Graphbasierte Genetische Programmierung

Dissertation

zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik
von

Jens Niehaus

Dortmund,

Juli 2003

Tag der mündlichen Prüfung:
22. Dezember 2003

Dekan:
Prof. Dr. Bernhard Steffen

Gutachter:
Prof. Dr. Wolfgang Banzhaf
Prof. Dr. Peter Marwedel

Diese Arbeit wäre ohne die Hilfe vieler Menschen nicht möglich gewesen. Mein Dank gilt Prof. Dr. Wolfgang Banzhaf, in dessen Forschungsgruppen am Informatik Centrum Dortmund und am Lehrstuhl für Systemanalyse des Fachbereichs Informatik an der Universität Dortmund ich sowohl sehr gute Arbeitsbedingungen nutzen konnte als auch viele, die Arbeit voranbringende Diskussionen führte. Besonders hervorzuheben sind hier die Gespräche mit Jens Busch, Markus Conrads, Wolfgang Kantschik und Dr. Jens Ziegler.

Für weitere interessante Diskussionen und Tipps danke ich Dr. Christian Igel, Sabine Josten und Dirk Thamer.

Besonderer Dank gebührt schließlich meinen Eltern und – vor allem – Birgit Köck, ohne die diese Arbeit nicht möglich gewesen wäre.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Viele Optimierungsprobleme lassen sich mit Methoden der klassischen Optimierungstheorie nur schwer lösen. Dies kann vielfältige Ursachen haben: Zum Beispiel kann der Suchraum bei Extremwertberechnungen nicht-differenzierbar sein oder es liegt eine multimodale Problemstellung vor. Auch kann die Komplexität der Aufgabenstellung so hoch sein, dass noch kein deterministisches Verfahren gefunden werden konnte.

In solchen Fällen ist der Einsatz von Heuristiken gerechtfertigt. Eine Klasse solcher heuristischer Algorithmen sind die *Evolutionären Algorithmen*, die darauf beruhen, dass mit aus der Evolutionstheorie bekannten Prinzipien wie Vererbung und Selektion probablistisch Lösungen gesucht werden. Potenzielle Problemlösungen werden als Individuen einer Population codiert. Diese Individuen werden über viele Iterationen verändert und mit anderen kombiniert, bis ein Individuum entsteht, das eine ausreichend gute Annäherung an die Lösung des Problems repräsentiert. Jedem Individuum wird eine *Fitness* zugeordnet, die sich daraus ergibt, wie nahe es der gesuchten Lösung kommt. Über diese Fitness wird ein Individuum mit anderen vergleichbar.

Evolutionäre Algorithmen unterteilen sich wiederum in mehrere Varianten, die sich unter anderem durch die Art des Suchraumes voneinander unterscheiden. *Evolutionsstrategien* suchen Lösungen zum Beispiel als reellwertige Vektoren und sind somit hauptsächlich zur Parameteroptimierung geeignet. Sollen Probleme gelöst werden, deren Suchraum nicht aus Vektoren besteht, muss eine Codierung vom Raum der reellwertigen Vektoren in den durch die Problemstellung vorgegebenen Suchraum der potenziellen Lösungen gefunden werden. Diese Kodierung wird, in Anlehnung an die Biologie, *Genotyp-Phänotyp*-Abbildung genannt.

Die *Genetische Programmierung (GP)* als weitere Variante der *Evolutionären Algorithmen* verwendet vornehmlich Bäume als Datenstruktur des Suchraums. Während zum Beispiel *Evolutionsstrategien* bevorzugt bei hochdimensionalen Extremwertberechnungen eingesetzt werden können, ist der Suchraum bei der *Genetischen Programmierung* eher zur Approximation von Funktionen geeignet, da sich diese als funktionale Ausdrücke in einer Baumstruktur repräsentieren lassen und somit *Genotyp* und *Phänotyp* identisch sein können.

Ein wichtiger Gesichtspunkt *Evolutionärer Algorithmen* ist das Prinzip der *starken Kausalität*, das als eine der Voraussetzungen der *Evolutionsstrategien* anzusehen ist. Es bedeutet, dass kleine Veränderungen eines Individuums im Durchschnitt auch kleine Veränderungen seiner Fitness zur Folge haben. Punkte, die im Suchraum nahe beieinander liegen, repräsentieren also ähnlich gute Lösungen. Wenn bei der *Genetischen Programmierung* eine Lösung im Phänotyp einem Graphen entspricht, dieser im Genotyp jedoch durch einen Baum kodiert wird, muss die starke Kausalität über zwei Stufen gelten: Zum einen muss eine kleine Veränderung des Genotyp-Baumes durchschnittlich eine ebenfalls kleine Veränderung des Phänotyp-Graphen zur Folge haben und zum anderen darf dies im Durchschnitt auch nur eine kleine Auswirkung auf die Fitness haben. Würden im Durchschnitt kleine Änderungen der Baumstrukturen zu großen Veränderungen der Graphen des Phänotyps führen, müsste durch die Fitnessfunktion sichergestellt werden können, dass sich die Fitness dieser Graphen trotzdem nur geringfügig ändert.

Zur Zeit ist die theoretische Untersuchung der starken Kausalität in der *Genetischen Programmierung* noch nicht sehr weit fortgeschritten. Das Vorhandensein stark kausaler Zusammenhänge zwischen Änderungen des Phänotyps und den zugehörigen Fitnessveränderungen ist sehr wahrscheinlich stark problemabhängig und kann nicht als gegeben vorausgesetzt werden. Betrachtet man stark kausale Zusammenhänge jedoch vom Genotyp bis zur Fitness, sind sie am ehesten gegeben, wenn Genotyp und Phänotyp identisch sind. Aus diesem Grund wird mit dieser Arbeit der Ansatz der *Genetischen Programmierung* so erweitert, dass Graphen als *Genotyp* für Probleme verwendet werden können, deren Lösung ebenfalls durch einen Graph repräsentiert wird. Problemstellungen, bei denen Graphen zur Repräsentation von Lösungen nahe liegen, sind beispielsweise Schaltungs- und Reglerentwurf oder der Entwurf von Algorithmen, wobei ein Algorithmus durch ein entsprechendes Flussdiagramm repräsentiert wird.

Ausgangspunkt für diese Arbeit war das mit Mitteln des Bundesministeriums für Bildung und Forschung geförderte Projekt *Genetisches Programmieren für Modellierung und Regelung dynamischer Systeme* [69]. Ziel des Projekts war es, in Kooperation mit *DaimlerChrysler* und der *Technischen Universität Berlin* ein hybrides System zu entwerfen, das die Strukturen der gesuchten Regler und Modelle mittels *Genetischer Programmierung* und die Parameter einzelner Regelelemente mit Hilfe von *Evolutionsstrategien* ermittelt. Das Fachwissen über *Evolutionsstrategien* wurde von Prof. Dr. Rechenbergs Arbeitsgruppe *Bionik & Evolutionstechnik* der *TU Berlin* in das Projekt eingebracht, die Testprobleme und das Fachwissen aus der Regelungstechnik stellte *DaimlerChrysler* zur Verfügung. Die zur empirischen Bearbeitung der Testprobleme notwendige Hard- und Software befand sich bei den Projektpartnern. Innerhalb des Projektes wurde das Programm *SCADS* erstellt, dessen *GP*-Komponente Graphen als Datenstruktur für den Genotyp verwendet. Das Programm ist in der Lage, die gestellten Anforderungen zu erfüllen, kann jedoch nur für relativ einfache Probleme Lösungen finden (Anhang A).

Nach der Beendigung des Projektes gab es zwei Alternativen, die Arbeit fortzusetzen. Zum einen gab es die Möglichkeit, *SCADS* durch weitere Integration problemspezifischen Regelungstechnikwissens als Spezialverfahren zur Regelung und Modellierung dynamischer Systeme zu verbessern. Allerdings war die für ein sinnvolles empirisches Arbeiten benötigte Hard- und Software nicht vorhanden.

Die zweite Möglichkeit bestand in der Generalisierung der GP-Komponente von *SCADS* zu einem allgemeinen Suchalgorithmus. Ziel war hierbei, eine Variante der Genetischen Programmierung zu erarbeiten, die ebenso allgemein einsetzbar sein sollte wie auf Baumstrukturen arbeitende GP-Systeme. Durch die Verwendung von Graphen im Genotyp sollte die starke Kausalität für Probleme, die im Phänotyp ebenfalls Graphen nahelegen, eher erfüllt sein als für herkömmliche baumbasierte GP-Systeme. Wenn also die Vermutung stimmt, dass starke Kausalität auch in der Genetischen Programmierung einen Einfluss auf die Güte der Ergebnisse hat, sollten sie mit graphbasierten Systemen bei entsprechenden Problemstellungen besser werden.

Wegen der fehlenden Voraussetzungen für eine gezielte Spezialisierung von *SCADS* wurde daraufhin das neue GP-System *GGP* entworfen, das Graphen als Datenstruktur des Genotyps verwendet. Anhand von im GP-Bereich allgemein akzeptierten Benchmarks wird die Performanz des neuen Systems durch empirische Versuchsreihen mit der eines GP-Systems verglichen, das den bisher üblichen Ansatz mit Baumstrukturen als Genotyp verwendet.

Die Ergebnisse der Systemvergleiche waren Grundlage für weitere Verbesserungen des neuen GP-Systems. Es wurde beim Hinzufügen neuer Eigenschaften darauf Wert gelegt, dass kein problemspezifisches Wissen in das GP-System einfließt. Auf diese Weise wurde sichergestellt, dass das System *GGP* weiterhin ein allgemeines Suchverfahren darstellt und der Vergleich mit dem ebenfalls allgemeinen Ansatz des baumbasierten GP-Systems legitim bleibt.

Das hier beschriebene Vorgehen spiegelt sich in der Gliederung der Arbeit wider: Nach einer kurzen Einführung zu *Evolutionären Algorithmen* im Allgemeinen und *Genetischer Programmierung* mit Graphstrukturen im Besonderen wird in Kapitel 3 beschrieben, wie ein Individuum einer Population im GP-System *GGP* durch einen Graphen repräsentiert wird.

In Kapitel 4 folgt die Vorstellung des GP-Systems *GGP*. Neben der prinzipiellen Arbeitsweise des Systems wird besonders auf das Erzeugen initialer Graphen und die Funktionen eingegangen, mit denen diese Graphen verändert werden können. Diese Funktionalitäten unterscheiden sich zum Teil sehr stark von den Methoden, die bei baumbasierten Systemen verwendet werden.

Kapitel 5 gibt eine Übersicht über die verwendeten Testprobleme. Sie unterteilen sich in zwei Gruppen: Für die erste Gruppe bietet sich eine Repräsentation der Lösungen als Graph an. Anhand dieser Probleme kann untersucht werden, ob das neue GP-System tatsächlich bessere Ergebnisse erzielen kann als herkömmliche baumbasierte Systeme. Die zweite Gruppe besteht aus Problemen, für die Bäume zur Repräsentation des Phänotyps ausreichen. Bei diesen entfällt somit auch bei baumbasierten GP-Systemen die Genotyp-Phänotyp-Codierung. Diese Probleme eignen sich zur Untersuchung der Fragestellung, ob der bei graphbasierten Systemen vergrößerte Suchraum zu einer Verlangsamung der Suche führt oder sich die Suche durch die zusätzlichen Punkte des Suchraums effizienter gestalten lässt. Zu jedem Testproblem werden die Ergebnisse empirischer Vergleichsreihen eines traditionellen baumbasierten GP-Systems und des neuen graphbasierten Systems direkt aufgeführt.

Die Kapitel 6 bis 10 beschreiben Verbesserungen des GP-Systems. Die Einführung aller Neuerungen wird ausführlich erläutert und die Funktionsweise neuer Teilalgorithmen im Detail dargestellt. Schliesslich werden alle Neuerungen mittels empirischer Tests mit den in Kapitel 5 vorgestellten Testproblemen auf ihren Nutzen hin untersucht.

In Kapitel 11 wird aufgezeigt, wie das GP-System *GGP* zum Entwurf paralleler Algorithmen eingesetzt werden kann. Dies ergibt sich daraus, dass sich Algorithmen als Flussdiagramme darstellen lassen und sich somit die Verwendung eines GP-Systems mit einer Lösungsrepräsentation als Graph anbietet. Ein Flussdiagramm kann nun so modifiziert werden, dass zwei Programmflüsse innerhalb desselben Diagramms modelliert werden.

Das Schlusskapitel 12 fasst noch einmal die Ergebnisse aller durchgeführten empirischen Untersuchungen zusammen und zeigt auf, wie das GP-System in der Praxis eingesetzt werden kann und an welchen Stellen es weitere Möglichkeiten zur Verbesserung gibt.

Kapitel 2

Grundlagen der Genetischen Programmierung

Dieses Kapitel gibt einen kurzen Überblick über die Grundlagen der *Genetischen Programmierung (GP)*, auf denen diese Arbeit aufbaut. Nach einer einführenden Beschreibung *Evolutionärer Algorithmen* in Abschnitt 2.1 folgt eine Betrachtung der Funktionsweise von *GP*-Systemen. Abschnitt 2.3 schließlich stellt den Zusammenhang zwischen Graphen und *GP* her: Es wird ein kurzer Überblick über verschiedene Ansätze gegeben, Graphen als Datenstruktur in der *Genetischen Programmierung* zu verwenden.

2.1 Evolutionäre Algorithmen

Evolutionäre Algorithmen sind stochastische Verfahren zur Lösung komplexer Probleme. Sie simulieren hierzu klassische Prinzipien der natürlichen Evolution nach DARWIN mit einfachen Modellen. Einige ihrer Einsatzgebiete sind Parameter- und Strukturoptimierung, Maschinelles Lernen und Klassifizierung. Potenzielle Lösungen eines Problems werden als Instanzen einer geeigneten Datenstruktur repräsentiert. Während des Ablaufs eines Evolutionären Algorithmus werden zu jedem Zeitpunkt mehrere potenzielle Lösungen vorgehalten.[1] In Anlehnung an die Biologie werden diese *Individuen einer Population* genannt. Die Instanz der Datenstruktur eines Individuums wird als dessen *Genom* bezeichnet. Jedem dieser Individuen wird eine *Fitness* zugeordnet. Dies ist in den meisten Fällen eine reellwertige Zahl oder ein entsprechender Vektor. Der Fitnesswert sagt aus, wie genau eine potenzielle Lösung dem gesuchten Optimum entspricht. Auf diese Weise können verschiedene Individuen miteinander verglichen werden und es wird ein Evolutionsdruck simuliert.

Der Verlauf eines *Evolutionären Algorithmus* zerfällt in drei Abschnitte: Zunächst wird eine Startpopulation erzeugt. Diese besteht aus einer vorher festgelegten Anzahl von zufällig

[1] Bei einigen speziellen Verfahren kann diese Zahl auch gleich eins sein, z.B. bei einer $(1, \lambda)$-Evolutionsstrategie [93].

erzeugten Individuen. Im zweiten Abschnitt unterliegt die Population einem Evolutionszyklus. In diesem werden neue Individuen aus Teilen alter Individuen rekombiniert. Leicht veränderte ältere Individuen können als neue Individuen in die Population aufgenommen werden und alte Individuen werden aus der Population entfernt. Die Abarbeitung des Algorithmus terminiert schließlich aufgrund eines *Abbruchkriteriums*. Hierfür kommen mehrere Bedingungen in Betracht: Das Optimum wurde gefunden, ein vorgegebener Zeitrahmen wurde ausgeschöpft oder die beste Fitness aller Individuen stagnierte über einen zu langen Zeitraum.

Verschiedene Varianten der *Evolutionären Algorithmen* unterscheiden sich durch die gewählte Datenstruktur zur Repräsentation der Lösungen sowie durch die Realisierungen des Evolutionszyklus. An dieser Stelle werden nur allgemeine Bemerkungen zu den einzelnen Punkten gemacht, die für alle Verfahren gelten.

Startpopulationen Aufgabe eines *Evolutionären Algorithmus* ist es, das Individuum mit der besten Fitness in einem vorgegebenen Suchraum zu finden. Der Suchraum wird duch die Datenstruktur festgelegt, mit der potenzielle Lösungen repräsentiert werden. Jedes Individuum steht für ein Element des Suchraums.

Die Startpopulation sollte aus möglichst unterschiedlichen Individuen bestehen. Auf diese Weise kann zu Beginn der simulierten Evolution parallel an vielen verschiedenen Stellen des Suchraums gesucht werden. Der Grad der Verschiedenheit der Individuen wird als *Diversität* bezeichnet. Je unterschiedlicher die Individuen der Startpopulation sind, desto höher ist die anfängliche Diversität.

Der Evolutionszyklus Der Evolutionszyklus simuliert das Entstehen genetisch neuer Arten und das Aussterben alter Individuen, deren Fitness zum Überleben zu schlecht ist. Es gibt zwei grundsätzlich unterschiedliche Ansätze: Entweder wird gleichzeitig die komplette Population durch eine neue ausgetauscht[1] (*generationsbasierter* Ansatz) oder es wird eine kleine Teilpopulation betrachtet und innerhalb dieser werden Individuen ersetzt (*steady-state*-Ansatz). Mittels *Selektion* werden Individuen ausgewählt, die – je nach Algorithmus – entweder in der Population bleiben und/oder zur Erzeugung neuer Individuen verwendet werden.

Es gibt verschiedene Selektionsverfahren, die hier jedoch nicht näher erläutert werden sollen (siehe z.B. [9]). Die Gemeinsamkeit aller Verfahren ist die Verwendung der Individuenfitness als Selektionskriterium. Wenn der verwendete *Evolutionäre Algorithmus* eine eindeutige Zuordnung zwischen Individuen vor und nach der Selektion zulässt, werden die alten als *Elter-Individuen* und die neu erzeugten als *Nachfahren* bezeichnet.

