpcyc07;

   Anzahl der Index-Teilmengen: $|T| = 448$;

   Anzahl der Indizes: $|M| = 672$;

   Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 4,0$;

   Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 6,0$;

   Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 0,9\%$.

7. Name: scpcyc08;
   Anzahl der Index-Teilmengen: $|T| = 1024$;
   Anzahl der Indizes: $|M| = 1792$;
   Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 4,0$;
   Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 7,0$;
   Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 0,4\%$.

8. Name: scpe1;
   Anzahl der Index-Teilmengen: $|T| = 500$;
   Anzahl der Indizes: $|M| = 50$;
   Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 98,3$;
   Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 9,8$;
   Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 19,7\%$.

Folgende Multicost-Probleme wurden für die Tests verwendet.

1. Name: scp401;
   Anzahl der Index-Teilmengen: $|T| = 1000$;
   Gesamtkosten: $\kappa(T) = 50050$
   Anzahl der Indizes: $|M| = 200$;
   Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 20,0$;
   Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 4,0$;
   Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 2,0\%$.

2. Name: scp501;
   Anzahl der Index-Teilmengen: $|T| = 1998$;
   Gesamtkosten: $\kappa(T) = 101157$

Anzahl der Indizes: $|M| = 200$;
Durchschnittliche Überdeckung eines Index: $u_\varnothing =_{def} \sum_{t \in T} |t|/|M| = 40,0$;
Durchschnittliche Größe der Index-Teilmengen: $g_\varnothing =_{def} \sum_{t \in T} |t|/|T| = 4,0$;
Durchschnittliche Dichte: $d_\varnothing =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 2,0\%$.

3. Name: scp61;
   Anzahl der Index-Teilmengen: $|T| = 1000$;
   Gesamtkosten: $\kappa(T) = 50050$
   Anzahl der Indizes: $|M| = 200$;
   Durchschnittliche Überdeckung eines Index: $u_\varnothing =_{def} \sum_{t \in T} |t|/|M| = 49,2$;
   Durchschnittliche Größe der Index-Teilmengen: $g_\varnothing =_{def} \sum_{t \in T} |t|/|T| = 9,8$;
   Durchschnittliche Dichte: $d_\varnothing =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 4,9\%$.

4. Name: scpa1;
   Anzahl der Index-Teilmengen: $|T| = 3000$;
   Gesamtkosten: $\kappa(T) = 151762$
   Anzahl der Indizes: $|M| = 300$;
   Durchschnittliche Überdeckung eines Index: $u_\varnothing =_{def} \sum_{t \in T} |t|/|M| = 60,3$;
   Durchschnittliche Größe der Index-Teilmengen: $g_\varnothing =_{def} \sum_{t \in T} |t|/|T| = 6,0$;
   Durchschnittliche Dichte: $d_\varnothing =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 2,0\%$.

5. Name: scpb1;
   Anzahl der Index-Teilmengen: $|T| = 3000$;
   Gesamtkosten: $\kappa(T) = 151890$
   Anzahl der Indizes: $|M| = 300$;
   Durchschnittliche Überdeckung eines Index: $u_\varnothing =_{def} \sum_{t \in T} |t|/|M| = 149,7$;
   Durchschnittliche Größe der Index-Teilmengen: $g_\varnothing =_{def} \sum_{t \in T} |t|/|T| = 15,0$;
   Durchschnittliche Dichte: $d_\varnothing =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 5,0\%$.

6. Name: scpc1;
   Anzahl der Index-Teilmengen: $|T| = 4000$;
   Gesamtkosten: $\kappa(T) = 203551$
   Anzahl der Indizes: $|M| = 400$;
   Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 80,1$;
   Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 8,0$;
   Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 2,0\%$.

7. Name: scpd1;
   Anzahl der Index-Teilmengen: $|T| = 4000$;
   Gesamtkosten: $\kappa(T) = 203574$
   Anzahl der Indizes: $|M| = 400$;
   Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 200,4$;
   Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 20,0$;
   Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 5,0\%$.

8. Name: scnre1;
   Anzahl der Index-Teilmengen: $|T| = 5000$;
   Gesamtkosten: $\kappa(T) = 255616$
   Anzahl der Indizes: $|M| = 500$;
   Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 498,9$;
   Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 49,9$;
   Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 10,0\%$.

9. Name: scpnrf1;
   Anzahl der Index-Teilmengen: $|T| = 5000$;
   Gesamtkosten: $\kappa(T) = 249142$

Anzahl der Indizes: $|M| = 500$;

Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 998,6$;

Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 99,9$;

Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 20,0\%$.

10. Name: scpnrg1;

    Anzahl der Index-Teilmengen: $|T| = 10000$;

    Gesamtkosten: $\kappa(T) = 505504$

    Anzahl der Indizes: $|M| = 1000$;

    Durchschnittliche Überdeckung eines Index: $u_{\varnothing} =_{def} \sum_{t \in T} |t|/|M| = 199,5$;

    Durchschnittliche Größe der Index-Teilmengen: $g_{\varnothing} =_{def} \sum_{t \in T} |t|/|T| = 19,9$;

    Durchschnittliche Dichte: $d_{\varnothing} =_{def} \sum_{t \in T} |t|/(|M| \cdot |T|) = 2,0\%$.

Vor jedem Test wurde ein Referenzwert mit dem Iterated-Greedy-Algorithmus von Dietmar Lippold ermittelt, der zur Steuerung der Terminierungsbedingung benutzt wurde. Der Iterated-Greedy-Algorithmus führte dabei jeweils 100 Iterationen durch. Danach wurden in allen Testläufen 10 Läufe des evolutionären Zyklus mit allen Problemen des jeweiligen Problemsatzes durchgeführt. Die Ausgabe des evolutionären Algorithmus bestand aus zwei Zahlen: a) die *Bewertung ohne Baldwin-Lernen* des besten Individuums, das jemals in der Population existiert hat, b) die *tatsächliche Bewertung* des besten Individuums, das jemals innerhalb der Population existiert hat (siehe auch Abschnitt 4). Diese beiden Zahlen sind bei denjenigen Testläufen identisch, bei denen das Baldwin-Lernen ausgeschaltet war. Bei den Läufen mit eingeschaltetem Baldwin-Lernen gibt die erste Zahl aber eine wichtige Auskunft über die Ausprägung des Baldwin-Effekts in der Population, denn sie repräsentiert gewissermaßen die Fitness des Genoms, die sich laut Baldwin durch das Lernen verbessern sollte. Die zweite ist dagegen eher von praktischem Nutzen, weil sie die tatsächlich beste Lösung repräsentiert, die während des Laufs gefunden wurde.

Die Tabellen mit den Ergebnissen der Testläufe sind alle in der gleichen Weise aufgebaut: In der ersten Spalte (Name) steht der Name des bearbeiteten Problems. In der zweiten Spalte (Ref.) steht ein Referenzwert, der durch den Iterated-Greedy-Algorithmus mit 100 Iterationen berechnet wurde (das muss wegen unterschiedlicher Zufallsgenerator-Initialisierungen nicht der gleiche Wert sein wie der bei dem jeweiligen Testlauf ermittelte Wert; in allen Tabellen wurde aber der gleiche Referenzwert von einem einzigen Lauf des Iterated-Greedy-Algorithmus eingetragen). Die Spalten 3-12 (L1 - L10) enthalten die beste Bewertung der Population ohne Baldwin-Lernen in den Läufen 1-10 (das ist die oben beschriebene Bewertung des Typs a). Die Spalten 13-22 (L1B - L10B) enthalten die beste Bewertung der

Population mit Baldwin-Lernen in den Läufen 1-10 (das ist die oben beschriebene Bewertung des Typs b); bei Läufen ohne Baldwin-Lernen sind diese Spalten leergelassen.

Die Tabellenunterschrift charakterisiert zusammen mit der Beschreibung am Beginn des jeweiligen Unterabschnitts eindeutig die verwendeten Parameter und Operatoren des entsprechenden Laufs. Zusätzlich enthält sie eine Angabe der Prozessorzeit, die insgesamt für den Lauf benötigt wurde.

A.1 Ergebnisse der ersten 32 Testläufe

Die folgenden Ergebnisse sind durch Testläufe entstanden, bei denen die meisten Operatoren und Parameter konstant gehalten wurden. An 5 Stellen wurde eine binäre Einstellmöglichkeit gelassen. Die 5 Einstellmöglichkeiten waren:

- Komma-Strategie vs. Plus-Strategie,
- Nichtelitäre Strategie vs. elitäre Strategie durch Sperren der besten 4% der Individuen,
- Baldwin-Lernen mit Berücksichtigung von 3 Individuen aus der Mutationsnachbarschaft vs. kein Baldwin-Lernen,
- Beasley-Startpopulationserzeugung (Parameter $k = 5$) vs. randomisierte Startpopulationserzeugung und
- Lösen von Unicost-Problemen mit maximal 200 Generationen vs. Lösen von Multicost-Problemen mit maximal 800 Generationen.

Jede Kombination dieser Einstellmöglichkeiten wurde getestet. Die Tabellenunterschriften bezeichnen die Kombination der Einstellungen, die bei dem entsprechenden Lauf benutzt wurde.

Die übrigen benutzten Operatoren und Parameter waren bei jedem Testlauf konstant. Für alle Operatoren galt, dass die Eingangspopulation die gleiche Größe hatte wie die Ausgangspopulation (und zwar 400), außer für die Umweltselektion bei Plusstrategie, bei der die Eingangspopulation die doppelte Größe hatte. Folgende Operatoren und Parameter wurden benutzt:

- Populationsgröße: Die Populationsgröße war 400. Bei der Plusstrategie wuchs sie durch die Rekombination zwischenzeitlich auf 800.
- Elternselektion: Turnierselektion mit einer Turniergröße von 5% der Populationsgröße.
- Rekombination: Fitnessproportionaler Rekombinationsoperator R_b^ξ wie beschrieben.