Nach der Selektion wird das Genom der ausgewählten Individuen verändert. Dies geschieht mit Hilfe *genetischer Operatoren*. Diese können entweder das Genom eines Individuums verändern (*Mutation*) oder aus den Genomen zweier Individuen ein neues erstellen (*Rekombination/Crossover*).

Durch das Erzeugen neuer Genome übersteigt die Anzahl der Individuen die vorgegebene Populationsgröße. Je nach Algorithmus wird die Originalgröße durch

[1] Es können dabei durchaus Individuen der alten Population übernommen werden.

Löschen schlechter Individuen oder implizit durch die nächste Selektion wieder hergestellt.

Abbruchkriterien Aufgabe eines *Evolutionären Algorithmus* ist es, eine optimale Lösung zu einem vorgegebenen Problem zu finden. Jedoch kann nicht sichergestellt werden, dass diese potenzielle Lösung auch Teil des Suchraums ist[1]. Bei komplexen Problemen muss nicht einmal die Existenz einer perfekten Lösung gegeben sein. In vielen Fällen findet ein *Evolutionärer Algorithmus* zunächst nur eine suboptimale Lösung und kann diese nicht in akzeptabler Laufzeit in eine optimale Lösung überführen.

Das Finden des Optimums ist somit kein hinreichendes Abbruchkriterium. Oftmals wird statt dessen eine feste Anzahl von Fitnessauswertungen vorgegeben, nach denen eine Simulation abgebrochen wird. Ein weiterer Ansatz ist die Terminierung nach einer vorgegebenen Zahl von Evolutionszyklen ohne Fitnessverbesserung.

Neben der in dieser Arbeit verwendeten *Genetischen Programmierung* gibt es als weitere Varianten der *Evolutionären Algorithmen* die *Evolutionsstrategien*, *Genetische Algorithmen* und die *Evolutionäre Programmierung*. Die *Genetische Programmierung* wurde 1992 von KOZA vorgestellt und wird Abschnitt 2.2 näher erläutert. *Genetische Algorithmen*, 1965 von HOLLAND eingeführt [42], verwenden Bitstrings fester Länge als Datenstruktur. Diese werden als Chromosom interpretiert, auf dem die genetischen Operationen ausgeführt werden. Ein Jahr später stellte FOGEL die *Evolutionäre Programmierung* vor, die endliche Automaten codiert und ohne Rekombination auskommt [32]. Ebenfalls in den sechziger Jahren entwarfen RECHENBERG und SCHWEFEL die *Evolutionsstrategien* [93, 103]. Diese verwenden reellwertige Vektoren zur Repräsentation von potenziellen Lösungen und werden dementsprechend für Parameteroptimierungen eingesetzt. Die drei letztgenannten Verfahren sollen an dieser Stelle nicht weiter betrachtet werden, weiterführende Informationen sind der angegebenen Literatur zu entnehmen. Weitere Übersichten über *Evolutionäre Algorithmen* finden sich zum Beispiel in [9, 121].

2.2 Grundlagen der Genetischen Programmierung

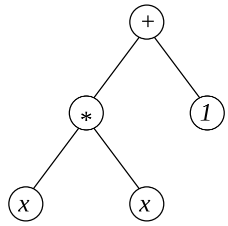
Im Allgemeinen versuchen Evolutionäre Algorithmen Lösungen zu einem Problem zu errechnen, wie zum Beispiel einen reellwertigen Lösungsvektor eines Parameteroptimierungsproblems. Ziel der Genetischen Programmierung (GP) hingegen ist das Evolvieren von Programmen, die selbst ein Problem lösen. Während zum Beispiel Evolutionsstrategien eingesetzt werden, um das Minimum einer Funktion zu berechnen, wird die Genetische Programmierung eher zur Approximierung der Funktion mittels mehrerer Stützstellen eingesetzt. Die Lösungsfunktion wird hierbei als Programm einer funktionalen Programmiersprache repräsentiert.

[1] Einschränkungen ergeben sich zum Beispiel durch die diskrete Darstellung reeller Zahlen im Computer oder durch zu restriktiv gewählte Datenstrukturen (Abschnitt 2.3.2)

2.2.1 Repräsentation

Ursprünglich wurden Bäume zur Repräsentation potenzieller Lösungen verwendet. In [57] steht ein Baum für einen funktionalen Ausdruck der Programmiersprache *LISP*. Die Größe des Baumes ist dabei nicht fixiert. Die inneren Knoten des Baumes repräsentieren jeweils eine Funktion. Die Blätter stehen für Terminale oder Variablen. Funktionen, Terminale und Variablen sind jeweils problemspezifisch zu wählen.

Beispiel 2.1
Es soll ein LISP-Ausdruck zu einer Funktion gefunden werden, deren Bild die Punkte $(0,1)$, $(1,2)$, $(2,5)$ *und* $(3,10)$ *enthält. Es ist Aufgabe des Anwenders, dem GP-System eine geeignete Menge von Funktionen vorzugeben. In diesem Fall wären zum Beispiel die vier Grundrechenarten sinnvoll. Als Konstanten könnten die Zahlen Eins, Zwei, Drei und Vier eingesetzt werden. Es wird nur eine Variable –* x *– benötigt, die den x-Wert der Stützpunkte repräsentiert.*



Der gesuchte LISP-Ausdruck ist (+ (x x) 1). Der entsprechende Baum ist rechts abgebildet.*

Das Problem selbst ist durch Testdaten definiert. In Beispiel 2.1 waren dies die vier Stützpunkte. Die einzelnen Elemente der Testdaten werden *Fitness-Cases* genannt und beinhalten jeweils Eingangs- und Ausgangsgrößen. Die Variablen, die als Blätter des Baumes verwendet werden, entsprechen jeweils einem Element der Eingangsgrößen.

2.2.2 Bewertung eines Individuums

Für jedes Individuum muss eine Fitness berechnet werden. Dieser Wert sagt aus, wie gut ein Individuum das Problem löst. Die Fitness wird berechnet, indem die Testdaten auf den Baum des Individuums angewendet werden. Der Baum wird für jeden Fitness-Case einzeln ausgewertet. Zunächst werden die Eingangsgrößen den entsprechenden Variablen der Blätter zugewiesen.[1] Der Baum wird dann – beginnend bei den Blättern – ausgewertet, bis der Wert der Wurzel bekannt ist. Der Wert der Wurzel soll nun der Ausgangsgröße des Testelements entsprechen. Die Abweichung zum Sollwert wird entweder als der Absolutwert der Differenz zwischen dem Wert der Wurzel und dem Sollwert berechnet oder als die quadrierte Differenz. Die Fitness eines Individuums wird als die Summe aller Abweichungen für sämtliche Testdaten berechnet. Der Fitnesswert Null repräsentiert somit ein Individuum, das das Problem optimal löst. Je größer der Wert ist, desto schlechter ist das Individuum.

Beispiel 2.2
Es wird eine Funktion gesucht, deren Graph durch die beiden Stützpunkte $(2,4)$ *und* $(3,1)$ *verläuft. Die Testdaten bestehen somit aus diesen beiden Fitness-Cases, wobei die X-Werte jeweils Eingangsgröße und die Y-Werte Ausgangsgröße sind. Es soll nun die Fitness*

[1]Es kann passieren, dass für eine Eingangsgröße kein oder mehr als ein Blatt existiert.

des Baumes aus Beispiel 2.1 berechnet werden, wobei für die Abweichung zur gesuchten Lösung die quadratische Differenz benutzt wird. Für den Fitness-Case $(2, 4)$ *wird der Variablen x der Wert 2 zugeordnet. Die Wurzel steht demnach für den Wert (+ (* 2 2) 1) = 5. Die quadratische Differenz zur Ausgangsgröße ist* $(5 - 4)^2 = 1$*. Für das zweite Element der Testdaten ist der Wert der Wurzel (+ (* 3 3) 1) = 10 und die quadratische Differenz somit* $(10 - 1)^2 = 81$*. Die Fitness des Individuums beträgt somit* $1 + 81 = 82$*.*

2.2.3 Initialisierung einer Population

Ein Programm, das versucht eine Problemlösung mittels Genetischer Programmierung zur errechnen, wird im Folgenden als *GP-System* bezeichnet. Die Durchführung eines Lösungsversuchs mit Initialisierung der Startpopulation, Evolutionszyklus und Abbruch wird als *GP-Lauf* bezeichnet. Zu Beginn eines GP-Laufs wird eine Startpopulation zufällig erzeugt.

Für jeden Baum wird zunächst eine Funktion für die Wurzel ausgewählt. Jede Funktion besitzt eine feste Arität. Dies ist gleichzeitig die Anzahl der zu einem inneren Knoten gehörenden Kanten in Richtung Blätter. Solange im Baum ein innerer Knoten weniger Söhne hat als die Arität der zugeordneten Funktion vorgibt, wird an entsprechender Stelle ein weiterer Knoten angehängt. Für diesen Knoten wird zufällig ein Element aus der Menge der Funktionen, Terminale und Variablen ausgewählt. Der Vorgang wird so lange fortgesetzt, bis alle inneren Knoten die richtige Anzahl von Söhnen besitzen. Diese Initialisierungsmethode wird als *grow* bezeichnet, da der Baum bei der Wurzel beginnend wächst und meist nicht vollständig ist, d.h. die Blätter haben meist keine einheitliche Entfernung zur Wurzel des Baumes.

Weitere Methoden zur Initialisierung (*full*, *ramped-half-and-half*) sind in [9] beschrieben.

2.2.4 Der Evolutionszyklus

Das von KOZA vorgestelle GP-System verwendete einen generationsbasierten Evolutionszyklus: Es wird die komplette Population durch eine neue ersetzt. Der Vorgang läuft in drei Schritten in einer Schleife ab:

- Solange keine vollständige neue Population erzeugt wurde,
  - wähle mittels Selektion ein oder zwei Individuen der gegebenen Population aus,
  - wende auf dieses Individuum/diese Individuen einen genetischen Operator an und
  - trage das so erzeugte Individuum in die neue Population ein.

Die Anzahl der zu selektierenden Individuen hängt von dem gewählten genetischen Operator ab. Nach dem Ausführen dieser Schleife wird die alte Population durch die neue ersetzt.

Selektion

Die Selektion erfolgt immer über die Fitnesswerte einzelner Individuen. Hierzu können unterschiedlichste Methoden angewendet werden:

Fitness-Proportional Selection: Die Wahrscheinlichkeit, dass ein Individuum ausgewählt wird, ist proportional zu dem Verhältnis seiner Fitness zu der Summe aller Fitnesswerte der Population.[1]

Ranked Selection: Die Fitnesswerte sämtlicher Individuen werden sortiert. Die Wahrscheinlichkeit für ein Individuum, ausgewählt zu werden, wird über eine Funktion über ihre Position in der Liste aller sortierten Individuen errechnet. Die Funktion kann wahlweise einen linearen oder exponentiellen Zusammenhang zwischen Listenposition und Selektionswahrscheinlichkeit erstellen.

Turnier-Selektion: Es wird zufällig eine kleine Teilpopulation ausgewählt – üblicherweise zwischen zwei und sieben Individuen. Diese werden nach ihrer Fitness geordnet. Ausgewählt wird das Individuum mit der besten Fitness. Alternativ können auch, beginnend beim besten Individuum, mehrere Individuen ausgewählt werden.

Genetische Operatoren

Genetische Operatoren dienen zur Erzeugung neuer Individuen aus bereits bestehenden. In der Genetischen Programmierung werden hauptsächlich zwei Operatoren eingesetzt: *Crossover* (auch *Rekombination* genannt) und *Mutation*. Zusätzlich wird die *Replikation* verwendet, bei der ein Individuum unverändert in die neue Population übernommen wird.

Crossover: Aus zwei Individuen werden durch Austausch eines Teilbaums zwei neue erzeugt. Abbildung 2.1 zeigt ein Beispiel.

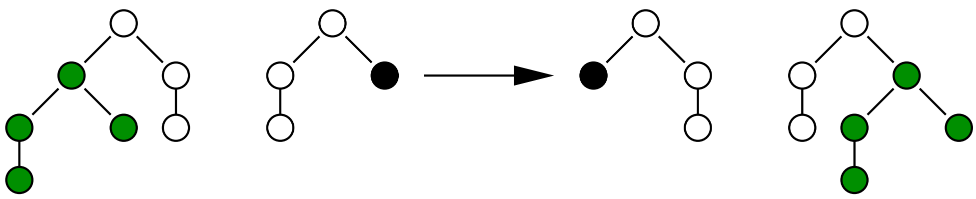


Abbildung 2.1: Beispiel für eine *Crossover*-Operation zweier Bäume

Mutation: Ein Knoten eines Baumes wird durch einen anderen ersetzt. Dieser neue Knoten kann für eine andere Funktion, eine andere Variable oder eine andere Konstante stehen. Wichtig ist nur, dass innere Knoten nur durch Funktionen und Blätter nur durch Variablen oder Konstanten ersetzt werden.

[1] Zur Berechnung wird die Fitness aller Individuen so normalisiert, dass der schlechteste Fitnesswert 0 und der beste 1 ist [9].

Die Auswahl der Operatoren erfolgt zufällig. Die Wahrscheinlichkeit, mit der ein Operator angewandt wird, wird dem GP-System dabei als Parameter übergeben. Neben der *Mutation* und dem ursprünglichen *Crossover* gibt es inzwischen viele Erweiterungen und zusätzliche Operatoren, die an dieser Stelle jedoch nicht näher erläutert werden sollen (z.B.: [112, 27, 118, 6]).

2.2.5 Abbruchkriterium

Das Abbruchkriterium legt fest, wann der – potentiell unendliche – Evolutionszyklus abgebrochen wird und das bis dahin beste gefundene Individuum als Ergebnis des GP-Laufs ausgegeben wird. Es muss abgewogen werden, ob in vertretbarer Rechenzeit ein besseres Individuum zu erwarten ist. Die verschiedenen Kriterien wurden bereits allgemein für Evolutionäre Algorithmen in Abschnitt 2.1 beschrieben. Die dortigen Bemerkungen gelten uneingeschränkt für die Genetische Programmierung.

In dieser Arbeit wird folgende Frage untersucht: Ist das neu entwickelte GP-System in der Lage, bei gleichem (Rechen-)Aufwand bessere Ergebnisse zu erzielen als herkömmliche, baumbasierte Systeme? Aus diesem Grund wird als Abbruchkriterium eine Anzahl von Fitnessauswertungen festgelegt, nach denen das GP-System terminiert.

2.2.6 Erweiterungen

Seit KOZAs Veröffentlichung von 1992 [57] wurden viele neue Ansätze zur Genetischen Programmierung vorgestellt. An dieser Stelle wird auf die Erweiterungen näher eingegangen, die zum Verständnis des neuen GP-Systems nötig sind.

Steady State-Selektion

Bei einem generationsbasierten Evolutionszyklus wird eine Population komplett durch eine neue ersetzt. In der natürlichen Evolution ist dies nicht der Fall. Vielmehr werden einige Individuen neu geboren, während andere sterben. Die *Steady-State*-Selektion versucht diesem Vorbild näher zu kommen: Es werden nur einige Individuen der Population zur Selektion herangezogen. Mit Hilfe genetischer Operatoren werden aus den selektierten Individuen Nachkommen generiert, die wiederum alte Individuen aus der selektierten Menge überschreiben.[1]

Der *Steady-State*-Ansatz wird in den meisten Fällen zusammen mit einer Turnier-Selektion verwendet. In dieser Kombination werden die Individuen eines Turniers nach ihrer Fitness sortiert und das schlechteste Individuum wird durch eine Mutation des besten (oder eine Rekombination beider Individuen) ersetzt.

[1] Es besteht somit ein direkter Zusammenhang zwischen neuen Individuen und solchen, die aus der Population ausscheiden. Dies ist ein weiterer Unterschied zur natürlichen Evolution.

Ein Turnier muss sich nicht zwangsläufig darauf beschränken, das schlechteste Individuum zu ersetzen. Statt dessen kann das zweitschlechteste Individuum ebenfalls durch eine Variation des zweitbesten Individuums ersetzt werden, usw.. Eine Turniergröße von 7/3 bezeichnet ein Turnier mit sieben Individuen, wobei Variationen der drei besten Individuen die drei schlechtesten überschreiben.

Das in dieser Arbeit vorgestellte GP-System *GGP* verwendet den *Steady-State*-Ansatz mit Turnierselektion. Die Größe der Turniere wird jeweils als Parameter bei der Vorstellung der Testprobleme angegeben.

Automatically Defined Functions (*ADF*)

In imperativen Programmiersprachen ist es möglich, häufig benötigte Abschnitte eines Algorithmus als Funktion abzulegen und diese bei Bedarf aufzurufen. *ADF*s [58] sind eine Übertragung dieses Konzepts auf die Genetische Programmierung: Ein *ADF* ist ein zusätzlicher Baum, der wie die Bäume einzelner Individuen durch genetische Operatoren verändert werden kann. Für jeden *ADF* wird die Terminalmenge um ein Element erweitert. Wird dieses Element als Blatt in den Baum eines Individuums eingefügt, wird bei der Fitnessberechnung dieses Blatt durch den Baum des *ADF*s ersetzt.

*ADF*s werden vom in dieser Arbeit vorgestellten GP-System nicht verwendet. Das zum Vergleichen der Ergebnisse verwendete baumbasierte System nach KOZA setzt sie jedoch ein. Des Weiteren werden *ADF*s in einer Beispielanwendung verwendet, um Graphstrukturen als Bäume darstellen zu können (Abschnitt 2.3.2).

Linear Genetic Programming

Neben der von KOZA verwendeten Baumstruktur gibt es weitere Datenstrukturen, durch die Genome in der genetischen Programmierung repräsentiert werden. Eine verbreitete Methode ist die Verwendung von linearen Sequenzen als lineares Genom [7, 9]. Im Unterschied zu den Genetischen Algorithmen ist die Länge eines Genoms jedoch variabel und ändert sich während des Evolutionszyklus dynamisch.

Bei der Verwendung von Bäumen, die *LISP*-Ausdrücke repräsentieren, entspricht eine Kante dem Wert des Sohnes, zu dem sie führt.[1] Betrachtet man ein lineares Genom als lineare Liste, stellen die Kanten zwischen den einzelnen Elementen keine Werte dar, sondern sagen lediglich aus, welcher Knoten als nächstes betrachtet wird. Die Knoten repräsentieren in diesem Fall Befehle, die auf Register zugreifen (vergleichbar einer Maschinensprache [83]). Die Register selbst sind nicht Bestandteil des Genoms, sondern sind Teil des Interpreters, der die Befehle ausführt. Der Ansatz unterscheidet sich somit grundlegend vom *LISP*-orientierten Baumansatz. Einige GP-Probleme sind inzwischen auch bei der Verwendung einer Baumstruktur so codiert, dass externe Register benötigt werden und die Kanten des Baumes lediglich die Ausführungsreihenfolge der Knoten festlegen. Ein Beispiel hierfür

[1] Die Auswertung erfolgt von den Blättern zur Wurzel

ist das *Artificial Ant*-Problem, das in Abschnitt 5.3.1 vorgestellt und in dieser Arbeit als Testproblem verwendet wird.

Das in dieser Arbeit eingeführte graphbasierte GP-System *GGP* kann Kanten sowohl zur Wertweitergabe als auch zur Festlegung des Programmflusses verwenden. Die beiden unterschiedlichen Ansätze werden einzeln als *funktionale* und *algorithmische* Semantik eines Graphen in den Abschnitten 3.2 und 3.4 vorgestellt.

2.3 Graphen in der Genetischen Programmierung

Wenn Graphen im Zusammenhang mit Genetischer Programmierung betrachtet werden, müssen zwei unterschiedliche Arten von GP-Systemen unterschieden werden: *graphbasierte* und *graphkodierende* GP-Systeme. Die *graphbasierten* Systeme, zu denen das im Rahmen dieser Arbeit enwickelte *GGP* gehört, verwenden Graphen als Genotypen der Individuen. *Graphkodierende* Systeme versuchen hingegen, mit einem Baum oder einem linearen Genom einen Graphen abzubilden. In Abschnitt 2.3.2 wird auf verschiedene Ansätze eingegangen, wie Graphen in verschiedenen Systemen als Baum oder lineares Genom abgelegt werden und welche Konsequenzen sich daraus ergeben. In Abschnitt 2.3.3 werden verschiedene graphbasierte GP-Systeme vorgestellt.

2.3.1 Begriffsbildung

Im Folgenden werden Graphen als zugrunde liegende Datenstruktur verwendet. An dieser Stelle werden einige Begriffe eingeführt, die zum Verständnis der restlichen Arbeit notwendig sind. Mehr zum Thema Graphentheorie ist in [29] zu finden.

Ein gerichteter Graph besteht aus *Knoten* und *Kanten*, welche die Knoten miteinander verbinden. Eine Kante, die zu einem Knoten führt, ist für diesen eine *Eingangskante*. Eine Kante, die von einem Knoten wegführt, ist dessen *Ausgangskante*. Die Anzahl der Eingangskanten eines Knotens wird als sein *Eingangsgrad* oder *Fan-in* bezeichnet, die Anzahl seiner Ausgangskanten ist sein *Ausgangsgrad* oder *Fan-out*.

Der *Startknoten* einer Kante ist der Knoten, bei dem sie beginnt. Analog ist der *Endknoten* einer Kante der Knoten, bei dem sie endet. Existieren in einem Graphen zwei Kanten, die denselben Startknoten haben und die sich ebenfalls einen Endknoten teilen, spricht man von einem *Multigraphen*.

Ein *Pfad* ist ein Kantenzug der zwei Knoten miteinander verbindet und dabei mehrere weitere Knoten beinhaltet (siehe Definition 4.2). Zwei Knoten bzw. Kanten sind miteinander verbunden, wenn sie Teil eines Pfades sind.

2.3.2 Graphkodierende GP-Systeme

Für viele Problemstellungen ist die natürliche Darstellungsform ein Graph und nicht die bei GP von KOZA verwendete Baumstruktur. In diesem Abschnitt werden verschiedene

Ansätze vorgestellt, mit denen ein Graph als Baum oder lineares Genom kodiert wird.

Cartestian Genetic Programming

In [78] stellt MILLER das *Cartestian Genetic Programming* (CGP) vor, mit dessen Hilfe digitale Schaltkreise evolviert werden sollen. In [78] wendet er es zunächst auf *Even Parity-* [57] und weitere BOOLEsche Probleme an, später ebenfalls auf *Artificial Life*-Simulationen [97].

Ein Programm bei CGP entspricht einer $m \times n$-Matrix, wobei jedes Matrixelement einer Funktion wie beispielsweise einem *AND-* oder *OR*-Gatter entspricht. Jedes Element verfügt über exakt drei Eingänge und einen Ausgang. Dieser Ausgang kann mit beliebig vielen Eingängen anderer Gatter verbunden werden oder unverbunden bleiben. Die Eingänge können nur mit Ausgängen verbunden sein, die sich in der Matrix weiter links befinden. Alternativ können sie einen der Eingangswerte des zu lösenden Problems zugewiesen bekommen.

Der Genotyp einer solchen Matrix wird als lineares Genom abgelegt, wobei jedem Element ein 4-Tupel zugewiesen ist. Die ersten drei Werte stehen für die Nummer des jeweiligen Matrixelements, von dem die Funktionen ihre Eingangswerte erhalten. Die Funktionen, die durch ein Matrixelement repräsentiert werden können, sind durchnummeriert. Die vierte Zahl des Tupels von einem Matrixelement ist die Nummer der Funktion, für die es steht.

Der evolutionäre Algorithmus hinter CGP benutzt eine $(1 + \lambda)$ Evolutionsstrategie zur Selektion [103], die Variation einzelner Individuen erfolgt über eine Mutation einzelner 4-Tupel. Bei der Mutation wird beachtet, dass die Verbindungen der Eingänge wiederum nur zu Elementen zeigen, deren Spaltennummer kleiner als die des betrachteten Elements ist.

Durch die Festlegung der Funktionen auf drei Eingänge und drei Ausgänge ist das System generell eingeschränkt. Sollen Funktionen verwendet werden, die weniger als drei Eingänge benötigen, ist dies durch Ignorieren eines Eingangswertes noch möglich, sollen jedoch Funktionen mit z.B zwei Eingängen und zwei Ausgängen verwendet werden, ist dies mit CGP nicht möglich.

Des Weiteren ist es nicht vorgesehen, dass einzelne Matrixelemente über zusätzliche Parameter verfügen können. Die Verwendung eines Proportionalgliedes aus der Regelungstechnik, bei dem der Eingangswert mit einer zum Matrixelement gehörenden reellen Zahl multipliziert würde, ist so zum Beispiel nicht möglich. Sollten entsprechende Verwendungen vorgesehen sein, müsste das lineare Genom um entsprechende Werte erweitert werden. Dies würde jedoch bedeuten, dass alle Funktionen immer dieselbe Anzahl von solchen zusätzlichen Parametern haben müssten.

Die Ergebnisse, die mit dem System im Bereich der BOOLEschen Probleme erzielt werden, sind vielversprechend, die Einschränkungen durch die Wahl des linearen Genoms schränken jedoch die Benutzbarkeit auf eine bestimmte Klasse von Anwendungsgebieten ein.

Das GP-System *SMOG* von MARENBACH

Als eine der ersten Gruppen versuchten MARENBACH *et al.* dynamische Systeme mit Hilfe der Genetischen Programmierung zu evolvieren. Zunächst wurde das GP-System für Fermentierungsprozesse verwendet [12], später dann für Aufgabenstellungen der Regelungstechnik [88]. Beispiele für die betrachteten Probleme aus der Regelungstechnik finden sich ebenfalls im Anhang A dieser Arbeit.

Die Graphen, die als Baum zu modellieren sind, sind gerichtet, und für jeden Knoten ist die Anzahl der eingehenden und herausführenden Kanten festgelegt. Zu Zeiten der ersten Veröffentlichungen war das System nur in der Lage, dynamische Systeme zu evolvieren, die sich direkt als Baum darstellen ließen [12]. Die Evolvierung von Rückführungen oder mehreren parallel liegenden Knoten war dementsprechend nicht möglich.

In dem aus diesem ersten Ansatz hervorgegangenen GP-System *SMOG* wurde die Menge der erlaubten Knotentypen um künstliche Knotentypen erweitert, mit denen die Kodierung einfacher Rückkopplungen und Vorwärtskopplungen möglich ist [72]. Trotz dieser Erweiterungen spannt die Menge aller mit dem GP-System *SMOG* evolvierbaren Bäume nicht den kompletten Suchraum der Graphen auf. Abbildung 2.2 zeigt ein entsprechendes Beispiel.

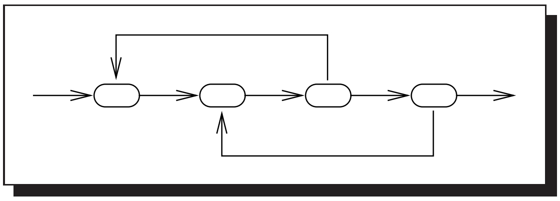


Abbildung 2.2: Ein mit *SMOG* nicht darstellbarer Graph

Graphkodierungen nach KOZA

Ähnlich MARENBACHs GP-System *SMOG* kodiert KOZA ebenfalls Schaltungsgraphen als Bäume. Zielsetzungen hierbei sind unter anderem der Entwurf von Schaltkreisen [59] oder der Entwurf von Regelungen dynamischer Systeme [60]. Die zu kodierenden Graphen enthalten somit – im Falle der digitalen Schaltkreise – Transistoren, Dioden und ähnliche Bauteile oder – im Fall der Regler – PIDs, Proportionalglieder und weitere Elemente der Regelungstechnik.

Alle diese Bauteile haben einen festgelegten *Fan-in* bzw. *Fan-out* – Eingangsgrad und Ausgangsgrad der Knoten sind somit fest vorgegeben. Sollen entsprechende Graphen als Baum kodiert werden, ergeben sich dieselben Probleme wie bei MARENBACH. KOZA entwickelte hierzu zwei verschiedene Verfahren. Im ersten Ansatz bestand der Baum nicht aus der zu kodierenden Schaltung, sondern aus einer Reihe von Anweisungen, wie aus einem vorgegebenen kleinen Schaltkreis – *Embrio-Schaltkreis* gennant – durch Einbau weiterer Elemente die eigentliche Schaltung entsteht [59]. Diese indirekte Kodierung geht auf GRUAU zurück, der auf ähnliche Weise *Neuronale Netze* erzeugte [37].

In einem weiteren Ansatz von KOZA werden die Schaltungen und Regler wie bei Marenbach als Bäume kodiert [51]. Das Problem der nicht als Baum darstellbaren Elemente der Schaltung wird durch den Einsatz von *ADFs* [58] kompensiert, da auf diese Weise eine Form von Rekursion erreicht werden kann, durch die Zyklen in Graphen simuliert werden können. Für das GP-System bedeutet dies jedoch, dass für jede Stelle der Schaltung, an der ein Signal zu zwei unterschiedlichen Schaltblöcken geführt werden soll, ein *ADF* eingesetzt werden muss. Die vor einem GP-Lauf festzulegende Anzahl der *ADFs* wirkt somit gleichzeitig als Restriktion auf die Anzahl der Verzweigungspunkte innerhalb der zu evolvierenden Schaltungen und somit als Einschränkung des Suchraums der Graphen.

2.3.3 Graphbasierte GP-Systeme

Parallel Distributed Genetic Programming (PDGP)

Bereits 1996 stellte POLI in [89] das GP-System *PDGP* (*Parallel Distributed Genetic Programming*) vor. Wie auch bei MILLER wird ein Graph durch eine $m \times n$-Matrix repräsentiert, einzelne Matrixelemente können jedoch nicht nur für Funktionen stehen, sondern zusätzlich auch für Terminale wie Variablen und Konstanten, die auch bei baumbasierten GP-Systemen als Blätter Verwendung finden.

Im Gegensatz zu MILLER verwendet POLI diese Matrixstruktur als interne Graphrepräsentation, es findet kein weiteres Mapping auf ein lineares Genom oder einen Baum statt. Ein weiterer Unterschied liegt darin, dass POLI für sein System Crossover-Operatoren entwickelt hat. Dies ist durch eine Einschränkung der erlaubten Verbindungen zwischen bestimmten Knoten gelungen. Aus diesem Grund kann eine $m \times n$-Matrix nicht alle Graphen mit $m \times n$ Knoten darstellen, sondern nur eine Untermenge.

Mit der von POLI gewählten Graphrepräsentation ist es möglich, die Anzahl der zu einem Knoten führenden Kanten exakt festzulegen, nicht jedoch die herausführende Kantenzahl. Somit ist auch POLIS System auf eine bestimmte Teilmenge aller Graphen eingeschränkt.

Parallel Algorithm Discovery and Orchestration (PADO)

TELLERS GP-System *PADO* [117] war der erste graphbasierte Ansatz, der den Genotyp eines Individuums als Graphen abspeichert. Während *PDGP* von POLI zur Abspeicherung des Graphen auf $m \times n$-Matrizen ausweicht und die Crossoveroperatoren für diese Datenstruktur definiert, werden die genetischen Operatoren bei *PADO* direkt auf Graphen angewandt.

Bei POLIs System stehen die Kanten für die Werte einer Funktion, *PADO* hingegen benutzt für diese Zwecke indizierten Speicher und einen Stack. Die Kanten stehen in diesem System für den Programmfluss, legen also fest, in welcher Reihenfolge die Funktionen der einzelnen Knoten hintereinander ausgeführt werden sollen. In dem in dieser Arbeit vorgestellten GP-System werden beide Ansätze unterstützt. Die unterschiedliche Bedeutung der Kanten wird unter den Stichworten *funktionale Semantik* (Abschnitt 3.2) und *algorithmische Semantik* (Abschnitt 3.4) näher beschrieben.

Jeder Knoten im *PADO*-GP-System kann beliebig viele Eingangs- und Ausgangskanten besitzen. Die einzige Festlegung besteht darin, dass nur ein als Endknoten markierter Knoten den Ausgangsgrad null hat und nur ein als Startknoten markierter den Eingangsgrad null.

Ein Knoten repräsentiert jeweils zwei Funktionen. Die erste entspricht den in GP-Systemen üblichen Funktionen wie zum Beispiel arithmetischen Operationen. Die zweite Funktion entscheidet als Auswahlfunktion darüber, welche Ausgangskante benutzt werden soll, um den im Programmfluss folgenden Knoten zu ermitteln.

Zum Crossover werden aus zwei Individuen jeweils meherere Knoten herausgenommen und in das andere Individuum eingesetzt. Die beim Herausnehmen aufgeteilten Kanten werden zufällig mit den Knoten des anderen Individuums verbunden. Die Auswahl der Knoten erfolgt über eine Metaevolution, d.h. es existiert ein weiterer Graph, der ein Programm darstellt, welches beim eigentlichen Graphen die Knoten für den Crossover auswählt. Bei diesem Metaprogramm, das ebenfalls dem Zyklus aus Selektion und Variation unterworfen ist, werden die Knoten für einen Crossover zufällig ausgewählt.

Im Bereich der Klassifikation konnten mit dem System gute Ergebnisse erzielt werden (Bilderkennung in [117] und [115], Geräuscherkennung in [116]). Versuche im Bereich der symbolischen Regression waren hingegen weniger erfolgreich [13]. Ein Problem des Systems liegt darin, dass normalerweise mehr Variablen vom Stack benötigt werden, als auf diesem vorhanden sind. Aus diesem Grund muss ein großer Teil des Graphen immer aus Konstantenfunktionen bestehen.

Für Probleme aus den Bereichen des Schaltungs- und Reglerentwurfs ist das System nicht geeignet, da Eingangs- und Ausgrangsgrade einzelner Knoten nicht vorgegeben werden können.

Ein GP-System für Turingmaschinen

In [87] zeigen Pereira *et al.*, wie sich eine Turingmaschine [43] als Graph kodieren und mit Hilfe von GP evolvieren lässt. Als Problemstellung verwenden sie das *Busy Beaver*-Problem, bei dem es darum geht, bei einer vorgegebenen Anzahl von Zuständen eine möglichst große, jedoch endliche Zahl von Einsen auf ein vorher leeres Band zu schreiben.

Die Repräsentation einer Turingmaschine innerhalb des Genoms eines GP-Individuums erfolgt über eine Identifizierung der Knoten eines gerichteten Graphens mit den Zuständen der Turingmaschine, die Kanten stehen für die Transitionen zwischen den Zuständen. Aus allen Zuständen außer dem Endzustand führen genau zwei Kanten heraus. Der Endzustand hat keine Ausgangskante. Die erste Kante benennt den Folgezustand, wenn auf dem Turingband eine Null steht, die zweite Kante den Folgezustand bei einer Eins. Der Eingangsgrad der Knoten ist nicht festgelegt.

Der Graph wird intern als lineares Genom verwaltet. Da jedoch der im Folgenden kurz beschriebene Crossover-Operator direkt auf dem Graphen und nicht auf dem linearen Genom arbeitet, kann im Gegensatz zu Millers *CGP*-System von einem graphbasierten GP-System gesprochen werden.