- Diversität: Messung nach der beschriebenen stochastischen Methode am Beginn jedes Zyklus. Falls der Messwert (echt) kleiner als $0,02$ war, wurde statt der Mutation die Makromutation durchgeführt.
- Mutation: k-Bit-Flip-Mutationsoperator M_b^ξ wie beschrieben mit $k = 0,02 \cdot |T|$, also 2% der Anzahl der Bits im Genom. Die Wahrscheinlichkeit, überhaupt mutiert zu werden, beträgt für jedes Individuum 1.
- Makromutation: Wie Mutation, nur dass k auf 20% der Bits im Genom gesetzt wird.
- Umweltselektion: Lineare Rangbasierte Selektion.
- Reparatur: Die Reparatur wurde nach dem stochastischen Verfahren durchgeführt.
- Bewertung: Die Bewertung eines Individuums folgte dem einzigen dazu beschriebenen Algorithmus.
- Terminierung: Die Terminierung erfolgte, wenn der Referenzwert seit mindestens 200 Generationen erreicht war oder 200 bzw. 800 Generationen nach dem Start des Zyklus (bei Unicost bzw. Multicost-Problemen).

Es folgen die Daten der Testläufe:

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	27	25	25	25	27	25	25	25	27	27	-	-	-	-	-	-	-	-	-	-
scpclr11	27	27	25	27	23	27	28	28	27	23	25	-	-	-	-	-	-	-	-	-	-
scpclr12	23	23	26	29	29	27	28	30	28	29	29	-	-	-	-	-	-	-	-	-	-
scpclr13	26	30	29	31	28	29	29	29	29	29	26	-	-	-	-	-	-	-	-	-	-
scpcyc06	62	61	62	62	62	63	61	61	62	61	60	-	-	-	-	-	-	-	-	-	-
scpcyc07	144	148	156	156	154	156	153	155	152	153	155	-	-	-	-	-	-	-	-	-	-
scpcyc08	346	380	370	375	374	376	376	377	373	372	373	-	-	-	-	-	-	-	-	-	-
scpe1	5	5	5	5	5	5	5	5	5	5	5	-	-	-	-	-	-	-	-	-	-

Tabelle 40: Testlauf – Komma-Strategie, elitär, ohne Baldwin, Beasley-Startpop., Unicost; Prozessorzeit – 17,7 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	459	447	459	466	520	520	471	466	460	493	-	-	-	-	-	-	-	-	-	-
scp501	253	306	284	317	332	298	294	301	323	309	311	-	-	-	-	-	-	-	-	-	-
scp61	138	164	158	168	184	159	162	146	157	162	142	-	-	-	-	-	-	-	-	-	-
scpa1	255	327	332	315	292	317	295	334	337	325	299	-	-	-	-	-	-	-	-	-	-
scpb1	69	87	94	96	84	80	77	80	84	96	90	-	-	-	-	-	-	-	-	-	-
scpc1	227	320	336	327	335	346	352	360	328	378	311	-	-	-	-	-	-	-	-	-	-
scpd1	60	77	76	67	67	82	82	81	81	82	74	-	-	-	-	-	-	-	-	-	-
scpnre1	29	34	33	34	33	35	34	34	36	34	35	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	15	15	15	14	15	14	14	14	15	15	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	268	267	257	271	268	273	270	256	277	269	-	-	-	-	-	-	-	-	-	-

Tabelle 41: Testlauf – Komma-Strategie, elitär, ohne Baldwin, Beasley-Startpop., Multicost; Prozessorzeit – 66,1 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	25	25	26	27	25	26	26	25	25	28	-	-	-	-	-	-	-	-	-	-
scpclr11	27	26	27	28	27	27	27	23	28	27	27	-	-	-	-	-	-	-	-	-	-
scpclr12	23	28	29	27	29	29	29	28	29	28	28	-	-	-	-	-	-	-	-	-	-
scpclr13	26	29	30	29	28	30	30	29	29	30	30	-	-	-	-	-	-	-	-	-	-
scpcyc06	62	62	62	62	61	61	63	60	63	62	61	-	-	-	-	-	-	-	-	-	-
scpcyc07	144	153	155	154	154	154	153	154	157	154	153	-	-	-	-	-	-	-	-	-	-
scpcyc08	346	376	375	373	383	373	372	383	370	374	382	-	-	-	-	-	-	-	-	-	-
scpe1	5	6	5	5	6	6	5	5	5	5	5	-	-	-	-	-	-	-	-	-	-

Tabelle 42: Testlauf – Komma-Strategie, elitär, ohne Baldwin, rand. Startpop., Unicost; Prozessorzeit – 17,2 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	550	429	464	480	545	483	543	463	560	527	-	-	-	-	-	-	-	-	-	-
scp501	253	392	378	351	449	351	322	351	358	433	378	-	-	-	-	-	-	-	-	-	-
scp61	138	176	220	218	207	228	146	260	210	208	174	-	-	-	-	-	-	-	-	-	-
scpa1	255	572	512	678	435	384	487	481	571	519	397	-	-	-	-	-	-	-	-	-	-
scpb1	69	220	157	144	268	230	264	218	252	167	173	-	-	-	-	-	-	-	-	-	-
scpc1	227	764	825	858	944	767	747	971	816	638	1208	-	-	-	-	-	-	-	-	-	-
scpd1	60	242	228	392	336	380	243	342	288	353	320	-	-	-	-	-	-	-	-	-	-
scpnre1	29	413	102	102	155	237	194	173	163	121	140	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	31	27	33	29	25	20	38	28	41	24	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	2390	1767	2453	2300	2247	3281	2369	2092	2239	1952	-	-	-	-	-	-	-	-	-	-

Tabelle 43: Testlauf – Komma-Strategie, elitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 71,3 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	25	26	28	28	27	25	26	26	26	27	25	26	25	27	25	25	25	25	26	26
scpclr11	27	31	27	28	28	30	27	29	29	28	27	27	27	28	28	28	27	26	28	28	27
scpclr12	23	31	27	30	28	28	32	31	32	29	28	28	27	28	28	28	28	28	29	29	28
scpclr13	26	30	28	27	27	28	30	31	30	29	29	30	28	27	24	28	27	28	29	29	29
scpcyc06	62	62	63	64	62	62	62	66	63	63	62	62	61	62	62	60	60	62	62	62	60
scpcyc07	144	160	151	155	154	149	155	154	154	154	158	157	151	155	154	148	155	154	154	154	158
scpcyc08	346	374	377	370	376	370	380	377	373	375	382	374	377	370	375	370	379	377	373	375	379
scpe1	5	5	5	6	8	7	6	7	7	5	5	5	5	5	5	5	5	5	5	5	5

Tabelle 44: Testlauf – Komma-Strategie, elitär, Baldwin, Beasley-Startpop., Unicost; Prozessorzeit – 31,8 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	442	539	438	464	457	432	470	939	504	466	442	539	438	464	457	432	470	638	504	466
scp501	253	297	314	328	296	335	304	297	328	295	300	297	314	328	296	335	304	297	328	295	300
scp61	138	263	269	264	252	150	157	162	151	211	169	172	170	160	166	150	157	162	151	152	169
scpa1	255	311	351	315	311	304	342	303	309	331	317	311	351	315	311	304	342	303	309	331	317
scpb1	69	95	84	85	137	83	89	95	84	86	146	95	84	85	85	83	89	95	84	86	83
scpc1	227	337	341	332	371	357	322	339	341	322	354	337	341	332	371	357	322	339	341	322	354
scpd1	60	93	90	76	114	77	96	82	87	84	115	71	90	76	82	77	78	82	87	84	77
scpnre1	29	35	33	35	35	35	35	35	34	34	36	35	33	35	35	35	35	35	34	34	36
scpnrf1	14	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
scpnrg1	176	283	279	278	273	284	288	284	277	277	271	283	279	278	273	284	288	284	277	277	271

Tabelle 45: Testlauf – Komma-Strategie, elitär, Baldwin, Beasley-Startpop., Multicost; Prozessorzeit – 257,8 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	26	25	27	27	30	26	25	25	25	25	25	25	25	27	27	26	25	25	25	25
scpclr11	27	29	29	29	31	29	30	28	31	25	29	28	27	27	28	27	27	26	28	25	27
scpclr12	23	32	27	27	32	28	28	32	34	28	28	29	26	27	28	28	28	28	30	28	27
scpclr13	26	28	29	34	27	29	40	30	30	32	31	24	29	30	26	29	32	30	30	30	28
scpcyc06	62	62	65	63	64	60	61	62	60	61	60	62	62	63	63	60	61	62	60	61	60
scpcyc07	144	158	155	152	158	157	153	165	155	153	154	154	152	152	157	156	153	161	155	153	154
scpcyc08	346	389	373	370	370	375	382	375	377	376	372	382	373	370	370	375	379	374	377	376	372
scpe1	5	9	8	9	9	8	7	7	9	10	9	5	5	5	5	5	5	5	5	5	5

Tabelle 46: Testlauf – Komma-Strategie, elitär, Baldwin, rand. Startpop., Unicost; Prozessorzeit – 20,0 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	1942	1099	1042	1033	1829	1501	1112	1589	1531	1414	1190	720	609	733	931	849	753	902	875	781
scp501	253	1614	1376	1633	1456	1514	1545	1292	1452	1365	1861	597	625	717	689	684	721	577	606	509	639
scp61	138	813	588	784	631	965	803	660	1091	676	759	293	273	212	217	232	321	211	390	238	232
scpa1	255	2045	2061	2502	2579	1889	3113	2229	2391	2527	2090	700	724	938	859	874	1204	931	982	937	1004
scpb1	69	1626	1332	1723	1649	1361	1770	1672	1769	1546	1727	270	190	216	241	217	290	233	212	263	224
scpc1	227	2725	3115	2658	2363	2654	2862	2478	2561	3147	3036	1077	1145	1114	1044	1029	1320	820	992	1149	1078
scpd1	60	1827	2012	2207	1855	2059	1737	1724	2037	1607	2010	217	255	260	222	289	149	163	169	197	212
scpnre1	29	1380	1420	1411	1269	1282	1291	1581	1373	1357	1438	39	39	39	40	41	40	37	37	38	39
scpnrf1	14	750	705	948	939	784	706	859	709	750	892	16	17	18	18	17	16	17	17	16	17
scpnrg1	176	4644	5202	5035	5219	4460	5225	5060	5199	5751	5200	1089	1278	1101	1289	1043	1207	1038	1169	1276	1237