Beim Crossover werden zusammenhängende Teilgraphen aus zwei Individuen miteinander vertauscht. Die Teilgraphen bestehen beide aus gleich vielen Knoten. Jedem Knoten aus dem ersten Graph wird einer aus dem zweiten zugeordnet, und die Knoten werden gemäß dieser Zuordnung ausgetauscht. Die genaue Vertauschung der Kanten ist der Quelle zu entnehmen [87]. Es wird hierbei darauf geachtet, dass weiterhin jeder Knoten über die zwei entsprechenden herausführenden Kanten verfügt. Bei dem verwendeten Algorithmus werden nur verhältnismäßig wenige Kanten aus dem neuen Teilgraphen übernommen

Das GP-System von PEREIRA kann nur für Problemstellungen mit fester Knotenzahl angewendet werden, bei denen die Anzahl der Ausgangskanten aller Knoten identisch ist. Für allgemeinere Problemstellungen, zum Beispiel aus der Regelungstechnik, ist es nicht geeignet.

Linear-Graph GP

Bei dem in [50] von KANTSCHIK vorgestellten GP-System *Linear-Graph GP (LGGP)* handelt es sich primär um ein System mit linearem Genom, das in mehrere Abschnitte unterteilt ist. Ab dem zweiten dieser Genomabschnitte existieren zu jedem mehrere alternative Genome, die nicht zwingend gleichlang sein müssen. Die Auswahl der Alternative des jeweiligen Folgeabschnitts erfolgt über eine Verzweigungsfunktion im letzten Knoten des aktuellen Abschnitts. Diese Funktionen können in der aktuellen Implementierung zwischen ein und drei nachfolgenden Genomabschnitten auswählen. Abbildung 2.3 zeigt ein Beispiel.

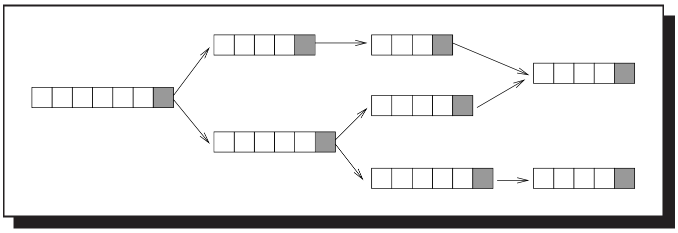


Abbildung 2.3: Beispiel eines *Linear-Graph*-Genoms

Werden die einzelnen Genomabschnitte als Knoten eines Graphen betrachtet und die möglichen Verzweigungen zu den jeweils nächsten Abschnitten als Kanten zwischen diesen, dann entsteht ein gerichteter, zyklenfreier Graph, bei dem in der Mikroansicht jeder Knoten wiederum aus einem linearen Genom besteht.

Das System verfügt über Mutations- und Crossover-Operatoren. Es können Befehle innerhab eines linearen Abschnitts verändert werden oder Kanten (die Ziele der Verzweigungsbefehle) ausgetauscht werden. Beim Crossover werden einzelne lineare Genomabschnitte mit anderen vertauscht. Die neue Zuordnung der von diesen Abschnitten aus erreichbaren Genome des nächsten Abschnittes erfolgt zufällig, wobei darauf geachtet wird, dass jede Alternative weiterhin zu erreichen ist.

Das System hat sich im Bereich der symbolischen Regression und Klassifikation bewährt [50], ist jedoch nicht für eine direkte Kodierung von Schaltkreisen und ähnlichen Graphen geeignet.

2.3.4 Zusammenfassung

In diesem Abschnitt wurden verschiedene Zusammenhänge aufgezeigt, in denen Graphen in der Genetischen Programmierung eingesetzt werden. Zunächst wurden Systeme vorgestellt, bei denen die natürliche Darstellungsweise des zu ermittelnden GP-Individuums ein Graph wäre, dieser aber durch einen Baum oder ein lineare Genom repräsentiert werden soll. All den hier besprochenen Ansätzen ist gemein, dass durch die Übertragung in eine restriktivere Datenstruktur der vom GP-System aufgespannte Suchraum immer kleiner ist, als der Raum aller für eine Lösung in Frage kommenden Graphen. In allen referenzierten Veröffentlichungen wurden die Ergebnisse des jeweiligen Systems als gut bezeichnet, es ist jedoch durchaus möglich, dass durch die Beschneidung des Suchraums bessere Lösungen von vornherein ausgeschlossen werden.

Des Weiteren wurden GP-Systeme vorgestellt, die Graphen als Datenstruktur für den Genotyp der Individuen verwenden. Diese wurden entweder explizit für spezielle Problemstellungen entworfen oder die verwendeten genetischen Operatoren sind nicht dafür ausgelegt, bestimmte Eigenschaften von Graphen zu erhalten (z. B. eine feste Zuordnung von in einen Knoten hineinführenden und ihn verlassenden Kanten).

Aus diesen Überlegungen ergab sich somit die Zielsetzung, ein graphbasiertes GP-System zu erstellen, das einerseits Schaltkreise und Graphen aus Anwendungen der Regeltechnik auf natürliche Weise als Genotyp verwendet, andererseits jedoch auch für Problemstellungen geeignet ist, die mit baumbasierten GP-Systemen bearbeitet werden. Eine besondere Herausforderung ist die Entwicklung geeigneter Crossover-Operatoren, die weitaus kompliziertere Operationen ausführen müssen als beim Baum-Crossover und die in anderen Publikationen als kompliziert zu realisieren eingestuft wurden [36].

Kapitel 3

Repräsentation der Individuen

Jedes Individuum einer Population wird im GP-System durch einen Graphen repräsentiert. Um Verwechslungen mit anderen Graphdefinitionen zu vermeiden, werden diese im Folgenden *GP-Graphen* genannt. In diesem Kapitel wird ihre Definition sowie ihre Verwendung im GP-System vorgestellt.

Bei GP-Graphen muss zwischen der *Syntax* und der *Semantik* des Graphen unterschieden werden. Unter der Syntax eines Graphen versteht man den Zusammenhang seiner Knoten. Die Semantik hingegen definiert die Bedeutung der Kanten und Knoten. Während die genetischen Operatoren des GP-Systems die Syntax eines Graphen verändern, bestimmt die Semantik, wie bei einer Fitnessberechnung zu verfahren ist.

Untersucht werden funktionale und algorithmische Semantiken. Bei funktionalen Semantiken steht jeder Knoten für eine Funktion. Die Dimension des Definitionsbereichs der Funktion entspricht dem Eingangsgrad des Knotens, die Dimension des Bildbereichs der Funktion dem Ausgangsgrad des Knotens. Beispiele für funktionale Semantiken sind eine Boolesche Semantik, bei der jeder Knoten für ein logisches Gatter steht oder eine *arithmetische* Semantik, bei der jeder Knoten für eine arithmetische Funktion wie der Addition steht.

Bei algorithmischen Semantiken stehen die Knoten ebenfalls für Funktionen, allerdings sind die Definitionsbereiche und Bilder der Funktionen nicht mit den Kanten verknüpft, sondern ergeben sich aus zusätzlichen Datenstrukturen. Die Kanten definieren lediglich die Reihenfolge, in denen die Funktionen der Knoten ausgeführt werden.

Beispiele für entsprechende Semantiken sind die in Abschnitt 5.3.1 vorgestellte Kodierung des *Artifical Ant*-Problems oder imperative Programme, die als Flussdiagramme dargestellt werden.

In den folgenden Abschnitten wird zuerst die Syntax eines GP-Graphen definiert. Danach werden die in dieser Arbeit verwendeten Semantiken beschrieben.

3.1 Syntax von GP-Graphen

Definition 3.1
Ein 4-Tupel (I, O, V, E) heiße GP-Graph, wenn folgende Bedingungen erfüllt sind:

1. *$I = \{i_1, \ldots, i_n\}$ und $O = \{o_1, \ldots, o_m\}$ sind endliche Mengen, über die jeweils eine Ordnung definiert ist.*
2. *$V = \{v_1, \ldots, v_p\}$ ist eine endliche Menge.*
3. *$E = \{(a, b) | a \in I \cup V \wedge b \in V \cup O\}$, E kann mehrere identische Elemente enthalten.*
4. *Für alle $i \in I$ existiert genau ein Tupel $(i, b) \in E$ mit $b \in V \cup O$.*
5. *Für alle $o \in O$ existiert genau ein Tupel $(a, o) \in E$ mit $a \in I \cup V$.*
6. *Zu jedem Element $v \in V$ ist auf den beiden Mengen $\{(a, v) | a \in I \cup V \wedge (a, v) \in E\}$ und $\{(v, b) | b \in V \cup O \wedge (v, b) \in E\}$ jeweils eine Ordnung definiert.*
7. *Zu jedem Element $v \in I \cup V$ existiert ein Pfad[1], von v zu einem Element $o \in O$.*
8. *Zu jedem Element $v \in V$ existiert mindestens ein Pfad, der bei einem Element $i \in I \cup V$ mit Eingangsgrad 0 beginnt und zu v führt.*

Durch die ersten drei Punkte der Definition wird ein gerichteter, endlicher Multigraph definiert. Hierbei ist I die Menge der Eingangsknoten und O die der Ausgangsknoten sowie V die Menge der (semantisch gesehen) inneren Knoten des Graphen.[2]

Die Punkte 4 und 5 legen fest, dass jeder Eingangsknoten den Ausgangsgrad 1 hat und alle Ausgangsknoten den Eingangsgrad 1 haben.

Punkt 6 stellt sicher, dass sowohl für die Eingangskanten als auch für die Ausgangskanten sämtlicher Knoten eine Reihenfolge festgelegt ist. Nur so können Knoten nichtkommutative Funktionen wie die Subtraktion, repräsentieren.

Durch die Punkte 7 und 8 wird der Suchraum für das GP-System weiter auf sinnvolle Graphen eingeschränkt. Punkt 7 stellt hierbei sicher, dass jeder Knoten in dem Graphen Einfluss auf die Lösung hat. Hierdurch wird eine mögliche Art von Introns vermieden. Ziel des GP-Systems soll es sein, mittels kleiner Graphveränderungen eine gerichtete Suche durchzuführen, wobei Introns keinen positiven Nutzen hätten, während Introns im klassischen Baum-GP häufig als Schutz für *gute* Teilbäume oder als Material zur Diversitätserhaltung betrachtet werden [9].

Punkt 8 stellt sicher, dass jeder Knoten, der einen Eingangsgrad größer null hat, mit einem Knoten verbunden ist, der über keine eigene Eingangskante verfügt. Wenn für azyklische Graphen Punkt 7 der Definition erfüllt ist, gilt automatisch auch Punkt 8. Bei Problemstellungen, für die zyklischen Graphen mit algorithmischer Semantik eingesetzt werden, stellt

[1] siehe Definition 4.2 auf Seite 39
[2] Es kann sehr wohl Knoten $v \in V$ geben, die einen Eingangsgrad null haben.

der Punkt sicher, dass jeder Knoten bei der Berechnung des Graphen erreichbar ist und somit kein Intron darstellt.

Bei funktionalen Semantiken werden im Allgemeinen nur azyklische Graphen verwendet, da Zyklen ein Gleichungssystem darstellen, das nicht notwendigerweise eine Lösung haben muss. In so einem Fall kann es passieren, abhängig von den Knoten außerhalb des Zyklus, dass für den Graphen keine Fitness berechnet werden kann.

Beispiel 3.1
Abbildung 3.1 zeigt einen Graphen mit funktionaler Semantik und einem Zyklus, bei dem die Knoten des hervorgehobenen Teilgraphen gegen Punkt 8 der Definition 3.1 verstoßen. Die Ausgangskante des Teilgraphen – und somit der gesamte Graph – kann nur in einem Fall für einen festen Wert stehen: Die Summe der Werte der Kanten, die in den Additionsblock des Teilgraphen führen, muss gleich dem Ergebnis dieser Addition sein.

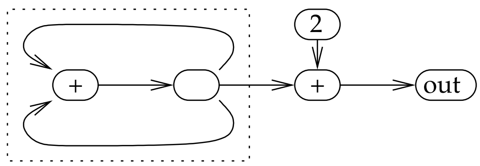


Abbildung 3.1: Die Auswertung des markierten Teilgraphen entspricht einem Gleichungssystem ($val = val + val$).

3.2 Funktionale Semantiken eines GP-Graphen

Aufgabe einer funktionalen Semantik eines GP-Graphen ist es, jedem Knoten der Menge V eines GP-Graphen jeweils eine Funktion zuzuordnen. Die Ergebnisse, die sich durch die Auswertung dieser Funktion ergeben, werden über die Ausgangskanten eines Knotens an die folgenden Knoten weitergegeben. Durch die Funktion, die zu einem Knoten v gehört, werden gleichzeitig Eingangsgrad $id(v)$ und Ausgangsgrad $od(v)$ desselben festgelegt. Bei der Berechnung des Wertes eines Graphen erhalten die Ausgangskanten eines Knotens die Werte, die sich aus der dem Knoten zugeordneten Funktion berechnen. Die Argumente der Funktion ergeben sich aus den Werten der Eingangskanten des Knotens, die ihrerseits wiederum Ausgangskanten anderer Knoten sind.

Zusätzlich kann eine Funktion weitere Argumente haben, die nicht von den Eingangskanten des zugehörigen Knotens abhängen, sondern lokale, zum Knoten gehörende Parameter sind.

Für eine Funktion f, die den Wert Eingangskante mit dem Faktor k multipliziert und dann an die Ausgangskante propagiert, ist der Parameter k zum Beispiel ein für den jeweiligen Knoten lokaler Wert. Die Funktion berechnet bei einem Knoten mit $k = 2$ der Wert $f(x,k)|_{k=2} = 2x$. Bei einem anderen Knoten hingegen, für den $k = 0.3$ gilt, wird der Wert $f(x,k)|_{k=0.3} = 0.3x$ erzeugt.

Definition 3.2
Eine Funktionenschar

$$F = \left\{ f_a : \mathbb{R}^{i_a} \times L \to \mathbb{R}^{o_a} \; | L \subseteq \mathbb{R}^{p_a}, \; i_a, p_a \in \mathbb{N}_0, \; o_a \in \mathbb{N} \right\},$$

heiße Menge der Grundfunktionen für funktionale Semantiken. Eine einzelne Funktion $f \in F$ mit $f : \mathbb{R}^i \times L \to \mathbb{R}^o$ heiße (i, o)-Grundfunktion.

Die Werte i_a und o_a entsprechen somit dem Eingangsgrad und Ausgangsgrad eines Knotens, der für diese Funktion f_a steht. L bezeichnet den lokalen Parameterraum der Funktion. Da der Definitionsbereich der p_a lokalen Parameter unterschiedlich sein kann, handelt es sich hierbei um eine Teilmenge des $\mathbb{R}^{p_a}$.

Wenn $i_a = 0$ für eine Funktion f_a gilt, bedeutet dies, dass die Funktion über keine Argumente verfügt. Ein zu dieser Funktion passender Knoten hat somit keine Eingangskante.$id(v) = 0$.

Andererseits kann es auch Funktionen geben, die über keine zusätzlichen Parameter verfügen, für die also $p_a = 0$ gilt. Beispiel hierfür sei $f : (x_1, x_2)^T \mapsto x_1 + x_2$.

Zur Beschreibung einer Semantik wird jedem Knoten v des Graphen eine Funktion $f_a \in F$ zugeordnet, wobei darauf geachtet werden muss, dass Eingangs- und Ausgangsgrad des Knotens zur Funktion passen. Die Reihenfolge der Kanten, die nach Definition 3.1 existieren muss, entspricht der Reihenfolge der Argumente der Funktion.

Jeder Kante eines Graphen ist eine reelle Zahl zugeordnet. Sind zu einem Knoten die Werte aller Eingangskanten bekannt, können mit Hilfe der dem Knoten zugeordneten Funktion die Werte der Ausgangskanten berechnet werden. Der Ergebnisvektor der Funktion wird hierbei komponentenweise den Ausgangskanten zugeordnet.

Die Dimension p_a der zweiten Komponente des Definitionsbereichs einer Funktion f_a steht für die Anzahl der zusätzlichen Parameter, die ein Knoten aufgrund der ihm zugewiesenen Funktion benötigt. Um eine übersichtliche Schreibweise zu finden, werden im Folgenden die beiden Komponenten des kartesischen Produkts des Definitionsbereich jeweils als Vektor dargestellt. Wenn für eine Funktion f_a entweder $i_a = 0$ oder $p_a = 0$ gilt, wird die entsprechende Komponente bei Wertzuweisungen nicht geschrieben.

Definition 3.3
Gegeben seien eine Grundfunktionsmenge für funktionale Semantiken F und ein GP-Graph (I, O, V, E). Das Funktionenpaar $G_{func} : V \to F$ und $G_{par} : V \to \mathbb{R} \cup \mathbb{R}^2 \cup \cdots \cup \mathbb{R}^d$ heiße Semantik des GP-Graphen (I, O, V, E), wenn gilt:

- *Für jeden Knoten $v \in V$ mit $G_{func}(v) = f_a$ und $f_a : \mathbb{R}^{i_a} \times L \to \mathbb{R}^{o_a}$ mit $L \subseteq \mathbb{R}^{p_a}$ gilt:*

$$i_a = id(v) \quad \text{und} \quad o_a = od(v)$$

- *Bei $p_a \neq 0$ muss zusätzlich gelten:*

$$dim(G_{par}(v)) = p_a$$

- *Für alle Funktionen $f_a \in F$ mit $f_a : \mathbb{R}^{i_a} \times L \to \mathbb{R}^{o_a}$, $L \subseteq \mathbb{R}^{p_a}$ und $p_a \neq 0$ gilt, dass L Teilmenge des Bildbereichs von G_{par} ist, also*

$$p_a \neq 0 \Rightarrow L \subseteq im(G_{par}).$$

Die Funktionen G_{func} und G_{par} stellen die Verbindung zwischen Funktionen und Knoten her: Die Anzahl der Argumente einer Funktion muss dem Eingangsgrad des Knotens entsprechen, die Dimension des Lösungsvektors einer Funktion passt zum Ausgangsgrad des Knotens und der Knoten verfügt über die von der Funktion benötigte Anzahl lokaler Parameter.

Der dritte Punkt stellt sicher, dass zu jeder möglichen Neuzuordnung der Funktionen zu den Knoten eine gültige Funktion G_{par} existiert.

3.3 Bewertung eines GP-Graphen mit funktionaler Semantik

Zur Bewertung eines GP-Graphen (I, O, V, E) werden allen Eingangsknoten aus I Werte zugewiesen, die dann über die aus diesen Knoten herausführenden Kanten entweder an innere Knoten aus V oder direkt an die Ausgangsknoten aus O weitergereicht werden. Durch die Syntax ist festgelegt, wie Werte zwischen den Knoten weitergeleitet werden, die Semantik legt fest, wie die Werte innerhalb der einzelnen Knoten verändert werden.

Ist der Wert einer zu einem Ausgangsknoten führenden Kante bekannt, so wird dieser Wert dem entsprechenden Knoten $o \in O$ zugeordnet. Sind die Werte aller Ausgangsknoten $o_1, \ldots, o_l$ bekannt (mit $O = \{o_1, \ldots, o_l\}$), so ergibt sich ein Vektor aus l Komponenten. Dieser wird als *Lösungsvektor* bezeichnet.

Die Werte, die den Eingangsknoten zu Beginn der Berechnung zugewiesen werden, sind immer durch die Problemstellung vorgegeben. Für einige Probleme reicht es, den Graphen mit einer Belegung der Eingangsknoten zu berechnen. Für andere Probleme mit mehreren Testcases können entsprechend viele Berechnungen mit unterschiedlichen Werten für die Eingangsknoten notwendig sein.

Beispiel 3.2
Gegeben sei der GP-Graph (I, O, V, E) mit

- $I = \{i_1, i_2\}$, $O = \{o_1\}$, $V = \{v_1, v_2, v_3\}$,
- $E = \{(i_1, v_1), (i_2, v_1), (v_1, v_3), (v_2, v_3), (v_3, o_1)\}$ *sowie folgender Reihenfolgen*
- $i_1 < i_2, (i_1, v_1) < (i_2, v_1)$ *und* $(v_1, v_3) < (v_2, v_3)$

Die Menge der Grundfunktionen $F = \{f_{add}, f_{sub}, f_{const}\}$ sei gegeben mit

- $f_{add} : \mathbb{R}^2 \times \varnothing \to \mathbb{R}$ *und* $f_{add} : (x_1, x_2)^T \mapsto x_1 + x_2$

- $f_{sub} : \mathbb{R}^2 \times \varnothing \to \mathbb{R}$ *und* $f_{sub} : (x_1, x_2)^T \mapsto x_1 - x_2$
- $f_{const} : \varnothing \times \mathbb{R} \to \mathbb{R}$ *und* $f_{const} : p \mapsto p$

Als Semantik werden die Funktion $G_{func} : V \to F$ *und* $G_{par} : V \to \mathbb{R}$ *benutzt:*

$$\begin{array}{lll} G_{func} : v_1 \mapsto f_{add} & G_{func} : v_2 \mapsto f_{const} & G_{func} : v_3 \mapsto f_{sub} \\ G_{par} : v_1 \mapsto 99 & G_{par} : v_2 \mapsto 4 & G_{par} : v_3 \mapsto -4 \end{array}$$

Der Wert des Graphen berechnet sich bei der Wertzuweisung $i_1 := 7$ *und* $i_2 := 5$ *zu* $o_1 = 8$. *Abbildung 3.2 zeigt den zugehörigen Graphen. Die Werte der Funktion* G_{par} *für* v_1 *und* v_3 *haben keinen Einfluss auf das Ergebnis und dienen nur der wohldefiniertheit der Funktion.*

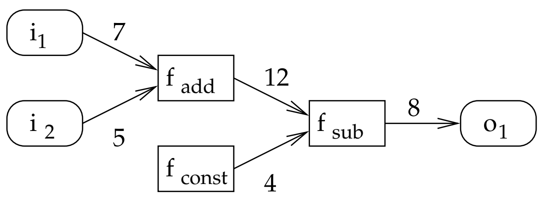


Abbildung 3.2: Der Graph zu Beispiel 3.2. Die Zahlen an den Kanten sind die Werte, die von den Eingängen zu den inneren Knoten propagiert werden bzw. die Werte, die für die inneren Knoten errechnet wurden

Am Beispiel werden folgende Punkte deutlich:

- Jeder Baum eines nach KOZAs Verfahren arbeitenden GP-Systems kann in einen GP-Graphen mit entsprechender Semantik umgewandelt werden. Hierzu wird, von der bisherigen Wurzel ausgehend, eine neue Kante zu einem neuen Knoten hinzugefügt, der die Funktion des Ausgangsknoten übernimmt. Die inneren Knoten des Baumes werden der Menge der inneren Knoten V des neuen GP-Graphen zugeordnet und die Blätter entweder den inneren Knoten oder den Eingangsknoten. Dies ist abhängig davon, ob es sich bei der Funktion der Blätter um eine Verarbeitung von Eingangswerten des durch den Baum repräsentierten Problems handelt. Bäume nach KOZA können somit als Spezialfall von GP-Graphen angesehen werden.

- Obwohl die Funktionen, die den Knoten v_1 und v_3 zugeordnet sind, keine lokalen Parameter benötigen, werden ihnen durch G_{par} Werte zugewiesen. Diese Werte werden bei der Auswertung des Graphen ignoriert und dienen nur der Wohldefiniertheit von G_{par}.

- Am Beispiel des Knotens v_3 zeigt sich, dass es wichtig ist, bei der Syntax-Definition eines GP-Graphen eine Ordnung auf den Eingängen der Knoten vorauszusetzen. Nur so ist es möglich, Grundfunktionen zu benutzen, die nicht kommutativ sind.

Die Reihenfolge der Auswertung der inneren Knoten richtet sich in jedem einzelnen Fall danach, für welche Knoten bereits alle Werte der eingehenden Kanten bekannt sind. Es lässt sich leicht zeigen, dass alle azyklischen GP-Graphen berechenbar sind. GP-Graphen, die Zyklen enthalten sind unter Umständen nicht berechenbar.

Satz 3.1
Gegeben sei ein azyklischer GP-Graph (I, O, V, E) sowie eine passende Semantik (G_{func}, G_{par}). Bei gegebenem Eingangsvektor **i** *lässt, sich der zugehörige Ausgangsvektor* **o** *berechnen.*

Beweis
Der Beweis erfolgt durch das Zeigen eines Widerspruchs. Wäre der Wert des Graphen nicht berechenbar, so müsste der Wert eines inneren Knotens nicht berechenbar sein, von dem ein Pfad zu einem Ausgangsknoten führt. Der Wert dieses inneren Knotens ist nur dann nicht berechenbar, wenn der Wert einer seiner Eingangskanten nicht berechnet werden kann. Dies kann nur der Fall sein, wenn der Wert des inneren Knotens, von dem diese ausgeht, nicht berechenbar wäre. Verfügt der Graph über n innere Knoten, so kann diese Argumentation so lange iteriert werden, bis sich ein Pfad von n hintereinander liegenden inneren Knoten ergibt, die alle aufgrund ihres nicht berechenbaren Vorgängers ebenfalls nicht berechenbar sind. Diese Aussage führt direkt zum Widerspruch, da der erste Knoten des Pfades entweder

- berechenbar sein muss, weil nur Kanten von Eingangsknoten zu ihm führen oder weil er über keine eingehende Kante verfügt oder
- eine seiner Eingangskanten von einem inneren Knoten ausgeht, der bereits im Pfad hinter ihm verwendet wurde – was bedeuten würde, dass der Graph zyklisch ist.

■

Zur Betrachtung zyklischer GP-Graphen muss somit bei der Wahl der Semantik darauf geachtet werden, dass die Berechnung des Graphen terminiert.

3.4 Algorithmische Semantiken

Die Semantikdefinition aus Abschnitt 3.2 eignet sich für Problemstellungen, bei denen Ausdrücke wie Polynome oder boolesche Ausdrücke evolviert werden sollen. Für Probleme, bei denen die Lösung ein Algorithmus sein soll, sind sie ungeeignet.

In solchen Fällen kann der Graph mit einer Semantik versehen werden, die ihn als Flussdiagramm interpretiert. Die Funktionen, die durch einen Knoten repräsentiert werden, haben somit zwei Aufgaben:

- Sie führen Berechnungen durch.
- Sie entscheiden, welcher Knoten und somit welche Funktion als nächstes ausgeführt wird.

Die Kanten des Graphen legen wie bei einem Flussdiagramm fest, in welcher Reihenfolge die Knoten betrachtet werden sollen. Hieraus ergibt sich, dass Kanten im Gegensatz zu den funktionalen Semantiken nicht zur Übermittlung von Funktionswerten zur Verfügung stehen. Die Variablen, die von den Funktionen beeinflusst werden, sind somit nicht explizit im Graphen vorhanden, sondern werden implizit über die Wahl der Grundfunktionen (Definition 3.4) eingeführt.

Für einen Knoten v mit dem Ausgangsgrad $od(v) = 1$ ist klar, dass nach der Berechnung der Funktion des Knotens als nächstes der Knoten betrachtet wird, zu dem die herausgehende Kante führt. Verfügt ein Knoten über einen höheren Ausgangsgrad $od(v) > 1$, so muss die Funktion, die durch den Knoten repräsentiert wird, ebenfalls berechnen, zu welchem Knoten als nächstes verzweigt wird.

Die Grundfunktionen, die für algorithmische Semantiken gewählt werden, haben als Definitionsbereich ein kartesisches Produkt aus einer Parametermenge und den bereits von den *Grundfunktionen für funktionale Semantiken* bekannten, lokalen Parametern. Der Bildbereich ergibt sich als Produkt aus Parametermenge und drei natürlichen Zahlen. Die Parametermenge repräsentiert die Variablen, die von der Funktion verändert werden. Die erste natürliche Zahl steht für die Nummer der Ausgangskante, über die der nächste Knoten erreicht wird. Hierbei entspricht die Nummer der Kante der Reihenfolge der Ausgangskanten, die durch die Syntax des GP-Graphen vorgegeben wurde.

Die zweite und dritte natürliche Zahl stehen für den Eingangsgrad und den Ausgangsgrad, die ein Knoten haben muss, dem die entsprechende Funktion zugeordnet ist. Diese Komponenten des Bildbereichs sind nötig, damit für den GP-Graphen bei algorithmischen Semantiken die gleichen Voraussetzungen gelten wie bei den funktionalen Semantiken. Auf diese Weisekann für beide Formen dasselbe GP-System mit identischen genetischen Operatoren benutzt werden.[1] Die Werte für Ein- und Ausgangsgrad müssen für alle Werte des Definitionsbereichs konstant sein. Die Nummer der nächsten zu betrachtenden Kante muss immer kleiner oder gleich dem Ausgangsgrad sein.

Definition 3.4
Eine Funktionenschar

$$F = \{f_a : P \times L \to P \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \quad | \quad L \subseteq \mathbb{R}^{p_a}, p_a \in \mathbb{N}\}$$

heiße Menge der Grundfunktionen für algorithmische Semantiken, wenn für alle Funktionen f_a die Menge P identisch ist und für jede Funktion f_a mit $f_a : (\boldsymbol{p}, \boldsymbol{q}) \mapsto (\boldsymbol{r}, n, i, o)$ gilt:

- *Für alle Elemente des Definitionsbereichs bleiben i und o konstant.*
- *Für alle Elemente des Definitionsbereichs gilt $n \leq o$.*

Eine Grundfunktion f mit $f : (\boldsymbol{p}, \boldsymbol{q}) \mapsto (\boldsymbol{r}, n, i, o)$ heiße (i,o)-Grundfunktion.

[1] Für ein GP-System, das ausschließlich Algorithmen evolvieren soll, ist eine Festlegung des Eingangsgrades eines Knotens nicht nötig.

Obwohl die Definitionen der Grundfunktionen für funktionale und algorithmische Semantiken unterschiedlich sind, gilt für beide, dass der Knoten, der zu einer (i, j)-Grundfunktion gehört, den Eingangsgrad i und den Ausgangsgrad j hat.

Definition 3.5
Ein Knoten mit dem Eingangsgrad i und dem Ausgangsgrad j wird (i, j)-Knoten genannt.

Beispiel 3.3
Betrachte $f : \mathbb{R}^3 \times \mathbb{R}^2 \to \mathbb{R}^3 \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, *mit*

$$f : ((a_1, a_2, a_3)^T, (x, y)^T) \mapsto \begin{cases} ((a_{mod(y,3)+1}, a_2, a_3)^T, 1, 1, 1) & \text{falls mod(x,3) = 0} \\ ((a_1, a_{mod(y,3)+1}, a_3)^T, 1, 1, 1) & \text{falls mod(x,3) = 1} \\ ((a_1, a_2, a_{mod(y,3)+1})^T, 1, 1, 1) & \text{falls mod(x,3) = 2,} \end{cases}$$

wobei $mod(a, b)$ *für die Berechnung von* a *Modulo* b *steht.*

Diese Funktion kann bei Problemen eingesetzt werden, bei denen ein Feld a mit drei Elementen als Variable benutzt wird. Die Funktion wird durch einen $(1, 1)$*-Knoten* v *repräsentiert und berechnet (in C-Syntax) die Anweisung* $a[x\%3] = a[y\%3]$.

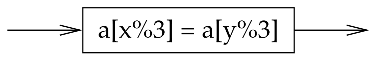


Abbildung 3.3: Ein Knoten, der die Funktion aus Beispiel 3.3 repräsentiert. Er verändert das globale Feld **a** mit Hilfe der lokalen Parameter x und y.

Eine algorithmische Semantik weist jedem inneren Knoten eines GP-Graphen eine Funktion aus der Menge der Grundfunktionen und eine lokale Parametermenge zu. Hierbei muss bei den Funktionen darauf geachtet werden, dass die Ein- und Ausgangsgrade der Knoten zu den zugeordneten Funktionen passen. Bei den lokalen Parametern gelten die gleichen Bedingungen wie in Definition 3.3.

Definition 3.6
Gegeben seien die Menge der Grundfunktionen für algorithmische Semantiken F *und ein GP-Graph* (I, O, V, E). *Das Funktionenpaar* $G_{func} : V \to F$ *und* $G_{par} : V \to \mathbb{R} \cup \mathbb{R}^2 \cup \cdots \cup \mathbb{R}^d$ *heiße Semantik des GP-Graphen* (I, O, V, E), *wenn gilt:*

- *Für jeden Knoten* $v \in V$ *mit* $G_{func}(v) = f_a$ *und* $f_a : P \times L \to P \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, *mit* $L \subseteq R^{p_a}$ *gilt:*

  Für jedes Tupel $(\boldsymbol{p}, \boldsymbol{q}) \in P \times \mathbb{R}^{p_a}$ *mit* $f_a : (\boldsymbol{p}, \boldsymbol{q}) \mapsto (\boldsymbol{r}, n, i, o)$ *gilt* $i = id(v)$ *und* $o = od(v)$

  Bei $p_a \neq 0$ *muss gelten:*
  $$dim(G_{par}(v)) = p_a$$

3.5 Bewertung eines GP-Graphen mit algorithmischer Semantik

Bei der Bewertung eines GP-Graphen (I, O, V, E) mit einer algorithmischen Semantik (G_{func}, G_{par}) sind zwei Fälle zu unterscheiden: $|I| = 1$ und $|I| > 1$. Im ersten Fall mit nur einem Eingangsknoten handelt es sich um ein sequenzielles Programm. Im zweiten, allgemeineren Fall liegt eine paralleles Programm vor. Die Anzahl der parallelen Programmflüsse (*Threads*) entspricht der Anzahl der Eingangsknoten des Graphen.

3.5.1 Der sequenzielle Fall

Vor Beginn der ersten Funktionsauswertung wird ein Vektor **p** festgelegt, der Element der Menge ist, die durch die erste Komponente der Definitionsbereiche aller Grundfunktionen vorgegeben ist. Bei diesem Vektor handelt es sich um die Initialisierung der globalen Variablen, die von dem durch den Graphen dargestellten Algorithmus benötigt werden.

Die Auswertung des Graphen beginnt bei dem Knoten, auf den die Kante des Eingangsknotens zeigt. Zur Auswertung eines Knotens v wird immer die diesem Knoten über $G_{func}(v)$ zugewiesene Funktion berechnet. Argumente für die Funktion sind der aktuell die globalen Variablen repräsentierende Vektor und die dem Knoten über $G_{par}(v)$ zugewiesenen lokalen Parameter. Die erste Komponente des Ergebnisses wird als neue Repräsentation der globalen Variablen verwendet. Ist der Ausgangsgrad des Knotens größer eins, so entscheidet die zweite Komponente des Funktionsergebnisses, welcher, an eine herausführende Kante angrenzende Knoten als nächstes betrachtet wird.

Die Bewertung terminiert, wenn der Programmfluss einen Ausgangsknoten des GP-Graphen erreicht hat[1]. Dies tritt immer ein, wenn der Graph azyklisch ist (siehe Satz 3.1). Enthält der Graph Zyklen, so ist bei der Wahl der Grundfunktionen sicherzustellen, dass jeder mit diesen darstellbare Algorithmus terminiert.