Tabelle 47: Testlauf – Komma-Strategie, elitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 189,3 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	28	26	26	25	28	25	27	25	28	27	-	-	-	-	-	-	-	-	-	-
scpclr11	27	28	27	26	24	26	27	25	28	25	24	-	-	-	-	-	-	-	-	-	-
scpclr12	23	30	29	27	29	28	29	28	27	28	27	-	-	-	-	-	-	-	-	-	-
scpclr13	26	30	28	28	28	29	29	29	30	28	29	-	-	-	-	-	-	-	-	-	-
scpcyc06	62	61	60	61	62	62	62	60	60	61	60	-	-	-	-	-	-	-	-	-	-
scpcyc07	144	151	150	153	144	151	153	153	154	152	148	-	-	-	-	-	-	-	-	-	-
scpcyc08	346	367	365	364	366	361	366	360	363	364	361	-	-	-	-	-	-	-	-	-	-
scpe1	5	5	5	5	5	5	5	5	5	5	5	-	-	-	-	-	-	-	-	-	-

Tabelle 48: Testlauf – Komma-Strategie, nichtelitär, ohne Baldwin, Beasley-Startpop., Unicost; Prozessorzeit – 7,6 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	561	721	539	544	643	670	694	630	675	616	-	-	-	-	-	-	-	-	-	-
scp501	253	414	442	446	434	439	427	436	441	383	429	-	-	-	-	-	-	-	-	-	-
scp61	138	210	202	193	206	210	210	215	209	205	213	-	-	-	-	-	-	-	-	-	-
scpa1	255	395	431	408	428	416	409	419	416	428	414	-	-	-	-	-	-	-	-	-	-
scpb1	69	97	97	101	104	103	108	105	106	109	105	-	-	-	-	-	-	-	-	-	-
scpc1	227	378	374	381	365	373	380	392	381	386	377	-	-	-	-	-	-	-	-	-	-
scpd1	60	92	88	88	90	93	94	88	94	89	86	-	-	-	-	-	-	-	-	-	-
scpnre1	29	35	38	35	33	35	37	35	36	36	38	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	16	15	16	15	16	15	16	15	16	15	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	279	281	286	283	279	273	285	270	274	283	-	-	-	-	-	-	-	-	-	-

Tabelle 49: Testlauf – Komma-Strategie, nichtelitär, ohne Baldwin, Beasley-Startpop., Multicost; Prozessorzeit – 114,7 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	27	25	26	26	25	27	27	25	26	27	-	-	-	-	-	-	-	-	-	-
scpclr11	27	23	28	28	27	23	23	27	27	26	26	-	-	-	-	-	-	-	-	-	-
scpclr12	23	28	28	29	29	29	27	29	29	23	28	-	-	-	-	-	-	-	-	-	-
scpclr13	26	30	29	29	30	29	29	28	29	28	28	-	-	-	-	-	-	-	-	-	-
scpcyc06	62	60	60	60	62	62	60	62	62	62	61	-	-	-	-	-	-	-	-	-	-
scpcyc07	144	156	154	152	150	154	154	150	155	154	151	-	-	-	-	-	-	-	-	-	-
scpcyc08	346	365	363	359	365	365	362	365	361	363	367	-	-	-	-	-	-	-	-	-	-
scpe1	5	5	5	5	5	5	5	5	5	5	5	-	-	-	-	-	-	-	-	-	-

Tabelle 50: Testlauf – Komma-Strategie, nichtelitär, ohne Baldwin, rand. Startpop., Unicost; Prozessorzeit – 16,6 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	730	743	758	666	737	738	813	671	622	634	-	-	-	-	-	-	-	-	-	-
scp501	253	550	646	578	649	586	491	750	593	619	655	-	-	-	-	-	-	-	-	-	-
scp61	138	338	369	255	270	249	325	276	289	268	328	-	-	-	-	-	-	-	-	-	-
scpa1	255	1490	1433	1493	1459	1585	1372	1424	765	1377	1415	-	-	-	-	-	-	-	-	-	-
scpb1	69	1128	1080	1118	1062	1099	1087	1102	1098	991	1088	-	-	-	-	-	-	-	-	-	-
scpc1	227	2232	2389	2295	2379	2316	2276	2339	2197	2175	2058	-	-	-	-	-	-	-	-	-	-
scpd1	60	1375	1270	1321	1299	1338	1286	1270	1330	1311	1193	-	-	-	-	-	-	-	-	-	-
scpnre1	29	751	756	784	725	792	832	795	856	758	796	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	329	318	332	338	330	331	332	312	289	230	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	4244	4321	4252	4296	4335	4244	4354	4282	4169	4328	-	-	-	-	-	-	-	-	-	-

Tabelle 51: Testlauf – Komma-Strategie, nichtelitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 114,6 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	25	26	25	26	25	25	26	25	25	27	25	26	25	26	25	25	26	25	25	27
scpclr11	27	26	25	27	27	27	24	26	27	28	27	26	25	27	27	27	24	26	27	28	27
scpclr12	23	28	25	28	28	28	28	29	27	28	29	28	25	28	28	28	28	29	27	28	29
scpclr13	26	25	29	28	27	30	30	28	28	27	27	25	29	28	27	30	30	28	28	27	27
scpcyc06	62	61	62	60	61	62	60	60	60	62	62	61	62	60	61	62	60	60	60	62	62
scpcyc07	144	150	149	155	151	153	151	153	152	152	154	150	149	155	151	153	151	153	152	152	154
scpcyc08	346	361	362	364	361	362	362	365	363	363	363	361	362	364	361	362	362	365	363	363	363
scpe1	5	5	5	5	6	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5

Tabelle 52: Testlauf – Komma-Strategie, nichtelitär, Baldwin, Beasley-Startpop., Unicost; Prozessorzeit – 21,4 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	691	724	788	864	862	825	839	874	833	852	691	614	625	702	681	647	691	640	677	717
scp501	253	407	427	385	449	416	430	433	417	392	431	407	423	385	449	392	430	432	417	392	431
scp61	138	206	201	225	241	202	197	210	179	201	216	200	201	173	201	202	194	210	178	201	170
scpa1	255	426	411	382	393	422	420	423	420	408	411	426	411	382	393	422	420	423	420	408	411
scpb1	69	105	100	108	106	98	103	106	95	102	100	105	100	108	106	98	103	106	95	102	100
scpc1	227	360	370	390	392	384	367	363	371	370	380	360	370	390	392	384	367	363	371	370	380
scpd1	60	91	77	89	124	93	85	86	91	91	129	91	77	89	85	93	85	86	91	91	89
scpnre1	29	34	37	33	36	35	34	36	35	38	36	34	37	33	36	34	34	36	35	38	36
scpnrf1	14	16	15	15	15	16	16	15	15	15	15	16	15	15	15	16	16	15	15	15	15
scpnrg1	176	285	278	282	284	275	281	280	278	279	280	285	278	282	284	275	281	280	278	279	280

Tabelle 53: Testlauf – Komma-Strategie, nichtelitär, Baldwin, Beasley-Startpop., Multicost; Prozessorzeit – 224,0 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	25	26	25	25	26	25	26	26	25	28	25	26	25	25	26	25	26	26	25	28
scpclr11	27	27	27	25	27	27	24	27	27	27	27	27	27	25	27	27	24	27	27	27	27
scpclr12	23	24	28	28	23	28	28	28	27	28	28	24	28	28	23	28	28	28	27	28	28
scpclr13	26	30	29	29	29	29	28	30	30	28	28	30	29	29	29	29	28	30	30	28	28
scpcyc06	62	60	61	60	62	62	62	62	62	62	62	60	61	60	62	62	62	62	62	62	62
scpcyc07	144	153	153	152	154	150	152	149	152	153	152	153	153	152	154	150	152	149	152	153	152
scpcyc08	346	366	360	365	364	363	361	366	366	360	363	366	360	365	364	363	361	366	366	360	363
scpe1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5

Tabelle 54: Testlauf – Komma-Strategie, nichtelitär, Baldwin, rand. Startpop., Unicost; Prozessorzeit – 31,6 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	1051	912	866	906	921	888	900	1164	1073	975	792	735	718	749	700	702	664	896	872	800
scp501	253	635	610	919	713	546	609	679	613	727	669	470	443	578	530	413	503	440	486	475	548
scp61	138	281	335	316	335	367	335	321	321	357	371	200	218	229	202	269	257	188	208	225	253
scpa1	255	1534	1666	1692	1576	1779	1701	1683	1535	1705	1830	816	923	974	1017	967	980	861	846	866	841
scpb1	69	1202	1153	1258	1299	1149	1200	1151	1174	1111	1263	259	348	291	304	256	279	231	299	313	310
scpc1	227	2498	2487	2259	2430	2462	2427	2349	2535	2591	2414	1321	1286	1381	1401	1363	1401	1219	1390	1403	1289
scpd1	60	1383	1532	1520	1540	1482	1332	1522	1437	1532	1432	275	224	302	264	301	276	239	268	237	290
scpnre1	29	972	976	985	969	789	838	1052	979	1074	925	41	39	42	40	38	39	41	41	33	38
scpnrf1	14	462	444	460	466	520	492	411	454	421	541	17	17	17	18	16	18	17	17	17	18
scpnrg1	176	4571	4559	4803	4401	4599	4581	4431	4341	4397	4506	1378	1403	1419	1224	1400	1426	1377	1385	1393	1367

Tabelle 55: Testlauf – Komma-Strategie, nichtelitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 206,3 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	27	27	28	25	25	25	26	25	25	27	-	-	-	-	-	-	-	-	-	-
scpclr11	27	27	27	28	28	26	28	27	29	24	27	-	-	-	-	-	-	-	-	-	-
scpclr12	23	29	27	26	27	25	30	27	28	29	25	-	-	-	-	-	-	-	-	-	-
scpclr13	26	28	23	28	29	28	28	30	29	31	28	-	-	-	-	-	-	-	-	-	-
scpcyc06	62	60	62	62	60	62	63	62	60	62	62	-	-	-	-	-	-	-	-	-	-
scpcyc07	144	155	155	155	155	152	154	158	154	155	150	-	-	-	-	-	-	-	-	-	-
scpcyc08	346	373	377	372	374	372	381	379	377	379	376	-	-	-	-	-	-	-	-	-	-
scpe1	5	5	5	5	5	5	5	5	5	5	5	-	-	-	-	-	-	-	-	-	-