Nach der Terminierung ist der zuletzt errechnete, die globalen Variablen repräsentierende Vektor der *Lösungsvektor*.

3.5.2 Der parallele Fall

Im Unterschied zum sequenziellen Fall werden für $|I| = k$ insgesamt k einzelne Threads parallel ausgeführt, die alle auf dieselben globalen Variablen zugreifen.

Für jeden einzelnen Thread erfolgt die Auswertung wie im sequenziellen Fall. Allerdings stehen zu jedem Zeitpunkt k Knoten zur Verfügung, die als nächstes ausgewertet werden können. Aus diesen k Knoten wird einer zufällig bestimmt. Die zugehörige Funktion wird mit dem letzten berechneten globalen Parametersatz ausgeführt, wodurch ein neuer globaler Parametersatz erzeugt wird und sich der nächste Knoten ergibt, der im Rahmen dieses Threads ausgeführt werden muss.

[1] In einer imperativen Programmiersprache entspräche jeder Ausgangsknoten somit einer *Exit*-Anweisung.

Befinden sich zu einem Zeitpunkt der Programmfluss zweier Threads im selben Knoten, so befindet sich dieser Knoten zweimal unter den als nächstes auszuwertenden Knoten. Erreicht ein Thread einen Ausgangsknoten, so verringert sich die Zahl der zu betrachtenden Programmflüsse um eins. Die Berechnung ist beendet, sobald alle Threads einen Ausgangsknoten erreicht haben. Der Lösungvektor ergibt sich aus den globalen Variablen, die der letzte Thread erzeugt hat.

Die Funktionen innerhalb eines Knotens werden als atomar betrachtet. Auf diese Weise werden Probleme vermieden, die sich durch Deadlocks bei der Vergabe von Ressourcen (also hier den globalen Variablen und lokalen Parametern der einzelnen Knoten) ergeben würden.

Die Ergebnisse einer Graphbewertung im sequenziellen Fall sind deterministisch, während ein Graph im parallelen Fall mehrere Lösungsvektoren haben kann. Diese ergeben sich in Abhängigkeit der Reihenfolge, in der die parallel auszuführenden Knoten bearbeitet werden.

Beispiel 3.4
Gegeben sei der GP-Graph (I, O, V, E) *mit*

- $I = \{i_1, i_2\}$, $O = \{o_1\}$, $V = \{v_1, v_2, v_3\}$,
- $E = \{(i_1, v_1), (i_2, v_2), (v_2, v_1), (v_1, v_3), (v_3, o_1)\}$ *sowie der Reihenfolge*
- $i_1 < i_2, (i_1, v_1) < (v_2, v_1)$.

Die Menge der Grundfunktionen $F = \{f_{inc}, f_{comb}, f_{mul}\}$ *sei gegeben mit*

- $f_{inc} : \mathbb{R} \times \varnothing \to \mathbb{R} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ *und* $f_{inc} : p \mapsto (p + 1, 1, 1, 1)$,
- $f_{comb} : \mathbb{R} \times \varnothing \to \mathbb{R} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ *und* $f_{comb} : p \mapsto (p, 1, 2, 1)$, *sowie*
- $f_{mul} : \mathbb{R} \times \varnothing \to \mathbb{R} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ *und* $f_{mul} : p \mapsto (2p, 1, 1, 1)$.

Die Funktion f_{comb} *hat keinen Einfluss auf die globalen Variablen. Mit ihr werden lediglich mehrere, von unterschiedlichen Knoten kommende Programmflüsse zu demselben Nachfolgeknoten gelenkt. In imperativen Programmiersprachen ist dies zum Beispiel mit dem Ende einer* If-Then-Else*-Struktur vergleichbar: Bis zu dieser Stelle kann der Programmfluss entweder den* If- *oder den* Else*-Zweig durchlaufen. In beiden Fällen wird jedoch danach die Anweisung ausgeführt, die der* If-Then-Else*-Struktur im Algorithmus folgt.*

Für die Semantik wird die Funktion $G_{func} : V \to F$ *mit folgenden Zuordnungen benutzt.*

$$G_{func} : v_1 \mapsto f_{comb} \quad G_{func} : v_2 \mapsto f_{inc} \quad G_{func} : v_3 \mapsto f_{mul}$$

Die Funktion G_{par} *kann in diesem Beispiel vernachlässigt werden, da alle Grundfunktionen ohne lokale Parameter auskommen.*

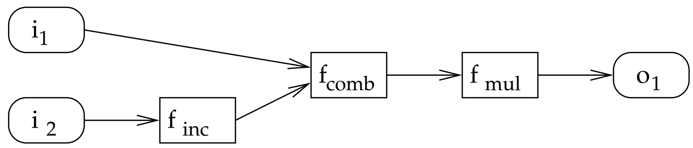


Abbildung 3.4: Der Graph zu Beispiel 3.4. Je nach Auswertungsreihenfolge kann der GP-Graph zwei verschiedene Lösungsvektoren haben.

Die Grundfunktionen benutzen als globalen Parameter eine reelle Variable, die vor der Bewertung mit dem Wert $p = 1$ initialisiert wird.

Der durch den Graphen dargestellte parallele Algorithmus kann zu zwei verschiedenen Lösungsvektoren führen. Werden zuerst die ersten beiden Funktionen vom Thread ausgeführt, der bei Knoten i_1 beginnt, so ergibt sich nach Terminierung beider Threads der Lösungsvektor (6). Wird die globale Variable durch den zweiten Thread um eins erhöht bevor der Wert durch den ersten Fluss verdoppelt wird, ergibt sich als Lösungsvektor (8). Tabelle 3.1 zeigt die beiden parallelen Teilalgorithmen sowie die beiden möglichen sequenziellen Algorithmen, die sich je nach Auswertungsreihenfolge ergeben können.

Thread 1	Thread 2	Sequenziell 1	Sequenziell 2
`p = 2p;`	`p = p + 1;`	`p = 2p;`	`p = p + 1;`
	`p = 2p;`	`p = p + 1;`	`p = 2p;`
		`p = 2p;`	`p = 2p;`

Tabelle 3.1: Die Algorithmen zum Graphen aus Abbildung 3.4. Die Spalten *Thread 1 & 2* enthalten die beiden durch je einen Thread repräsentierten Algorithmen. Die Spalten *Sequenziell 1 & 2* geben die beiden möglichen sequenziellen Algorithmen an, die sich – je nach Auswertungsreihenfolge – ergeben können.

Kapitel 4

Das GP-System *GGP*

In diesem Kapitel wird das neu entworfene, graphbasierte GP-System *GGP* vorgestellt. Grundlage des Systems ist der Algorithmus aus Abbildung 4.1. Alle weiteren Erklärungen einzelner Teilalgorithmen setzen auf diesem auf. Es handelt sich um einen steady-state Algorithmus. Als Selektion wird die Turnierselektion verwendet.

Der Algorithmus entspricht weitestgehend den üblichen GP-Systemen. Der größte Unterschied ist die Art der Verwendung des Crossover-Operators: In vielen Veröffentlichungen wird Crossover als genetischer Hauptoperator benutzt und Mutation nur als ein zusätzlicher Operator. In dem hier vorgestellten Algorithmus wird Crossover als einer von mehreren gleichberechtigten Variationsoperatoren aufgefasst, zu denen ebenfalls mehrere Arten der Mutation zählen. Crossover unterscheidet sich von allen übrigen verwendeten Operatoren dadurch, dass er aus zwei Elter-Individuen ein neues Individuum generiert. Alle anderen Operatoren erzeugen das neue Individuum aus nur einem Elter-Individuum.

Der in Abbildung 4.1 dargestellte Algorithmus benutzt die Variablen *popsize*, *toursize* und *tourwinner* sowie ein näher zu spezifizierendes Abbruchkriterium.

Mit *popsize* ist die Anzahl der Individuen in der Population gemeint. Besonderheiten wie eine Aufteilung der Population in Demes oder eine Topologie über der Population spielen im Grundalgorithmus zunächst keine Rolle.

Die Variablen *toursize* und *tourwinner* sind Parameter der Turnierselektion. Mit *toursize* ist die Anzahl der Individuen gemeint, die an einem Turnier teilnehmen. Turnier bedeutet, dass die teilnehmenden Individuen der Fitness nach geordnet werden. Die Individuen teilen sich in drei Gruppen auf: die *tourwinner* Individuen mit der besten Fitness, die *tourwinner* Individuen mit der schlechtesten Fitness und die Individuen dazwischen. Hieraus ergibt sich, dass $2 \times \textit{tourwinner} \leq \textit{toursize}$ gelten muss. Durch die Wahl des Paares (*toursize*, *tourwinner*) lässt sich der Selektionsdruck variieren. Übliche Werte sind (4,2), (4,1) oder (2,1).

Das *Abbruchkriterium* hängt stark von der Problemstellung ab. In den meisten Praxisfällen wird abgebrochen, wenn die Fitness des besten Individuums einen bestimmten Wert erreicht hat oder wenn eine vorher festgelegte Anzahl von Fitnessberechnungen erfolgt ist.

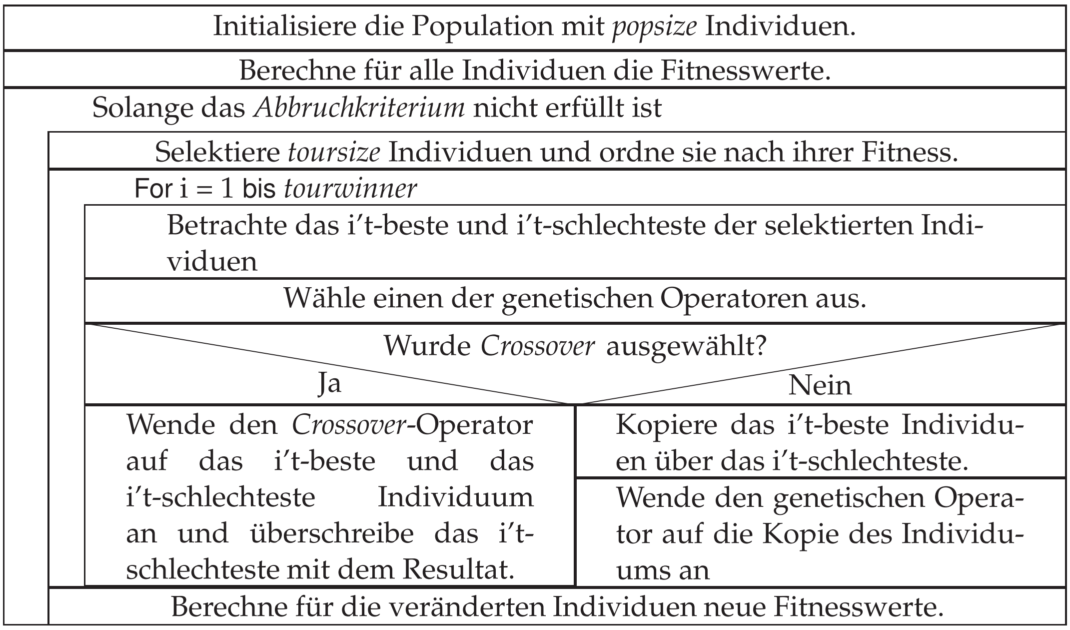


Abbildung 4.1: Grundalgorithmus, der im graphbasierten GP-System eingesetzt wird

4.1 Algorithmus zur Initialisierung der Population

Zur Initialisierung der Population wird für jedes Individuum ein GP-Graph erzeugt. Hierzu werden die vorher festgelegten Grundfunktionen verwendet. Die Initialisierung ist bei funktionalen oder algorithmischen Semantiken identisch: Zum Zeitpunkt der Initialisierung muss lediglich der Ein- und Ausgangsgrad der zu den Grundfunktionen gehörenden Knoten bekannt sein.

Der entsprechende Algorithmus muss folgende Eigenschaften besitzen:

1. Der Algorithmus soll Graphen erzeugen, die den Definitionen 3.1 und 3.3 bzw. 3.6 entsprechen. Durch die letzten beiden Definitionen findet eine Typisierung der Knoten statt.

2. Die Graphen sollen eine vorgegebene maximale Größe nicht überschreiten.

3. Die erzeugten Graphen sollen azyklisch sein.

4. Die Graphen sollen möglichst im Suchraum verteilt sein.

5. Die Graphen sollen nicht zu großen Teilen einer linearen Liste entsprechen.

Zu den Forderungen, die durch den ersten Punkt entstehen, gehört zum Beispiel, dass jedem Knoten des erzeugten Graphen eine Grundfunktion zugeordnet wird, dass die Anzahl der Ein- und Ausgänge eines Knotens zu der entsprechenden Grundfunktion passt, dass jede Kante tatsächlich von einem Knoten zu einem anderen führt und nicht *in der Luft* endet und dass jeder Knoten einen Pfad zu einem Ausgangsknoten besitzt.

Durch den zweiten Punkt wird es möglich, den Einfluss einer vorgegebenen maximalen Graphgröße auf die Performance des GP-Systems zu untersuchen.

In der Definition des GP-Graphen wird nicht verlangt, dass der Graph azyklisch sein muss. Allerdings ist dies für viele Problemstellungen wünschenswert. Aus diesem Grund werden nur azyklische Graphen erzeugt, die während der Evolutionszyklen mittels dafür vorgesehener genetischer Operatoren in zyklische Graphen verwandelt werden können.

Die Forderung des vierten Punktes lässt sich quantitativ nicht genauer beschreiben. Hierzu wäre erst die Definition eines Abstandbegriffs nötig. In diesem Fall könnte an Stelle von Punkt vier gefordert werden, dass alle Inidividuen untereinander einen Mindestabstand nicht unterschreiten. Informell lässt sich sagen, dass die Graphen der Population sowohl semantisch als auch syntaktisch möglichst unterschiedlich sein sollen.

Die letzte Forderung ist Ergebnis von Voruntersuchungen mit einem einfacheren Initialisierungsalgorithmus, bei dessen Verwendung die Performance des genetischen Grundalgorithmus deutlich schlechter war, als bei dem hier vorgestellten. Die Forderung ergibt sich auch direkt als Konsequenz des vierten Punktes.

Abbildung 4.2 zeigt den Algorithmus, der zum Initialisieren eines Individuums benutzt wird und somit bei einer Populationsgröße von *popsize* für jedes Individuum einmal aufgerufen wird.

Die Größe des zu erzeugenden Graphen wird im Unterprogramm *′Lege die Anzahl der inneren ...′* festgelegt. Die Größe wird mittels Zufallszahlengenerator gleichverteilt gewählt, wobei als untere Grenze die halbe Anzahl der noch bis *maxsize* fehlenden Knoten benutzt wird. Als obere Grenze dient der Wert *maxsize*. Hierbei wird berücksichtigt, dass der Wert *maxsize* sich auf die Gesamtgröße des Graphen inklusive Eingangs- und Ausgangsknoten bezieht. Wenn die Anzahl der Knoten, die maximal in V sein dürfen, es zulässt, wird zusätzlich gefordert, dass *KnotenInV* mindestens die Differenz zwischen Eingangs- und Ausgangsknoten umfassen soll.

Mit dem Unterprogramm *„Wähle mit Hilfe der Werte...“* wird jedem neuen Knoten v eine Funktion aus der Menge der Grundfunktionen zugeordnet und somit auch festgelegt, wie viele Eingänge $id(v)$ und Ausgänge $od(v)$ er besitzt. Zuerst wird festgestellt, welche Grundfunktionen für den nächsten Knoten v in Frage kommen. Hierzu werden folgende Bedingungen überprüft:

- Wurde eine (i, j)-Grundfunktion gewählt, muss die Menge *ToDo* mindestens i Elemente enthalten.

- Existiert eine Ergänzung des bisher gefundenen Graphen mit den Knoten V und dem neuen Knoten v zu einem Graphen mit *KnotenInV* Knoten, wenn nur Knoten mit Eingangs-/Ausgangsgrad (1,2), (2,1) und (1,1) hinzugefügt werden?

Gegeben:

$\mathsf{I} = \{i_1, \ldots, i_n\}$	{Menge der Eingangsknoten}
$\mathsf{O} = \{o_1, \ldots, o_m\}$	{Menge der Ausgangsknoten}
maxsize	{die maximale Graphgröße}
F	{Menge der Grundfunktionen}

Lege die Anzahl der inneren Knoten KnotenInV in Abhängigkeit von I, O und maxsize fest.

Erzeuge eine Tupelmenge ToDo $:= \{(i_1, 1), \ldots, (i_n, 1)\}$.

V $:= \varnothing$, E $:= \varnothing$, fehler $:= 0$

Solange $|\mathsf{V}| <$ KnotenInV und fehler < 1000

- Wähle mit Hilfe der Werte |ToDo|, KnotenInV und |V| eine Funktion f $\in$ F aus
- Wurde eine passende Funktion gefunden?
  - Ja
    - Erzeugen einen neuen Knoten v mit V $:=$ V $+$ v und G_{func} : v $\mapsto f$
    - For j $:= 1$ To id(v)
      - Wähle zufällig ein Tupel (a, b) aus ToDo
      - Erzeuge neue Kante e = (a, v) mit E $:=$ E $+$ e.
    - Die Reihenfolge der Kanten, die nach v führen, entspricht der Reihenfolge der Definition in der Schleife
    - For j $:= 1$ To od(v)
      - ToDo = ToDo + (v, j)
    - Die Reihenfolge der Kanten, die aus v herausführen, entspricht den Zahlen in den entsprechenden ToDo-Tupeln.
  - Nein
    - ToDo $:= \{(i_1, 1), \ldots, (i_n, 1)\}$
    - V $:= \varnothing$, E $:= \varnothing$
    - fehler $:=$ fehler $+ 1$
    - $\varnothing$

Erzeuge für jedes der m in ToDo verblieben Tupel eine Kante vom entsprechenden Knoten zu einem der Ausgangsknoten aus O

Erzeuge G_{par}, wobei jedem Knoten so viele $(0, 4)$-normalverteilte Werte zugeordnet werden, wie die zugehörige Funktion F benötigt.

Abbildung 4.2: Grundalgorithmus zur Erzeugung des GP-Graphen für ein Individuum

Wenn die erste Bedingung nicht erfüllt wäre, könnten nicht genug Kanten von der bisherigen Menge $I \cup V$ zum neuen Knoten v erzeugt werden. Somit würde die Definition der Syntax eines GP-Graphen verletzt.

Die zweite Bedingung stellt zusätzlich sicher, dass der *Nein*-Zweig in der *„Wurde eine passende Funktion gefunden?"*-Abfrage des folgenden Schleifendurchlaufs normalerweise nicht durchlaufen wird und somit iterativ überhaupt nicht aufgerufen wird. Dies gilt für alle Mengen von Grundfunktionen, die mindestens je eine (1,1)-, (1,2)- und (2,1)-Grundfunktion enthalten. Dies ist bei allen in dieser Arbeit verwendeten Grundfunktionsmengen gegeben. Würde auf diese Regel verzichtet, könnte es passieren, dass nur noch ein Knoten in den Graphen eingefügt werden soll, dieser aber zehn Kanten mit einem Ausgang verbinden müsste. Wenn keine entsprechende Grundfunktion vorhanden ist, würde der komplette Graph neu aufgebaut.

Wurden alle in Frage kommenden Grundfunktionen ermittelt, wird eine von diesen ausgewählt. Hierbei wird, je nach Anzahl der bereits in V befindlichen Knoten, mit einer unterschiedlichen Wahrscheinlichkeit eine Funktion gewählt, für deren zugeordneten Knoten v die Ungleichung $id(v) < od(v)$ erfüllt ist, deren Knoten also einen höheren Ausgangs- als Eingangsgrad besitzen. Diese Wahrscheinlichkeit ist in Abbildung 4.3 über die Anzahl der bereits in V befindlichen Knoten aufgetragen.

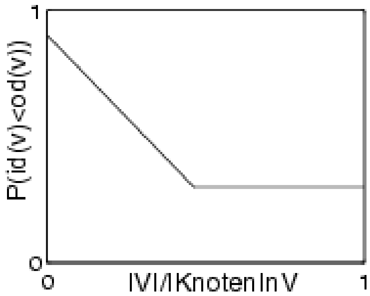


Abbildung 4.3: Die Wahrscheinlichkeiten, dass eine Funktion ausgewählt wird, die einen höheren Ausgangs- als Eingangsgrad hat

Durch eine hohe Anfangswahrscheinlichkeit zu Gunsten von Funktionen, die den Graphen verbreitern, wird erreicht, dass der fünfte Punkt der eingangs aufgestellten Forderungen erfüllt wird. Bei den Grundfunktionsmengen, die in dieser Arbeit benutzt werden, ist die Zahl der (i, j)-Grundfunktionen mit $i > j$ wesentlich größer als die mit einem umgekehrten Verhältnis. Würde aus den möglichen Grundfunktionen gleichverteilt eine beliebige ausgewählt, würde die Größe der Menge *ToDo* während der Initialisierung eines GP-Graphen meistens gleich eins sein. Dies führt wiederum zu Graphen, die zu großen Teilen aus einer linearen Kette von Knoten bestehen.

Die Wahrscheinlichkeiten aus Abbildung 4.3 sind nur qualitativ zu sehen. So fällt die Wahrscheinlichkeit zum Beispiel automatisch auf 0, wenn kein Knoten mit einer entsprechenden Grundfunktion in den Graphen eingefügt werden kann. Andererseits wird immer, wenn nicht explizit eine (i, j)-Grundfunktion mit $i < j$ verlangt wird, gleichverteilt aus allen möglichen Grundfunktionen ausgewählt. Somit steigt die Wahrscheinlichkeit je nach Grundfunktionsmenge wiederum etwas an.

Durch das Unterprogramm *„Erzeuge G_{par}, wobei jedem Knoten..."* werden die lokalen Parameter zu den Knoten erzeugt, die diese benötigen. Hierbei hat es sich gezeigt, dass es ausreicht, (0,4)-normalverteilte Werte zu benutzen, da GP selbst in der Lage ist, andere Werte durch entsprechende Rechenoperationen innerhalb des Graphen zu erzeugen.

4.2 Genetische Operatoren

Ein genetischer Operator ist eine Funktion, die Veränderungen an einem GP-Graphen vornimmt. Während GP-Systeme nach KOZA hauptsächlich Crossover als Variationsoperator einsetzen, stehen in dem entwickelten System acht verschiedene Operatoren zur Verfügung. Crossover hat hierbei keine Sonderstellung, sondern wird wie die übrigen Mutationsoperatoren behandelt.

Die Operatoren verändern den Graphen auf verschiedene Weisen. Bei Knotenmutationen bleibt der GP-Graph erhalten und nur die Semantik eines Knotens ändert sich. Bei knotenorientierten Operatoren wird lediglich ein Knoten verändert. Bei pfadorientierten Operatoren wird ein Teilgraph verändert, der bestimmten Bedingungen genügt. Beim Crossover werden Teilgraphen zweier GP-Graphen zu einem neuen zusammengefügt.

Alle hier vorgestellten Algorithmen benötigen als Eingabe einen GP-Graphen mit Semantik. Ausgabe ist ein veränderter Graph mit einer neuen Semantik. Ausnahme bildet der Crossover-Algorithmus, bei dem der neue GP-Graph aus zwei Graphen zusammengesetzt wird.

Im Folgenden werden die einzelnen genetischen Operatoren vorgestellt. Zuvor folgen kurze Bemerkungen zur Kantenreihenfolge und weitere Begriffsdefinitionen.

4.2.1 Vorüberlegungen

Unter einem *GP-Pfad* wird im Zusammenhang der pfadorientierten Operatoren eine durch Kanten verbundene Knotenmenge bezeichnet, bei der alle Knoten im Inneren (1,1)-Knoten sind. Dies ist eine Einschränkung gegenüber dem in Abschnitt 2.3.1 umgangssprachlich eingeführten Pfadbegriff, der in Definition 4.2 genauer beschrieben wird.

Definition 4.1
Ein Teilgraph $(\{v_1, \ldots, v_n\}, \{e_0, \ldots, e_n\})$ eines GP-Graphen (V, E) heißt GP-Pfad, wenn folgende Bedingungen erfüllt sind:

- *Für alle Knoten $v_i \in \{v_1, \ldots, v_n\}$ gilt:* $id(v_i) = od(v_i) = 1$.
- *Für jede Kante $e_i \in \{e_1, \ldots, e_{n-1}\}$ gilt:* $e_i = (v_i, v_{i+1})$

- *Sei* $e_0 = (v_{in}, v_1)$ *mit* $v_{in} \notin \{v_1, \ldots, v_n\}$. *Für* v_{in} *gilt*

$$id(v_{in}) = 0 \wedge od(v_{in}) = 1 \text{ oder } od(v_{in}) > 1).$$

- *Sei* $e_n = (v_n, v_{out})$ *mit* $v_{out} \notin \{v_1, \ldots, v_n\}$. *Für* v_{out} *gilt* $id(v_{out}) > 1$.

Die Knoten v_{in} *und* v_{out} *heißen Pfad-Anfangsknoten bzw. Pfad-Endknoten. Die Länge des Pfades sei die Anzahl seiner Kanten, also* $n + 1$.

Definition 4.2
Ein Teilgraph $(\{v_{in}, v_1, \ldots, v_n, v_{out}\}, \{e_0, \ldots, e_n\})$ *eines Graphen* (V, E) *heißt Pfad, wenn folgende Bedingungen erfüllt sind:*

- *Für jede Kante* $e_i \in \{e_1, \ldots, e_{n-1}\}$ *gilt:* $\quad e_i = (v_i, v_{i+1})$
- *Für* e_0 *gilt:* $e_0 = (v_{in}, v_1)$ *mit* $v_{in} \in V$
- *Für* e_n *gilt:* $e_n = (v_n, v_{out})$ *mit* $v_{out} \in V$

Die Knoten v_{in} *und* v_{out} *heißen Anfangsknoten bzw. Endknoten des Pfades. Die Länge des Pfades sei die Anzahl ihrer Kanten, also* $n + 1$.

Operatoren, die Syntaxänderungen bei einem GP-Graphen vornehmen, müssen sicherstellen, dass nach der Änderung wieder Reihenfolgen auf allen Eingangs- und Ausgangskanten definiert sind (Punkt 6 von Definition 3.1).

Bei der Beschreibung aller Operatoren, die neue Knoten in einen Graphen einfügen, wird im Folgenden jeweils die Kantenreihenfolge am neuen Knoten genannt. Wird von einem Knoten eine Kante entfernt und durch eine andere ersetzt, so nimmt die neue Kante in der Kantenreihenfolge die Stelle der alten ein. In den Beschreibungen der Operatoren wird dieser Fall als Standardvorgehensweise immer implizit angenommen und nicht weiter ausgeführt.

Nach den Semantik-Regeln für GP-Graphen ist es nicht möglich, dass ein Knoten ausschließlich eine Kante hinzugefügt bekommt oder eine verliert. Aus diesem Grund sind mit den beiden, im vorigen Absatz beschriebenen Fällen, alle Graphänderungen abgedeckt, die genauere Betrachtungen der Kantenreihenfolge erforderlich machen würden.

4.2.2 Knoten mutieren

Bei dem genetischen Operator *Knoten mutieren* handelt es um eine Variation der Semantik zu einem Knoten des GP-Graphen. Die Syntax des Graphen bleibt erhalten, es wird nur die Grundfunktion ausgetauscht, die dem Knoten zugeordnet ist.

Abbildung 4.4 zeigt den im GP-System implementierten Algorithmus. Es kann passieren, dass für einen Knoten bei der Mutation dieselbe Grundfunktion gewählt wird, die dem

[1] Alle Knoten haben bei der Auswahl die gleiche Wahrscheinlichkeit

Wähle (gleichverteilt)[1]einen Knoten $v \in V$ aus.
Bestimme alle $(id(v), od(v))$-Grundfunktionen.
Wähle (gleichverteilt) eine dieser Grundfunktionen aus.
Ordne diese Funktion f dem Knoten v zu: $G_{func} : v \mapsto f$.
Lege, wenn nötig, neue Werte für $G_{par}(v)$ fest.

Abbildung 4.4: Der genetische Operator *Knoten mutieren* verändert die Semantik eines Knotens.

Knoten bereits vorher zugeordnet war. In diesem Fall unterscheidet sich der neue Graph nur durch eventuell vorhandene lokale Parameter an diesem Knoten, die neu belegt wurden. Eine Neubelegung der lokalen Parameter findet mittels (0,4)-normalverteilter Zufallsvariablen statt.

4.2.3 Knoten einfügen

Der Operator *Knoten einfügen* trennt eine der Kanten des Graphen auf und fügt einen Knoten mit Ein- und Ausgangsgrad eins ein. Der Operator kann demnach nur benutzt werden, wenn mindestens eine (1,1)-Grundfunktion existiert.

Voraussetzung: $\{$Es existiert mindestens eine $(1,1)$-Grundfunktion.$\}$ $\{$Der GP-Graph enthält mindestens einen Knoten weniger, als die durch maxsize vorgegebene maximale Größe.$\}$
Wähle (gleichverteilt) eine Kante (v_i, v_j) des GP-Graphen.
Wähle (gleichverteilt) eine $(1,1)$-Grundfunktion f_k.
Ergänze die Knotenmenge V um einen neuen Knoten v.
Ergänze die Kantenmenge E um die beiden Kanten (v_i, v) und (v, v_j).
Lösche die Kante (v_i, v_j) aus E.
$G_{func} : v \mapsto f_k$
Lege, wenn nötig, neue Werte für $G_{par}(v)$ fest.

Abbildung 4.5: Der genetische Operator *Knoten einfügen* fügt einen (1,1)-Knoten v in den GP-Graphen ein.

4.2.4 Knoten löschen

Dieser Operator löscht einen (1,1)-Knoten aus dem Graphen. Ein einzelner Knoten v, bei dem sich Eingangsgrad $id(v)$ und Ausgangsgrad $od(v)$ unterscheiden, kann nicht gelöscht werden. Es könnten $\min(id(v), od(v))$ Paare aus Eingangs- und Ausgangskanten zu jeweils einer Kante zusammengefasst werden. Es würden jedoch $|id(v) - od(v)|$ Kanten unverbindbar bleiben, so dass die Definition eines GP-Graphen verletzt wäre. Der Operator kann deshalb nur angewendet werden, wenn im GP-Graph mindestens ein (1,1)-Knoten existiert. Der Algorithmus ist in Abbildung 4.6 dargestellt.

Voraussetzung: {Es existiert mindestens ein $(1,1)$-Knoten im Graphen.}
Wähle (gleichverteilt) einen $(1,1)$-Knoten v des Graphen, die angrenzenden Kanten seien (v_i, v) und (v, v_j).
Lösche v aus der Knotenmenge V. Die Definitionsbereiche von G_{func} und G_{par} werden um den Knoten v reduziert.
Lösche die Kanten (v_i, v) und (v, v_j) aus der Kantenmenge E.
Füge die Kante (v_i, v_j) zur Kantenmenge E hinzu.

Abbildung 4.6: Der genetische Operator *Knoten löschen* löscht einen $(1,1)$-Knoten v aus dem GP-Graphen.

4.2.5 Knoten verschieben

Es wird ein $(1,1)$-Knoten entlang einer seiner Kanten innerhalb des Graphen verschoben. Abbildung 4.7 verdeutlicht dies, Abbildung 4.8 enthält den Algorithmus.

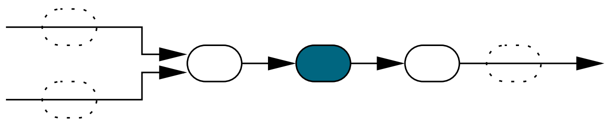


Abbildung 4.7: *Knoten verschieben:* Der markierte Knoten kann – vorbei an einem der beiden benachbarten Knoten – an eine der gestrichelten Positionen verschoben werden.

Aus allen Eingangskanten des Vorgängerknotens und Ausgangskanten des nachfolgenden wird eine Kante ausgewählt. Der zu verschiebende Knoten wird zwischen die beiden an diese Kante angrenzenden Knoten gesetzt.

4.2.6 Pfad einfügen

Es ist nicht möglich, in einem GP-Graphen eine Anzahl von Knoten einzufügen, wenn die Summe ihrer Eingangsgrade ungleich der Summe ihrer Ausgangsgrade ist, da in diesem Fall Definition 3.1 verletzt würde. Aus diesem Grund ist es auch nicht möglich, einen einzelnen (i,j)-Knoten mit $i \neq j$ einzufügen. Da jedoch jeder GP-Graph im Suchraum durch Anwendung von genetischen Operatoren erreichbar sein soll, existieren Funktionen, die Knotenpaare einfügen oder löschen. Bei diesen Knotenpaaren wird darauf geachtet, dass die Summe der Eingangs- und Ausgangsgrade aller Knoten gleich ist.

Voraussetzung:
{Es existiert mindestens ein $(1,1)$-Knoten im Graph.}

Wähle (gleichverteilt) einen $(1,1)$-Knoten v des Graphen. Die angrenzenden Knoten seien v_i und v_o, die dazwischenliegenden Kanten somit (v_i, v) und (v, v_o).

Erzeuge die Menge K mit allen Kanten, die zu v_i führen oder v_o verlassen: K = $\{(a, v_i)|(a, v_i) \in E\} \cup \{(v_o, b)|(v_o, b) \in E\}$

Wähle (gleichverteilt) eine Kante $(a, b) \in$ K.

b $=$ v_i ?

Ja	Nein
$\mathsf{E} = \mathsf{E} + (\mathsf{a}, \mathsf{v}) + (\mathsf{v}, v_i)$	$\mathsf{E} = \mathsf{E} + (v_o, \mathsf{v}) + (\mathsf{v}, \mathsf{b})$

$\mathsf{E} = \mathsf{E} - (v_i, \mathsf{v}) - (\mathsf{v}, v_o)$

Abbildung 4.8: Der genetische Operator *Knoten verschieben* verändert die Position eines $(1,1)$-Knotens innerhalb des Graphen.

Das Einfügen der neuen Knoten geschieht durch Hinzufügen eines neuen GP-Pfades der Länge eins, wodurch zwei neue Knoten mit gemeinsamer neuer Kante eingefügt werden. Dazu werden zwei Kanten des Graphen bestimmt. In die erste wird ein $(1,2)$-Knoten und in die zweite ein $(2,1)$-Knoten eingefügt. Der freie Ausgang des ersten Knotens wird mit dem freien Eingang des zweiten Knoten durch eine neue Kante verbunden. Hierbei handelt es sich somit um einen neuen GP-Pfad der Länge eins. Alternativ kann statt des $(1,2)$-Knoten ein $(0,1)$-Knoten verwendet werden. In diesem Fall wird insgesamt nur eine alte Kante aufgespalten.

Es muss sichergestellt werden, dass azyklische GP-Graphen auch bei der Anwendung des Operators *Pfad einfügen* azyklisch bleiben. Aus diesem Grund muss bei der Auswahl der beiden Kanten darauf geachtet werden, dass kein Kantenzug innerhalb des Graphen von der zweiten aufzuspaltenden Kante zur ersten führt, da auf diese Weise durch das Einfügen des neuen GP-Pfades ein Zyklus entstehen würde.