Tabelle 56: Testlauf – Plus-Strategie, elitär, ohne Baldwin, Beasley-Startpop., Unicost; Prozessorzeit – 18,5 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	471	578	450	457	455	506	533	541	453	573	-	-	-	-	-	-	-	-	-	-
scp501	253	368	394	426	430	329	377	374	427	313	352	-	-	-	-	-	-	-	-	-	-
scp61	138	166	165	169	168	184	165	163	179	159	143	-	-	-	-	-	-	-	-	-	-
scpa1	255	405	424	425	426	410	424	429	404	410	423	-	-	-	-	-	-	-	-	-	-
scpb1	69	101	101	105	100	102	105	95	97	97	103	-	-	-	-	-	-	-	-	-	-
scpc1	227	375	369	388	396	399	380	387	392	383	383	-	-	-	-	-	-	-	-	-	-
scpd1	60	85	76	89	86	92	85	94	91	93	76	-	-	-	-	-	-	-	-	-	-
scpnre1	29	34	35	36	36	33	36	33	36	34	34	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	15	15	15	15	15	15	14	15	15	15	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	280	280	275	281	285	285	283	281	282	286	-	-	-	-	-	-	-	-	-	-

Tabelle 57: Testlauf – Plus-Strategie, elitär, ohne Baldwin, Beasley-Startpop., Multicost; Prozessorzeit – 65,5 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	27	28	28	27	25	28	25	26	27	27	-	-	-	-	-	-	-	-	-	-
scpclr11	27	28	27	27	28	28	27	28	27	27	28	-	-	-	-	-	-	-	-	-	-
scpclr12	23	29	28	29	28	29	28	29	27	29	29	-	-	-	-	-	-	-	-	-	-
scpclr13	26	29	31	39	31	30	30	30	30	29	29	-	-	-	-	-	-	-	-	-	-
scpcyc06	62	63	61	61	62	63	62	61	60	62	62	-	-	-	-	-	-	-	-	-	-
scpcyc07	144	154	154	155	154	153	155	151	152	155	151	-	-	-	-	-	-	-	-	-	-
scpcyc08	346	382	375	378	381	377	377	380	371	375	378	-	-	-	-	-	-	-	-	-	-
scpe1	5	5	6	5	5	5	5	6	5	5	5	-	-	-	-	-	-	-	-	-	-

Tabelle 58: Testlauf – Plus-Strategie, elitär, ohne Baldwin, rand. Startpop., Unicost; Prozessorzeit – 18,1 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	492	547	528	521	545	550	561	446	552	507	-	-	-	-	-	-	-	-	-	-
scp501	253	468	506	588	574	565	446	463	448	471	476	-	-	-	-	-	-	-	-	-	-
scp61	138	186	177	214	182	197	176	181	212	211	168	-	-	-	-	-	-	-	-	-	-
scpa1	255	1034	680	1123	899	805	1030	967	783	862	928	-	-	-	-	-	-	-	-	-	-
scpb1	69	503	406	464	420	614	707	439	487	744	407	-	-	-	-	-	-	-	-	-	-
scpc1	227	1204	1363	1390	1361	1188	1548	1486	1465	1201	1637	-	-	-	-	-	-	-	-	-	-
scpd1	60	674	626	516	787	505	578	641	588	491	529	-	-	-	-	-	-	-	-	-	-
scpnre1	29	469	282	450	594	327	368	367	207	247	412	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	91	105	102	162	99	198	171	131	93	78	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	2272	1962	1974	2357	3842	1876	2636	2269	3038	1952	-	-	-	-	-	-	-	-	-	-

Tabelle 59: Testlauf – Plus-Strategie, elitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 55,0 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	29	28	25	28	26	25	26	29	31	29	27	27	25	26	25	25	25	27	28	25
scpclr11	27	32	34	32	27	23	30	36	39	38	33	28	29	28	26	23	26	26	28	28	29
scpclr12	23	31	28	29	29	28	36	28	28	29	29	29	28	29	28	28	30	28	28	28	28
scpclr13	26	33	28	27	32	27	34	36	34	30	28	28	28	27	30	27	30	30	30	30	27
scpcyc06	62	61	63	64	64	64	61	63	63	63	62	60	62	62	62	62	60	63	62	62	62
scpcyc07	144	155	155	154	156	151	160	161	162	159	158	154	154	150	156	151	156	160	157	156	156
scpcyc08	346	381	379	392	381	392	383	377	382	380	389	380	379	385	380	386	382	377	376	379	384
scpe1	5	5	5	6	6	6	7	5	7	6	7	5	5	5	5	5	5	5	5	5	5

Tabelle 60: Testlauf – Plus-Strategie, elitär, Baldwin, Beasley-Startpop., Unicost; Prozessorzeit – 33,7 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	497	487	895	475	475	472	897	1058	885	933	497	487	652	475	475	472	745	706	538	780
scp501	253	409	424	359	336	360	366	362	398	360	401	409	424	359	336	360	366	362	398	360	401
scp61	138	155	186	214	219	198	166	192	156	169	305	155	186	169	170	179	166	154	156	169	183
scpa1	255	421	415	415	422	413	393	424	429	408	419	421	415	415	422	413	393	424	429	408	419
scpb1	69	105	103	105	100	100	101	102	107	102	104	105	103	89	100	100	101	102	107	102	104
scpc1	227	362	373	397	386	371	382	381	385	373	386	362	373	397	386	371	382	381	385	373	386
scpd1	60	88	90	87	91	87	92	110	90	89	89	82	78	87	91	87	92	76	90	89	89
scpnre1	29	36	32	34	33	34	36	35	35	36	36	36	32	34	33	34	36	35	35	36	36
scpnrf1	14	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
scpnrg1	176	277	270	286	285	277	280	278	292	270	284	277	270	286	285	277	280	278	292	270	284

Tabelle 61: Testlauf – Plus-Strategie, elitär, Baldwin, Beasley-Startpop., Multicost; Prozessorzeit – 244,4 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	26	27	28	26	25	27	30	26	27	28	26	27	28	25	25	26	27	25	26	25
scpclr11	27	28	32	28	31	31	30	28	28	31	30	28	29	27	28	27	28	27	27	28	27
scpclr12	23	28	33	28	36	30	30	30	32	32	29	28	29	28	30	28	28	29	30	29	28
scpclr13	26	32	36	29	31	31	30	33	34	29	28	30	30	29	29	28	30	29	28	28	28
scpcyc06	62	65	61	65	61	64	65	62	62	64	62	63	61	60	61	63	63	60	62	62	62
scpcyc07	144	154	157	155	155	165	163	152	157	156	155	154	154	155	155	159	159	152	156	154	155
scpcyc08	346	378	387	378	380	377	374	376	397	387	380	378	386	378	379	377	373	376	392	384	380
scpe1	5	8	9	9	8	7	8	9	8	9	9	5	5	5	5	5	5	5	5	5	5

Tabelle 62: Testlauf – Plus-Strategie, elitär, Baldwin, rand. Startpop., Unicost; Prozessorzeit – 32,2 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	1639	1448	1723	1878	1378	1343	1401	1443	1492	1337	930	928	1157	1001	885	885	797	935	978	675
scp501	253	1384	1562	1786	1216	1887	1880	1890	1974	2172	1648	550	699	792	633	879	817	777	806	847	792
scp61	138	623	882	559	1084	1062	1122	619	1092	910	817	231	344	293	365	346	376	214	398	283	243
scpa1	255	2459	2232	2291	2451	2186	2551	2630	2649	2348	2304	905	904	1060	969	925	990	1082	1222	964	1091
scpb1	69	1438	1807	1768	1921	1651	1523	1619	1593	1861	1690	184	188	211	219	229	240	260	276	213	263
scpc1	227	2462	3243	3060	2783	3208	2827	2974	2964	3165	4071	987	1277	1265	1084	1108	1153	1270	1170	1321	1410
scpd1	60	2030	2053	2092	1670	1973	1767	1974	2053	1670	1922	217	241	232	210	214	226	189	254	154	227
scpnre1	29	1281	1391	1216	1465	1234	1522	1551	1287	1547	1301	40	39	39	38	37	35	40	39	41	40
scpnrf1	14	782	879	796	660	787	776	694	853	828	657	17	17	17	18	16	17	17	16	17	16
scpnrg1	176	5144	5311	5047	5378	5019	5428	5241	4937	5734	5248	1281	1270	1131	1267	1116	1337	1126	1056	1252	1410

Tabelle 63: Testlauf – Plus-Strategie, elitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 169,8 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	25	25	25	26	27	25	25	28	25	25	-	-	-	-	-	-	-	-	-	-
scpclr11	27	28	26	23	28	25	28	27	27	28	28	-	-	-	-	-	-	-	-	-	-
scpclr12	23	26	29	28	27	28	28	29	28	28	29	-	-	-	-	-	-	-	-	-	-
scpclr13	26	30	29	29	29	30	30	29	28	30	28	-	-	-	-	-	-	-	-	-	-
scpcyc06	62	62	62	63	62	63	62	60	60	62	62	-	-	-	-	-	-	-	-	-	-
scpcyc07	144	155	151	152	153	152	155	158	153	148	154	-	-	-	-	-	-	-	-	-	-
scpcyc08	346	367	372	365	372	370	370	374	375	373	367	-	-	-	-	-	-	-	-	-	-
scpe1	5	5	5	5	5	5	5	5	5	5	5	-	-	-	-	-	-	-	-	-	-

Tabelle 64: Testlauf – Plus-Strategie, nichtelitär, ohne Baldwin, Beasley-Startpop., Unicost; Prozessorzeit – 18,6 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	589	539	553	534	517	516	638	542	601	498	-	-	-	-	-	-	-	-	-	-
scp501	253	423	441	418	412	425	428	428	433	447	452	-	-	-	-	-	-	-	-	-	-
scp61	138	192	188	209	216	194	209	193	218	194	184	-	-	-	-	-	-	-	-	-	-
scpa1	255	414	410	434	392	429	427	421	434	415	422	-	-	-	-	-	-	-	-	-	-
scpb1	69	106	100	107	103	106	105	109	102	104	108	-	-	-	-	-	-	-	-	-	-
scpc1	227	384	375	385	371	364	380	377	374	373	383	-	-	-	-	-	-	-	-	-	-
scpd1	60	94	92	90	93	90	90	87	92	93	93	-	-	-	-	-	-	-	-	-	-
scpnre1	29	36	37	36	34	37	38	38	37	36	36	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	15	15	15	16	15	14	15	15	15	15	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	280	283	273	274	282	275	279	278	271	287	-	-	-	-	-	-	-	-	-	-

Tabelle 65: Testlauf – Plus-Strategie, nichtelitär, ohne Baldwin, Beasley-Startpop., Multicost; Prozessorzeit – 99,7 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	25	26	27	27	27	25	26	28	25	25	-	-	-	-	-	-	-	-	-	-
scpclr11	27	27	27	23	28	28	28	27	28	27	28	-	-	-	-	-	-	-	-	-	-
scpclr12	23	29	29	29	28	29	28	28	29	29	30	-	-	-	-	-	-	-	-	-	-
scpclr13	26	30	29	30	30	30	30	28	30	30	28	-	-	-	-	-	-	-	-	-	-
scpcyc06	62	60	62	62	60	61	62	61	60	62	60	-	-	-	-	-	-	-	-	-	-
scpcyc07	144	155	154	153	147	151	151	147	153	153	152	-	-	-	-	-	-	-	-	-	-
scpcyc08	346	370	370	370	367	370	369	371	367	372	371	-	-	-	-	-	-	-	-	-	-
scpe1	5	6	5	5	5	5	6	6	5	5	5	-	-	-	-	-	-	-	-	-	-

Tabelle 66: Testlauf – Plus-Strategie, nichtelitär, ohne Baldwin, rand. Startpop., Unicost; Prozessorzeit – 17,9 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	607	525	560	540	646	619	627	752	676	633	-	-	-	-	-	-	-	-	-	-
scp501	253	550	744	649	513	660	609	614	675	688	627	-	-	-	-	-	-	-	-	-	-
scp61	138	206	198	257	235	246	261	336	330	236	227	-	-	-	-	-	-	-	-	-	-
scpa1	255	1144	1258	1252	1017	1001	1103	1147	1187	1069	1131	-	-	-	-	-	-	-	-	-	-
scpb1	69	896	913	907	851	884	930	732	944	937	866	-	-	-	-	-	-	-	-	-	-
scpc1	227	1851	1731	1805	1849	2013	1921	1649	2003	1857	1923	-	-	-	-	-	-	-	-	-	-
scpd1	60	1213	1121	1259	1045	1180	1170	1214	1129	1204	1186	-	-	-	-	-	-	-	-	-	-
scpnre1	29	771	772	772	805	770	803	796	792	766	809	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	347	319	319	244	340	330	337	316	296	328	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	4219	4155	4194	4122	4143	4198	4037	4142	4142	4232	-	-	-	-	-	-	-	-	-	-

Tabelle 67: Testlauf – Plus-Strategie, nichtelitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 104,5 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	25	25	25	26	25	25	27	25	25	25	25	25	25	26	25	25	26	25	25	25
scpclr11	27	25	27	27	26	27	27	23	28	27	28	25	27	27	26	27	27	23	28	27	28
scpclr12	23	28	29	28	28	28	28	28	28	27	27	28	29	28	28	28	28	28	28	27	27
scpclr13	26	27	28	27	29	27	30	30	29	29	29	27	28	27	29	27	30	30	29	29	29
scpcyc06	62	62	62	62	63	62	62	60	60	60	62	62	62	62	63	62	62	60	60	60	62
scpcyc07	144	155	153	154	153	152	152	153	153	153	151	155	153	154	153	152	152	153	153	153	151
scpcyc08	346	371	365	370	371	371	371	373	367	369	371	371	365	370	371	371	371	373	367	369	371
scpe1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5

Tabelle 68: Testlauf – Plus-Strategie, nichtelitär, Baldwin, Beasley-Startpop., Unicost; Prozessorzeit – 34,5 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	501	568	602	546	607	876	623	547	891	550	501	568	602	546	607	633	623	547	598	550
scp501	253	437	449	442	423	440	417	402	442	429	460	437	410	442	423	440	417	402	442	429	429
scp61	138	210	204	213	194	208	174	203	213	212	182	206	182	173	194	208	174	203	169	180	182
scpa1	255	422	431	428	421	425	394	432	426	436	376	422	431	428	421	425	394	432	426	436	376
scpb1	69	110	108	103	109	102	105	95	103	104	107	110	108	95	108	102	105	95	103	104	107
scpc1	227	376	381	383	370	381	381	388	396	379	375	376	381	383	370	381	381	388	396	379	375
scpd1	60	90	89	93	88	89	87	88	91	93	93	90	87	93	88	89	87	88	91	93	88
scpnre1	29	36	35	35	37	36	35	36	36	36	34	36	35	35	37	36	35	36	36	36	34
scpnrf1	14	16	15	16	15	16	14	15	15	15	15	16	15	16	15	16	14	15	15	15	15
scpnrg1	176	273	278	285	281	275	278	280	279	282	279	273	278	285	281	275	278	280	279	282	279

Tabelle 69: Testlauf – Plus-Strategie, nichtelitär, Baldwin, Beasley-Startpop., Multicost; Prozessorzeit – 202,9 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scpclr10	25	27	25	25	25	27	25	25	25	25	25	27	25	25	25	27	25	25	25	25	25
scpclr11	27	25	27	27	27	28	26	27	28	27	27	25	27	27	27	28	26	27	28	27	27
scpclr12	23	28	28	29	29	29	28	28	27	28	29	28	28	29	29	29	28	28	27	28	29
scpclr13	26	28	30	31	29	30	29	28	29	31	29	28	30	30	29	30	29	28	29	31	29
scpcyc06	62	60	62	61	61	62	62	62	62	63	61	60	62	61	61	62	62	62	62	63	61
scpcyc07	144	153	153	153	154	154	151	152	153	152	153	153	153	153	154	154	151	152	153	152	153
scpcyc08	346	371	364	366	366	368	363	367	372	368	371	371	364	366	366	368	363	367	372	368	371
scpe1	5	6	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5

Tabelle 70: Testlauf – Plus-Strategie, nichtelitär, Baldwin, rand. Startpop., Unicost; Prozessorzeit – 33,5 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	1099	1024	1066	985	1080	1136	1092	898	1178	1243	736	732	838	788	733	829	772	701	804	777
scp501	253	1029	989	933	837	1041	959	1115	1088	977	808	555	496	594	466	576	565	584	711	481	486
scp61	138	483	554	528	580	594	442	484	475	548	750	202	280	216	310	277	217	251	237	234	302
scpa1	255	1659	1679	1522	1665	1564	1816	1605	1564	1478	1766	907	942	874	854	860	845	903	830	734	879
scpb1	69	1314	1254	1308	1195	1156	1159	1353	1181	1146	1240	278	271	322	303	245	261	294	297	294	225
scpc1	227	2563	2382	2355	2506	2585	2495	2360	2400	2450	2634	1217	1170	1148	1254	1325	1142	1213	1008	1164	1220
scpd1	60	1574	1550	1322	1477	1560	1611	1511	1545	1471	1340	245	270	240	280	276	250	258	312	244	237
scpnre1	29	1036	897	1037	1075	1149	1090	961	1078	1158	1028	40	38	38	40	41	37	40	43	40	42
scpnrf1	14	464	587	554	673	523	397	492	607	590	566	17	18	17	18	17	18	18	18	17	17
scpnrg1	176	4409	4810	4861	4768	4723	4701	4702	4629	4693	4631	1352	1454	1382	1331	1297	1348	1356	1384	1385	1286

Tabelle 71: Testlauf – Plus-Strategie, nichtelitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 213,2 h

A.2 Ergebnisse des Langzeittest

Der im Folgenden dargestellte Test benutzte bis auf die Terminierung die gleichen Operatoren und Parameter wie der entsprechende Test aus Abschnitt A.1 (Tabelle 41). Die Terminierung wurde aber 1000 Generationen nach Erreichen des Referenzwerts bzw. spätestens 8000 Generationen nach Beginn des Zyklus eingeleitet.

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	440	435	452	431	449	447	436	432	443	434	-	-	-	-	-	-	-	-	-	-
scp501	253	294	281	297	273	295	311	273	285	269	278	-	-	-	-	-	-	-	-	-	-
scp61	138	154	159	167	146	148	151	138	162	162	153	-	-	-	-	-	-	-	-	-	-
scpa1	255	289	273	288	294	291	271	274	297	315	282	-	-	-	-	-	-	-	-	-	-
scpb1	69	90	80	85	93	82	83	87	96	93	83	-	-	-	-	-	-	-	-	-	-
scpc1	227	291	311	269	272	320	330	338	280	310	299	-	-	-	-	-	-	-	-	-	-
scpd1	60	76	74	73	67	82	75	69	76	67	70	-	-	-	-	-	-	-	-	-	-
scpnre1	29	31	36	33	34	33	35	35	33	32	35	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	15	15	15	14	14	15	15	15	15	15	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	243	253	243	243	247	284	247	237	245	235	-	-	-	-	-	-	-	-	-	-

Tabelle 72: Testlauf (Lang) – Komma-Strategie, elitär, ohne Baldwin, Beasley-Startpop., Multicost; Prozessorzeit – 668,4 h

A.3 Ergebnisse der Tests mit kleinerer linearer Mutationsrate

Die im Folgenden dargestellten Tests benutzten bis auf die Mutation die gleichen Parameter und Operatoren wie die entsprechenden in Abschnitt A.1 beschriebenen Tests (Tabellen 43 und 47). Die Mutationsrate wurde aber schrittweise herabgesetzt. Während die oben bereits beschriebenen Tests eine Mutationsrate von 2% der Genomlänge hatten, wurden hier die Raten 1%, 0,75%, 0,5%, 0,25%, 0,1% der Genomlänge eingesetzt. Diese Raten wurden jeweils für einen Test mit und einen ohne Baldwin-Lernen benutzt.

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	461	494	496	440	534	493	448	460	497	483	-	-	-	-	-	-	-	-	-	-
scp501	253	370	323	320	440	316	372	338	305	349	388	-	-	-	-	-	-	-	-	-	-
scp61	138	178	260	222	227	179	167	152	173	175	145	-	-	-	-	-	-	-	-	-	-
scpa1	255	517	335	481	460	389	396	350	433	389	504	-	-	-	-	-	-	-	-	-	-
scpb1	69	182	155	158	133	162	142	180	230	176	141	-	-	-	-	-	-	-	-	-	-
scpc1	227	479	388	473	482	749	672	350	609	665	654	-	-	-	-	-	-	-	-	-	-
scpd1	60	223	197	160	228	200	153	119	132	166	154	-	-	-	-	-	-	-	-	-	-
scpnre1	29	72	120	113	98	109	130	89	52	87	110	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	30	39	25	27	31	30	34	39	23	38	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	2285	1762	1824	1711	2130	1757	1543	2137	1654	2105	-	-	-	-	-	-	-	-	-	-

Tabelle 73: Testlauf (Mut: 1%) – Komma-Strategie, elitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 56,3 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	472	482	457	448	475	494	462	448	484	451	-	-	-	-	-	-	-	-	-	-
scp501	253	287	315	382	480	393	344	389	347	374	311	-	-	-	-	-	-	-	-	-	-
scp61	138	164	295	151	146	146	188	181	144	161	236	-	-	-	-	-	-	-	-	-	-
scpa1	255	334	417	433	304	429	405	459	312	319	372	-	-	-	-	-	-	-	-	-	-
scpb1	69	134	160	203	131	220	179	168	145	120	119	-	-	-	-	-	-	-	-	-	-
scpc1	227	421	424	689	390	453	698	578	553	339	478	-	-	-	-	-	-	-	-	-	-
scpd1	60	177	121	282	204	154	191	202	188	171	169	-	-	-	-	-	-	-	-	-	-
scpnre1	29	127	89	125	144	114	114	183	67	99	70	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	33	50	38	29	34	34	23	48	23	26	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	1329	1159	1635	1164	1556	1751	1122	1211	1454	1454	-	-	-	-	-	-	-	-	-	-

Tabelle 74: Testlauf (Mut: 0,75%) – Komma-Strategie, elitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 56,3 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	438	460	431	432	458	432	473	467	468	454	-	-	-	-	-	-	-	-	-	-
scp501	253	328	291	341	331	439	291	299	308	395	296	-	-	-	-	-	-	-	-	-	-
scp61	138	184	163	292	244	156	148	167	204	160	144	-	-	-	-	-	-	-	-	-	-
scpa1	255	376	307	422	353	398	411	437	317	383	382	-	-	-	-	-	-	-	-	-	-
scpb1	69	120	95	118	122	113	164	127	108	106	104	-	-	-	-	-	-	-	-	-	-
scpc1	227	356	404	429	443	468	426	383	639	514	526	-	-	-	-	-	-	-	-	-	-
scpd1	60	124	183	159	117	113	162	216	85	145	207	-	-	-	-	-	-	-	-	-	-
scpnre1	29	61	103	73	76	148	66	112	104	102	74	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	39	38	52	32	34	26	48	28	47	22	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	718	914	814	898	604	830	651	842	833	1065	-	-	-	-	-	-	-	-	-	-

Tabelle 75: Testlauf (Mut: 0,5%) – Komma-Strategie, elitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 54,0 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	520	493	461	496	508	439	453	429	455	442	-	-	-	-	-	-	-	-	-	-
scp501	253	363	400	389	363	386	323	385	341	374	370	-	-	-	-	-	-	-	-	-	-
scp61	138	156	185	192	177	171	183	175	145	229	229	-	-	-	-	-	-	-	-	-	-
scpa1	255	433	434	349	366	337	357	438	437	464	362	-	-	-	-	-	-	-	-	-	-
scpb1	69	120	136	150	139	124	85	144	135	96	204	-	-	-	-	-	-	-	-	-	-
scpc1	227	407	519	422	372	387	373	386	472	614	306	-	-	-	-	-	-	-	-	-	-
scpd1	60	106	111	111	156	149	167	107	106	142	96	-	-	-	-	-	-	-	-	-	-
scpnre1	29	46	63	52	90	84	47	91	52	91	67	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	20	24	33	29	45	29	32	37	31	31	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	611	532	800	507	640	554	529	600	730	527	-	-	-	-	-	-	-	-	-	-

Tabelle 76: Testlauf (Mut: 0,25%) – Komma-Strategie, elitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 49,8 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	436	502	482	436	504	488	489	484	435	498	-	-	-	-	-	-	-	-	-	-
scp501	253	285	409	356	335	355	373	452	335	393	358	-	-	-	-	-	-	-	-	-	-
scp61	138	151	181	188	187	207	206	227	198	157	164	-	-	-	-	-	-	-	-	-	-
scpa1	255	350	500	408	328	316	310	618	433	346	469	-	-	-	-	-	-	-	-	-	-
scpb1	69	108	153	149	126	149	206	96	121	264	229	-	-	-	-	-	-	-	-	-	-
scpc1	227	479	386	339	412	586	618	389	529	439	482	-	-	-	-	-	-	-	-	-	-
scpd1	60	136	228	184	157	162	200	159	267	134	145	-	-	-	-	-	-	-	-	-	-
scpnre1	29	52	62	62	51	46	73	41	61	46	80	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	25	26	27	18	24	19	30	31	24	27	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	703	667	624	515	660	528	482	557	595	518	-	-	-	-	-	-	-	-	-	-

Tabelle 77: Testlauf (Mut: 0,1%) – Komma-Strategie, elitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 49,6 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	944	840	1064	1146	1163	1158	909	1054	1012	1028	593	630	766	725	669	694	553	716	686	598
scp501	253	945	854	783	1141	954	1212	888	940	867	1323	506	399	380	604	471	500	500	493	495	818
scp61	138	581	413	406	380	437	582	686	477	677	431	221	200	202	215	219	228	256	208	270	166
scpa1	255	2024	1484	2083	1734	2215	1517	1492	1626	1633	1740	1086	815	1042	883	1085	982	713	794	847	956
scpb1	69	1323	1314	1102	1183	1251	831	1276	1470	1030	765	295	450	305	324	469	221	415	458	329	252
scpc1	227	1872	2458	2026	2330	1661	1786	1651	2013	2180	1950	1007	1402	1102	1342	1022	964	1046	1332	1110	1108
scpd1	60	1439	1897	1379	1793	1356	1498	1651	1882	1627	1660	401	289	351	394	285	415	375	478	368	440
scpnre1	29	1161	1215	1311	1377	1380	1187	1171	1035	1195	1232	67	60	57	81	53	62	66	58	70	54
scpnrf1	14	565	757	706	815	842	962	770	730	883	725	14	15	14	15	15	15	15	15	15	14
scpnrg1	176	3926	4073	5148	4263	4665	4420	4722	4202	4537	4168	1563	1457	1771	1600	2108	2026	1866	1820	1963	1811

Tabelle 78: Testlauf (Mut: 1%) – Komma-Strategie, elitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 151,4 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	761	893	986	1208	859	1077	912	860	1103	925	497	534	625	579	587	741	650	539	730	663
scp501	253	1178	827	973	811	833	1122	905	976	1114	1055	808	523	481	410	524	631	564	428	567	487
scp61	138	545	451	554	442	643	537	605	554	435	510	260	248	275	224	295	287	272	221	228	217
scpa1	255	1778	1449	1731	1672	1658	1983	1208	1741	1948	1736	893	718	891	986	833	1068	603	1001	1263	826
scpb1	69	1396	1081	1069	953	1502	1287	1040	1243	938	1434	528	316	408	261	443	392	359	389	340	548
scpc1	227	1722	2136	2049	2172	2097	2341	1927	1449	2117	2180	817	1144	1150	1258	1157	1288	1149	933	1245	1245
scpd1	60	1792	1070	1156	1315	1280	1227	1598	1480	1524	1327	408	285	312	367	327	347	355	285	396	443
scpnre1	29	1391	1256	1189	926	1179	1336	1109	1273	1507	1191	96	93	62	67	65	83	75	68	81	63
scpnrf1	14	912	676	648	793	842	808	847	610	669	647	14	15	14	14	14	14	14	14	14	14
scpnrg1	176	3993	3937	3920	4026	4307	4801	3738	4000	4266	3663	1972	1937	1841	1907	2218	2091	1703	1727	1864	1684

Tabelle 79: Testlauf (Mut: 0,75%) – Komma-Strategie, elitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 126,5 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	875	726	806	451	1006	822	773	670	891	781	634	487	449	451	704	553	490	472	606	526
scp501	253	1042	674	720	937	757	801	844	620	1027	744	499	436	501	407	358	475	460	349	455	529
scp61	138	379	398	416	399	262	294	389	311	381	344	184	180	189	293	189	178	257	187	229	176
scpa1	255	878	1400	1165	1557	1406	1028	1396	1229	803	1445	632	712	723	922	868	716	634	729	509	830
scpb1	69	948	972	563	1131	907	906	906	754	804	986	310	334	250	324	292	289	260	299	189	280
scpc1	227	2293	1490	1484	1757	1651	2040	1554	1944	1519	1662	1282	882	752	1038	1063	1244	1005	1070	963	903
scpd1	60	1427	1547	1323	1247	1631	1084	1150	1386	1007	1251	434	525	500	326	590	495	313	542	311	404
scpnre1	29	877	971	1058	823	1021	1045	1245	802	1082	912	112	134	128	112	125	162	111	119	145	132
scpnrf1	14	547	698	536	523	535	639	608	483	558	821	18	19	17	20	16	19	18	15	20	17
scpnrg1	176	3477	4556	4376	3546	3805	4019	3258	3254	3131	3930	2097	2526	2150	1872	2055	2034	1762	1651	1662	2116

Tabelle 80: Testlauf (Mut: 0,5%) – Komma-Strategie, elitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 127,2 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	631	918	507	571	643	767	516	448	669	702	486	750	507	492	512	531	442	448	489	514
scp501	253	562	691	822	560	463	745	635	633	358	749	366	406	423	336	354	434	485	424	290	475
scp61	138	393	215	261	244	186	358	336	327	387	440	227	215	185	244	186	221	183	224	245	283
scpa1	255	745	1108	1045	865	771	947	1330	742	941	1007	511	661	631	512	402	554	848	451	484	664
scpb1	69	501	675	382	559	523	419	608	484	518	434	206	269	250	249	290	182	297	187	231	174
scpc1	227	942	1143	1162	899	953	946	1075	839	1275	1238	667	762	770	674	672	653	858	536	774	815
scpd1	60	674	639	806	602	545	896	1024	724	552	572	244	236	348	209	211	320	350	264	221	210
scpnre1	29	868	684	780	755	674	427	718	470	575	502	243	189	156	114	180	141	121	157	235	149
scpnrf1	14	480	344	380	382	526	360	376	475	483	437	18	36	28	29	24	26	29	37	23	34
scpnrg1	176	3278	3226	2368	2753	3134	3157	3075	3019	2852	4220	2090	2030	1676	1753	2053	2034	1923	1952	1733	2607

Tabelle 81: Testlauf (Mut: 0,25%) – Komma-Strategie, elitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 104,4 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	474	438	457	455	463	510	482	530	485	503	474	438	457	455	463	510	482	530	485	503
scp501	253	341	403	356	381	424	462	508	319	348	451	341	401	356	370	424	450	426	316	348	406
scp61	138	164	207	163	230	182	296	214	232	335	223	164	205	136	230	180	291	214	225	206	220
scpa1	255	563	410	577	508	809	848	743	702	738	669	391	370	435	404	641	554	446	462	575	518
scpb1	69	338	446	393	151	239	257	340	495	186	242	199	278	160	134	158	198	226	278	121	154
scpc1	227	633	479	723	646	815	755	1293	789	739	675	588	448	554	583	574	644	879	582	706	632
scpd1	60	363	356	422	382	698	448	421	353	353	479	171	152	294	162	482	227	205	183	176	322
scpnre1	29	265	223	418	298	265	392	204	361	332	297	87	83	138	81	99	82	87	111	103	109
scpnrf1	14	189	296	189	236	184	140	213	199	117	110	34	62	34	49	34	22	19	30	20	23
scpnrg1	176	1994	1873	1444	1694	2110	1354	1858	1376	1660	2485	1440	1110	1042	1305	1489	1115	1388	1008	1219	1864

Tabelle 82: Testlauf (Mut: 0,1%) – Komma-Strategie, elitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 103,4 h

A.4 Ergebnisse der Tests mit konstanter Mutationsrate

Die im Folgenden dargestellten Tests benutzten bis auf die Mutation die gleichen Parameter und Operatoren wie die entsprechenden in Abschnitt A.1 beschriebenen Tests (Tabellen 43 und 47). Die Mutationsrate wurde aber auf eine konstante Anzahl von zu flippenden Bits gesetzt. Die Werte waren: 16, 12, 8 und 4 zu flippende Bits. Diese Werte wurden jeweils für einen Test mit und einen ohne Baldwin-Lernen benutzt.

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	475	437	462	476	446	436	482	483	456	447	-	-	-	-	-	-	-	-	-	-
scp501	253	329	387	359	362	286	362	376	353	332	375	-	-	-	-	-	-	-	-	-	-
scp61	138	160	175	191	221	211	172	188	176	167	186	-	-	-	-	-	-	-	-	-	-
scpa1	255	418	346	452	413	386	440	336	470	373	304	-	-	-	-	-	-	-	-	-	-
scpb1	69	127	99	91	124	158	136	209	161	143	112	-	-	-	-	-	-	-	-	-	-
scpc1	227	502	380	556	372	462	507	459	639	451	395	-	-	-	-	-	-	-	-	-	-
scpd1	60	109	138	121	146	121	139	180	134	125	156	-	-	-	-	-	-	-	-	-	-
scpnre1	29	60	68	75	50	64	149	90	54	70	93	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	16	28	31	32	19	19	52	25	27	21	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	860	573	788	735	870	709	627	712	632	651	-	-	-	-	-	-	-	-	-	-

Tabelle 83: Testlauf (Mut: 4 Bits) – Komma-Strategie, elitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 41,6 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	508	441	523	447	493	481	505	450	462	474	-	-	-	-	-	-	-	-	-	-
scp501	253	272	342	326	325	323	299	384	331	274	369	-	-	-	-	-	-	-	-	-	-
scp61	138	195	207	198	186	180	164	160	177	196	181	-	-	-	-	-	-	-	-	-	-
scpa1	255	435	356	325	467	421	334	490	379	460	351	-	-	-	-	-	-	-	-	-	-
scpb1	69	139	97	127	150	101	99	105	114	188	113	-	-	-	-	-	-	-	-	-	-
scpc1	227	481	520	492	430	447	352	376	558	453	430	-	-	-	-	-	-	-	-	-	-
scpd1	60	136	102	123	112	151	173	105	108	165	242	-	-	-	-	-	-	-	-	-	-
scpnre1	29	64	43	43	69	54	142	54	51	62	71	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	24	25	22	23	39	21	26	24	29	27	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	737	757	516	618	712	829	730	649	667	915	-	-	-	-	-	-	-	-	-	-

Tabelle 84: Testlauf (Mut: 8 Bits) – Komma-Strategie, elitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 48,3 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	454	478	546	438	492	478	450	431	431	480	-	-	-	-	-	-	-	-	-	-
scp501	253	323	331	298	276	340	317	270	326	334	416	-	-	-	-	-	-	-	-	-	-
scp61	138	166	174	180	156	180	204	184	184	198	144	-	-	-	-	-	-	-	-	-	-
scpa1	255	368	422	308	405	361	381	488	405	408	319	-	-	-	-	-	-	-	-	-	-
scpb1	69	113	104	110	140	91	116	103	133	150	140	-	-	-	-	-	-	-	-	-	-
scpc1	227	375	515	569	396	414	370	396	393	396	352	-	-	-	-	-	-	-	-	-	-
scpd1	60	144	104	119	118	112	111	145	120	113	98	-	-	-	-	-	-	-	-	-	-
scpnre1	29	52	76	53	55	47	88	50	49	50	53	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	33	33	19	34	31	38	35	34	19	19	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	620	520	590	643	544	462	483	716	451	482	-	-	-	-	-	-	-	-	-	-

Tabelle 85: Testlauf (Mut: 12 Bits) – Komma-Strategie, elitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 48,3 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	557	470	465	497	504	529	457	439	566	471	-	-	-	-	-	-	-	-	-	-
scp501	253	311	371	335	327	323	365	326	341	340	370	-	-	-	-	-	-	-	-	-	-
scp61	138	252	148	165	187	175	167	170	189	204	146	-	-	-	-	-	-	-	-	-	-
scpa1	255	387	352	338	314	431	362	358	409	540	433	-	-	-	-	-	-	-	-	-	-
scpb1	69	117	162	142	208	123	126	135	104	106	122	-	-	-	-	-	-	-	-	-	-
scpc1	227	354	381	499	578	376	580	397	409	399	450	-	-	-	-	-	-	-	-	-	-
scpd1	60	131	153	88	231	150	126	125	164	111	160	-	-	-	-	-	-	-	-	-	-
scpnre1	29	59	100	54	67	57	58	52	76	105	110	-	-	-	-	-	-	-	-	-	-
scpnrf1	14	58	26	35	47	40	47	34	36	38	43	-	-	-	-	-	-	-	-	-	-
scpnrg1	176	534	675	477	589	763	559	517	476	502	725	-	-	-	-	-	-	-	-	-	-

Tabelle 86: Testlauf (Mut: 16 Bits) – Komma-Strategie, elitär, ohne Baldwin, rand. Startpop., Multicost; Prozessorzeit – 48,4 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	717	793	711	765	773	730	768	834	717	697	574	617	540	492	478	526	526	588	482	505
scp501	253	535	481	594	705	601	570	616	370	679	606	346	356	316	457	426	430	428	370	489	428
scp61	138	311	472	231	473	485	370	269	363	478	279	235	279	184	244	276	183	206	231	192	253
scpa1	255	756	497	882	822	908	845	737	472	798	722	456	372	703	574	689	618	491	472	585	537
scpb1	69	263	226	308	221	173	225	221	305	436	153	173	134	254	130	150	150	201	213	257	150
scpc1	227	756	952	683	652	617	622	885	506	1028	705	719	640	464	604	515	586	698	409	836	500
scpd1	60	482	252	322	396	604	621	284	382	381	389	208	159	161	209	339	338	187	228	230	244
scpnre1	29	147	172	145	72	269	236	251	190	182	58	77	44	42	43	78	84	97	72	94	44
scpnrf1	14	68	25	126	99	154	107	67	189	105	269	19	18	27	25	18	28	24	33	23	24
scpnrg1	176	1097	819	1484	1282	861	1653	997	1496	894	972	988	786	1089	1091	833	1440	900	1289	873	842

Tabelle 87: Testlauf (Mut: 4 Bits) – Komma-Strategie, elitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 115,6 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	1324	782	996	953	814	893	800	864	906	944	776	542	586	644	560	617	532	577	619	642
scp501	253	1032	759	355	764	653	794	648	764	746	720	474	344	355	527	471	458	387	425	432	459
scp61	138	478	416	764	597	461	427	548	437	370	432	224	194	316	263	232	207	217	189	176	237
scpa1	255	924	853	609	1150	728	748	1101	779	945	562	615	651	490	676	422	459	697	571	494	432
scpb1	69	759	389	853	583	624	588	405	532	915	775	231	203	392	169	299	296	133	217	414	337
scpc1	227	1028	1139	701	819	1123	1905	1190	1211	1159	926	654	916	583	642	710	1306	780	915	716	556
scpd1	60	484	632	654	704	534	716	814	633	383	517	194	351	228	316	184	304	437	327	218	255
scpnre1	29	205	122	121	265	79	134	94	126	148	220	112	62	62	75	65	69	78	55	90	63
scpnrf1	14	132	126	101	50	87	89	106	103	30	107	40	52	18	28	17	34	27	23	17	22
scpnrg1	176	987	913	1305	1284	929	948	716	956	918	1108	925	794	1244	1029	826	697	625	801	775	887

Tabelle 88: Testlauf (Mut: 8 Bits) – Komma-Strategie, elitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 112,6 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	1103	1163	1296	1197	1286	1109	907	1291	1073	1085	681	663	851	797	768	656	598	738	714	669
scp501	253	689	904	1194	871	1099	1000	934	834	790	680	442	462	589	454	524	459	465	432	402	394
scp61	138	591	573	540	495	399	647	732	598	562	480	226	186	221	173	187	277	306	241	179	215
scpa1	255	908	1050	1374	1135	1496	1356	1284	1283	1309	976	557	626	804	643	912	955	732	960	703	478
scpb1	69	851	378	786	639	963	829	862	713	791	1071	213	202	284	206	293	233	360	319	271	297
scpc1	227	1151	1206	1343	1084	1181	1129	1818	1124	1029	893	827	710	813	674	794	822	1103	673	681	512
scpd1	60	629	883	802	820	705	717	571	764	770	1014	218	317	261	249	280	315	326	211	287	360
scpnre1	29	763	544	755	429	655	404	665	577	656	482	165	131	221	84	160	107	233	140	126	122
scpnrf1	14	359	425	350	360	425	240	365	314	352	509	29	23	29	26	27	38	33	20	24	34
scpnrg1	176	1907	1786	1756	1809	1293	1145	1945	1440	1771	1318	1237	1287	1138	1211	912	839	1328	1035	1259	819

Tabelle 89: Testlauf (Mut: 12 Bits) – Komma-Strategie, elitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 107,2 h

Name	Ref.	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L1B	L2B	L3B	L4B	L5B	L6B	L7B	L8B	L9B	L10B
scp401	430	1328	1439	1490	1453	1293	1353	1349	947	1603	1168	850	849	893	826	743	815	681	611	841	687
scp501	253	817	819	758	906	848	775	700	946	874	1012	467	423	421	442	516	447	452	431	506	544
scp61	138	617	631	733	866	422	677	886	681	532	659	199	250	178	258	240	279	422	217	317	298
scpa1	255	1328	1281	1753	1260	1510	1498	1509	1164	1734	1343	789	793	1135	597	883	942	838	664	1085	743
scpb1	69	615	913	737	785	849	709	881	1110	829	780	190	266	300	267	248	258	235	323	391	273
scpc1	227	1354	1309	1473	865	1469	1873	1600	1401	1669	1457	1000	698	948	642	790	1084	889	790	903	787
scpd1	60	855	798	1047	1200	795	940	1170	972	939	1118	343	307	366	348	322	303	353	436	455	406
scpnre1	29	711	880	493	771	526	769	705	954	1013	713	126	122	120	91	112	110	107	139	145	140
scpnrf1	14	580	542	488	569	548	536	385	598	513	652	21	21	21	27	23	23	22	19	30	31
scpnrg1	176	2569	2979	3013	2363	1575	1996	2587	3104	2183	2725	1696	2214	1885	1667	1193	1412	1644	2245	1568	1840

Tabelle 90: Testlauf (Mut: 16 Bits) – Komma-Strategie, elitär, Baldwin, rand. Startpop., Multicost; Prozessorzeit – 108,7 h

B Auflistung der implementierten Pakete und Klassen

Es folgt eine Auflistung aller Klassen, Interfaces und Pakete, die für diese Studienarbeit implementiert wurden. Klassen und Interfaces beginnen mit einem Großbuchstaben, Pakete mit einem Kleinbuchstaben. Die Auflistung entspricht dem Stand zum Zeitpunkt des Einfrierens des Programms vor den Testläufen.

- evolutionaererAlgorithmus
  - BewertungsVergleich
  - EvolutionaereUeberdeckung
  - Genom
  - Individuum
  - Konstanten
  - OperatorenAuswahl
  - OpUndParErzeugung
  - ParameterAuswahl
  - Population
  - Statistik
  - TeilmengenZuordnung
- operatoren
  - bewertung
    - BaldwinVerfahren (Interface)
    - BewertungsVerfahren (Interface)
    - BewKostenLinMitBaldwin
  - diversitaetsmessung
    - DiversMessVerfahren (Interface)
    - DiversitaetStochastisch
  - gueltigkeit
    - GueltigkeitsVerfahren (Interface)
    - GueltigkeitLinear
    - GueltigkeitRandomisiert
  - mutation
    - MutationsVerfahren (Interface)
    - MutationFlip
    - MutationFlipAnzahlBits
    - MutationSwap
  - rekombination
    - einzelRek
      - EinzelRekVerfahren (Interface)

            - WkeitZuordnen (Interface)
            - RekEinzelnZweiZuEins
            - WahrschEinfach
            - WahrschFitnessAbh
        * RekombinationsVerfahren (Interface)
        * RekombinationEinfach
        * RekombinationFitnessabh
    - selektion
        * SelektionsVerfahren (Interface)
        * SelektionRangLinear
        * SelektionTurnier
    - sperren
        * SperrVerfahren (Interface)
        * SperrungBesteInd
    - startpopulation
        * StartpopErzeugVerfahren (Interface)
        * StartpopAusListe
        * StartpopulationBeasley
        * StartpopulationRand
    - teilmengenSortierung
        * TeilmengenVergleich
    - terminierung
        * TerminierungsVerfahren (Interface)
        * TermMaxGen
        * TermMaxGenErweitert

- graphVis
    - darstellungsModi
        * DarstellungsKit (Interface)
        * DarstellungsModus1
        * DarstellungsModus2
        * DarstellungsModus3
        * Konstanten
    - zeichenModi
        * ZeichenKit (Interface)
        * ZeichenModus1
        * AusgabeMerkmale
        * KreisKlasse
        * Konstanten
    - graph
        * Graph

        * Knoten
    - DargestellterGraph
    - Grapherzeugung
    - Konstanten
    - OpUndParErzeugungBaldwin
    - OpUndParErzeugungNormal
    - Vis
- test
    - GrapherzeugungZufall
    - TestGraph
    - ResultStatistikCreation
    - StatistikAusgeben
    - TestEvolUeberdeckBib
    - TestEvolUeberdeckZufall
    - TestItGreedyBib
    - Konstanten

Literatur

[Ackley1992] Ackley, D. H.; Littman, M. L.: Interactions between learning and evolution. In: C. G. Langton, C. Taylor, J. D. Farmer und S. Rasmussen (Hrsg.): *Artificial Life II*, 487-507, Reading, MA: Addison-Wesley 1992

[Ackley1993] Ackley, D. H.; Littman, M. L.: A case for Lamarckian evolution. In: C. G. Langton (Hrsg.): *Artificial Life III* (Proceedings Volume XVIII, Santa Fe Institute Studies, Sciences of Complexity), 487-509, Reading, MA: Addison-Wesley 1993

[Aickelin2000] Aickelin, U.: A New Genetic Algorithm for Set Covering Problems. *Annual Operational Research Conference 42*, (2000)

[Baldwin1896] Baldwin, J. M.: A new Factor in Evolution, *The American Naturalist 30*, 441-451 (1896); Neudruck in: *Adaptive Individual in Evolving Populations: Models and Algorithms.* In: R.K. Belew und M. Mitchel (Hrsg.), 59-80, Reading, MA: Addison Wesley 1996

[Batista1994] Battista, G.; Tamassia, R.; Eades, P.; Tollis, I. G.: Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry Theory & Applications 4*, 235-282 (1994)

[Beasley1994] Beasley, J. E.; Chu, P. C.: A Genetic Algorithm for the Set Covering Problem. *European Journal of Operation Research*, 394-404, (1994)

[Bortz1999] Bortz, J.: *Statistik*. Berlin, Heidelberg, New York: Springer-Verlag 1999

[Eremeev1999] Eremeev, A.: A genetic algorithm with a non-binary representation for the set covering problem. In: *Proc. of Operations Research (OR'98)*, 175-181, Berlin, Heidelberg, New York: Springer-Verlag 1999

[Grefenstette1991] Grefenstette, J. J.: Lamarckian learning in multi-agent environments. In R. K. Belew und L. B. Booker (Hrsg.): *Proceedings of the Fourth International Conference on Genetic Algorithms*, 303-310, San Mateo, CA: Morgan Kaufmann 1991

[Gruau1993] Gruau, F.; Whitley, D.: Adding learning to the cellular development of neural networks: Evolution and the Baldwin effect. *Evolutionary Computation 1.3*, 213-233 (1993)

[Hinton1987] Hinton, G. E.; Nowlan, S. J.: How learning can guide evolution. *Complex Systems 1*, 495-502 (1987)

[IPVS] Homepage des Instituts für Parallele und Verteilte Systeme der Uni Stuttgart: http://www.ipvs.uni-stuttgart.de/start (Stand: 11.12.2006)

[Karp1972] Karp, R. M.: Reducibility Among Combinatorial Problems. In R.E. Miller und J.W. Thatcher (Hrsg.): *Complexity of Computer Computations*, 85-103, New York: Plenum 1972

[Lamarck1809] Lamarck, J. B.: *Zoologische Philosophie*. Leipzig: Akademische Verlagsgesellschaft Geest & Portig K.-G. 1990

[Lippold2006] Lippold, D.: *Framework zum Set Covering Problem*, Universität Stuttgart. Version vom 06.11.2006

[Lipton1985] Lipton, R. J.; North, S. C.; Sandberg, J. S.: A method for drawing graphs; *SCG '85: Proceedings of the first annual symposium on Computational geometry*, 153-160, Baltimore, Maryland, United States: ACM Press 1985

[Marchiori1998] Marchiori, E.; Steenbeek, A.: An iterated heuristic for the set covering problem. In K. Mehlhorn (Hrsg.): *Proceedings of the Workshop on Algorithm Engineering*, 155-166 (1998)

[Marchiori2000] Marchiori, E.; Steenbeek, A.: An Evolutionary Algorithm for Large Scale Set Covering Problems with Application to Airline Crew Scheduling. *EvoWorkshops*, 367-381 (2000)

[Mayley1996] Mayley, G.: The Evolutionary Cost of Learning. In P. Maes und M. J. Mataric und J. A. Meyer und J. Pollack und S. W. Wilson (Hrsg.): *From Animals to Animats 4: Proc. of the Fourth Int. Conf. on Simulation of Adaptive Behavior*, Cambridge, MA: The MIT Press 1996

[Nissen1997] Nissen, V.: *Einführung in Evolutionäre Algorithmen.* Braunschweig, Wiesbaden: Friedr. Vieweg & Sohn Verlagsgesellschaft mbH 1997

[Weicker2002] Weicker, K.: *Evolutionäre Algorithmen.* Stuttgart, Leipzig, Wiesbaden: Teubner Verlag 2002

[Whitley1994] Whitley, D. L.; Gordon, V. S.; Mathias, K. E.: Lamarckian evolution, the Baldwin Effect and function optimization. In Y. Davidor und H. P. Schwefel und R. Männer (Hrsg.): *Parallel Problem Solving from Nature - PPSN III*, 6-15, Berlin: Springer 1994

Versicherung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

Lukas König