Die Reihenfolge der vom (1,2)-Knoten abgehenden Kanten und die der beiden Kanten, die am neuen (2,1)-Knoten ankommen, wird zufällig festgelegt.

Abbildung 4.9 zeigt den Algorithmus zum Operator. Ein auf diese Weise eingefügter GP-Pfad kann in folgenden Evolutionszyklen mit dem Operator *Knoten einfügen* um weitere Knoten verlängert werden.

4.2.7 Zyklus

Der Operator *Zyklus* entspricht dem Operator *Pfad einfügen* mit dem Unterschied, dass hier ein Zyklus in den Graph eingefügt wird. Zwei Unterschiede, ergeben sich zum Algorithmus aus Abbildung 4.9: Zum einen wird die Zielkante aus der Menge *Before* ausgewählt, also der

<table>
<tr><td colspan="2">Voraussetzung:
{Es existiert mindestens eine $(2,1)$-Grundfunktion und entweder mindestens eine $(1,2)$- oder $(0,1)$-Grundfunktion}
{Der GP-Graph enthält mindestens zwei Knoten weniger als die durch maxsize vorgegebene maximale Größe.}</td></tr>
<tr><td colspan="2">Wähle (gleichverteilt) eine Startkante $(s_1, s_2) \in \mathsf{E}$ aus.</td></tr>
<tr><td colspan="2">Erstelle mittels Tiefendurchlaufs die Menge Before mit allen Kanten, die Teil eines Pfades von einem (0,1)-Knoten (inklusive der Eingangsknoten des GP-Graphen) nach s_1 sind.</td></tr>
<tr><td colspan="2">Bestimme aus der Menge $\mathsf{E} \setminus \mathsf{Before}$ eine Zielkante (z_1, z_2).</td></tr>
<tr><td colspan="2">Bestimme aus der Menge aller $(0,1)$- und $(1,2)$-Grundfunktionen eine Funktion f_v</td></tr>
<tr><td colspan="2">Füge einen neuen Knoten v zur Menge V hinzu, $G_{func} : \mathsf{v} \mapsto f_v$.</td></tr>
<tr><td colspan="2">Bestimme aus der Menge aller $(2,1)$-Grundfunktionen eine Funktion f_w</td></tr>
<tr><td colspan="2">Füge einen neuen (Ziel-)Knoten w zur Menge V hinzu, $G_{func} : \mathsf{w} \mapsto f_w$.</td></tr>
<tr><td colspan="2">$\mathsf{E} = \mathsf{E} + (z_1, \mathsf{w}) + (\mathsf{w}, z_2) - (z_1, z_2) + (v, w)$,</td></tr>
<tr><td colspan="2">Lege die Reihenfolge der Kanten (z_1, w) und (v, w) fest.</td></tr>
<tr><td colspan="2">Ist f eine $(1,2)$-Grundfunktion?</td></tr>
<tr><td>Ja</td><td>Nein</td></tr>
<tr><td>$\mathsf{E} = \mathsf{E} + (s_1, \mathsf{v}) + (\mathsf{v}, s_2) - (s_1, s_2)$
Lege die Reihenfolge der Kanten (v, s_2) und (v, w) fest.</td><td>$\varnothing$</td></tr>
<tr><td colspan="2">Lege, wenn nötig, neue Werte für $G_{par}(v)$ und $G_{par}(w)$ fest.</td></tr>
</table>

Abbildung 4.9: Der genetische Operator *Pfad einfügen* fügt einen neuen Teilgraphen aus zwei Knoten und einer Kante in den GP-Graphen ein.

Menge aller Kanten, von denen es einen Pfad in Richtung der Startkante gibt. Zum anderen darf keine (0,1)-Grundfunktion benutzt werden.

4.2.8 Pfad löschen

Der genetische Operator *Pfad löschen* ist das Gegenstück zu den beiden Operatoren *Pfad einfügen* und *Zyklus*. Es ist möglich GP-Pfade beliebier Länge zu löschen. Beim Löschen wird der Pfad-Anfangsknoten durch einen Knoten ersetzt, dessen Ausgangsgrad um eins kleiner als der des bisherigen Knotens ist. Für den Pfad-Endknoten wird entsprechend ein Knoten mit um eins verringertem Eingangsgrad verwendet. Würde beim Ersetzen ein (1,1)-Knoten eingefügt, so kann dieser alternativ auch aus dem Graphen entfernt werden und durch eine Kante ersetzt werden. Liegt beim Pfadanfang ein (0,1)-Knoten vor, so wird dieser gelöscht.

Der Operator arbeitet in drei Schritten. Zuerst werden Kanten bestimmt, die auf keinen Fall gelöscht werden dürfen, weil durch das Entfernen eine Bedingung aus Definition 3.1 verletzt würde. In einem zweiten Schritt werden GP-Pfade des Graphen bestimmt, die diese Kanten nicht enthalten. Abschließend wird einer dieser GP-Pfade ausgewählt und gelöscht. Im Folgenden wird der im System verwendete Algorithmus vorgestellt. Bei der Entwicklung wurde besonderer Wert auf eine effiziente Implementierung gelegt.

4.2.8.1 Kennzeichnung nicht-löschbarer Kanten

Der Graph, der nach dem Löschen eines GP-Pfades entsteht, muss die Definition eines GP-Graphen 3.1 erfüllen. Die Punkte 1 bis 6 werden dabei durch den Teilalgorithmus zur Auswahl der GP-Pfade sichergestellt. Die Kennzeichnung nicht-löschbarer Kanten stellt sicher, das auch die Punkte 7 und 8 erfüllt werden. Ein GP-Pfad wird nur gelöscht, wenn alle seine Kanten löschbar sind. Aus diesem Grund brauchen nur die Anfangs- und Endkanten eines potenziellen GP-Pfades auf Löschbarkeit untersucht werden: Alle weiteren Kanten können nicht für eine Verletzung der Punkte 7 und 8 sorgen.

Handelt es sich bei dem zu untersuchenden GP-Graphen um einen zyklenfreien Graphen, so darf jeder GP-Pfad dieses Graphen gelöscht werden. Der neue GP-Graph bleibt zyklenfrei und alle verbleibenden Knoten haben Verbindungen von einem Eingangsknoten und zu einem Ausgangknoten. Dies ergibt sich anschaulich daraus, dass der für den GP-Pfad-Endknoten eingesetzte neue Knoten sowohl Verbindungen zu einem Eingang als auch zu einem Ausgang haben muss. Die gleiche, im Folgenden dargelegte Argumentation gilt auch für den Knoten, der den alten GP-Pfad-Anfangsknoten ersetzt:

Der neue Knoten v, der den alten GP-Pfad-Endknoten ersetzt, hat weiterhin eine Verbindung zu denselben Ausgängen, die der ehemalige GP-Pfad-Endknoten v_e hatte, da die gelöschten Kanten im Graphen vor v_e lagen und der Graph zyklenfrei ist. Der Knoten v hat weiterhin mindestens eine Verbindung zu einem Eingang, da v_e nach Definition 4.1 mindestens eine weitere Verbindung zu einem Eingangsknoten haben musste. Dieser Pfad kann ebenfalls nach Definition 4.1 durch das Löschen der Kanten des GP-Pfades nicht unterbrochen worden sein, da kein Knoten des GP-Pfades einen Ausgangsgrad größer als eins haben darf und die alte Verbindung zwischen v_e und dem Eingang den zu löschenden GP-Pfad somit auch nicht kreuzen kann.

Die Punkte 7 und 8 der Definition 3.1 können beim Löschen eines GP-Pfades nur in einem Fall verletzt werden: Der Graph enthält einen Zyklus in den genau eine Kante hineinführt oder den genau eine Kante verlässt und diese Kante ist Teil des zu löschenden Pfades (Abbildung 4.10).

Der Teilalgorithmus zur Kennzeichnung nicht-löschbarer Kanten muss demnach jeweils die Kante markieren, die in einen Zyklus hineinführt sowie diejenige, die aus ihm herausführt. Hierbei ist allerdings zu beachten, dass aus einem Zyklus Kanten herausführen können, die lediglich der Beginn eines GP-Pfades sind, der wieder in denselben Zyklus hineinführt,

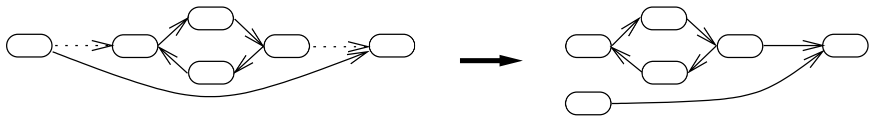


Abbildung 4.10: Die beide gestrichelten Kanten des linken Graphen dürfen nicht entfernt werden, da sonst der Zyklus entweder keinen Eingang oder keinen Ausgang hat.

so dass sich geschachtelte Zyklen ergeben. Würde der Algorithmus diese Kante als nichtlöschbar markieren und die eigentliche Kante, die vom Zyklus aus einen Pfad zu einem der Ausgangsknoten besitzt und nach Definition3.1 existiert, als Pfadbeginn löschen, würde der erzeugte Graph Punkt 7 der GP-Graph-Definition verletzen (Abb. 4.11).

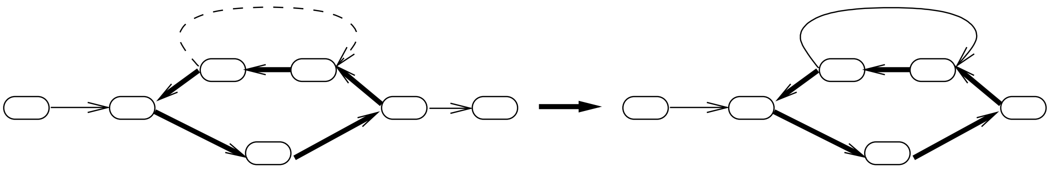


Abbildung 4.11: Die gestrichelte Kante verlässt zwar den Zyklus, führt jedoch direkt wieder in diesen hinein. Wird sie, statt der rechten Kante als nicht-löschbar markiert,so verletzt der neue Graph bei einer Entfernung der rechten Kante den Punkt 7 der GP-Graph-Definition 3.1.

Zur effizienten Markierung der Knoten wird, beginnend bei jedem (0,1)-Knoten, per Rekursion ein Tiefendurchlauf durch den Graphen durchgeführt. Beim Betreten eines Knotens wird die Kante markiert, über die der Knoten betreten wurde. Wurde der Knoten während des Durchlaufs bereits betreten, erhält die aufrufenden Rekursionsebene den Rückgabewert *Zyklus*. Diese Meldung bedeutet für die aufrufende Rekursionsebene, dass über diese Kante ausschließlich Knoten eines Zyklus erreicht werden können.

Ist dies nicht der Fall, wird für alle herausführenden Kanten nacheinander in der nächsten Rekursionsebene abgefragt, ob die Rückgabewerte *Zyklus* oder *kein Zyklus* lauten. Wenn alle Ergebnisse vorliegen, können folgende unterschiedliche Situationen eingetreten sein.

- Alle nachfolgenden Kanten führen in keinen Zyklus.

  Alle Kanten besitzen somit einen Pfad zu einem Ausgangsknoten. Wenn eine dieser Kanten der Anfang eines GP-Pfades wäre, würde somit bei der Löschung dieses Pfades auf jeden Fall eine Verbindung zu einem Ausgang über eine der anderen Kanten erhalten bleiben. Der aufrufenden Rekursionsebene wird als Rückgabewert ebenfalls *kein Zyklus* übergeben.

- Eine oder mehrere der Kanten liefern den Wert *Zyklus*, aber genau eine Kante liefert den Wert *kein Zyklus*.

  In diesem Fall ist der aktuelle Knoten Teil von mindestens einem Zyklus, der aber gleichzeitig eine Kante besitzt, die eine Verbindung zu einem Ausgangsknoten hat.

Diese Kante wird als nicht-löschbar markiert. Der Rückgabewert an die aufrufende Rekursionsebene lautet *kein Zyklus*.

- Mehrere nachfolgende Kanten führen in keinen Zyklus.

  Dieser Fall entspricht dem ersten. Selbst wenn der aktuelle Knoten Teil eines Zyklus ist, können alle angrenzenden Kanten gelöscht werden, da immer mindestens eine erhalten bliebt, die von diesem Knoten aus dem Zyklus herausführt. Rückgabewert ist ebenfalls *kein Zyklus*.

- Alle Kanten liefern den Rückgabewert *Zyklus*.

  Dieser Fall ist der komplizierteste, der von dem hier verwendeten Algorithmus aus Effizienzgründen nur als übervorsichtige Schätzung behandelt wird. Der Rückgabewert *Zyklus* sagt aus, dass eine Kante Teil eines Zyklus ist. Wenn allerdings mehrere Zyklen geschachtelt sind, werden diese vom Algorithmus nicht als einzelne Zyklen erkannt, sondern als ein Zyklenteilgraph betrachtet. Es wird zwar an jeder Stelle, wo ein Knoten zum wiederholten Mal betreten wird, die Eingangskante des ersten Betretens als nicht-löschbar markiert, für den kompletten Zyklenteilgraph wird jedoch nur eine verlassende Kante als nicht-löschbar markiert.

  Gehen von einem Knoten zwei Kanten aus, die beide als *Zyklus* markiert sind, ist für die einzelne Kante nicht klar, ob sie eine Verbindung zu der den Zyklus verlassenden Kante besitzt. Dieses Problem wird in Abbildung 4.12 verdeutlicht. Aus Effizienzgründen werden in diesem Fall beide Kanten als nicht-löschbar markiert.

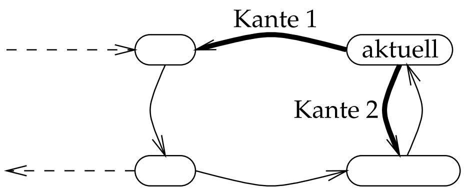


Abbildung 4.12: Die Rückgabewerte für die beiden Kanten 1 und 2 am Knoten *aktuell* sind jeweils *Zyklus*. Während Kante 2 gelöscht werden könnte, würde das Entfernen von Kante 1 dazu führen, dass vom Knoten *aktuell* aus kein Pfad zu einem Ausgangsknoten führen würde und der Graph somit Punkt 7 der GP-Graph-Definition verletzen würde.

Sollten im genannten Fall nur die Kanten als nicht-löschbar markiert werden, die die Verbindung zur den Zyklus verlassenden Kante sicherstellen, so ließe sich dies durch eine weitere Tiefensuche erreichen. Diese würde am aktuellen Knoten ansetzten und überprüfen, welche der Ausgangskanten des Knotens eine Verbindung zu einem Ausgangsknoten haben, die nicht noch einmal am aktuellen Knoten vorbeiführt. Dies würde allerdings zu einer Verschlechterung des Laufzeitverhaltens führen, das so nicht mehr linear zur Anzahl der Kanten wäre.

Wenn alle Rückgabewerte an einem Knoten *Zyklus* sind, werden deshalb alle herausführenden Kanten als nicht-löschbar markiert und ebenfalls *Zyklus* als Rückgabewert geliefert.

Der Tiefendurchlauf terminiert, wenn alle Knoten, die vom ursprünglichen (0,1)-Knoten erreichbar sind, einmal betrachtet wurden. Da ein GP-Graph mehrere Eingangsknoten bzw. auch weitere (0,1)-Knoten enthalten kann, muss für diese Knoten jeweils ein weiterer Tiefendurchlauf durchgeführt werden. Wird während einer dieser Tiefendurchläufe ein Knoten erreicht, der in einem der vorigen Durchläufe bereits betrachtet wurde, so liefert die Rekursion den Wert *kein Zyklus* zurück, da von diesem Knoten aus kein Pfad zu einem im aktuellen Durchgang bereits betrachteten Knoten existieren kann.

Der in Abbildung 4.13 vorgestellte Algorithmus markiert somit alle Kanten an einem Pfadanfang, die nicht gelöscht werden dürfen sowie alle Kanten an einem Pfadende, die ebenfalls nicht entfernt werden dürfen. Hierzu betrachtet er jede Kante des Graphen genau einmal, die Laufzeit ist somit linear zur Anzahl der Kanten.

4.2.8.2 Löschbare GP-Pfade bestimmen

Es können im Graph alle GP-Pfade gelöscht werden, bei denen weder die erste noch die letzte Kante als *nicht-löschbar* markiert ist.

Hierzu werden nacheinander alle inneren Knoten des GP-Graphen betrachtet. Handelt es sich um einen (0,1)-Knoten oder um einen Knoten mit einem Ausgangsgrad größer eins, so liegt ein potentieller Pfad-Anfangsknoten vor. In diesem Fall wird für alle herausführenden Kanten, die nicht als *nicht-löschbar* markiert wurden, untersucht, ob sie eine Verbindung zu einem Knoten mit einem Eingangsgrad größer eins haben. Auf dieser Verbindung dürfen ausschließlich (1,1)-Knoten liegen und die letzte Kante dieser Verbindung darf ebenfalls nicht als *nicht-löschbar* markiert sein. Tritt dieser Fall ein, so wurde ein löschbarer GP-Pfad gefunden.

Dieser Teilalgorithmus betrachtet nur eine Teilmenge aller Kanten des GP-Graphen und ist somit auch linear zu der Kantenzahl.

4.2.8.3 GP-Pfad löschen

Aus allen Pfaden, die im vorigen Schritt bestimmt wurden, wird einer zufällig ausgewählt und auf die im einleitenden Absatz beschriebene Weise gelöscht. Hierzu werden alle Kanten des Pfades sowie die Kanten des Pfad-Anfangs- und Pfad-Endknotens betrachtet bzw. verändert.

Die Laufzeit des gesamten Operators *Pfad löschen* ist somit linear zur Anzahl der Kanten des Graphen.

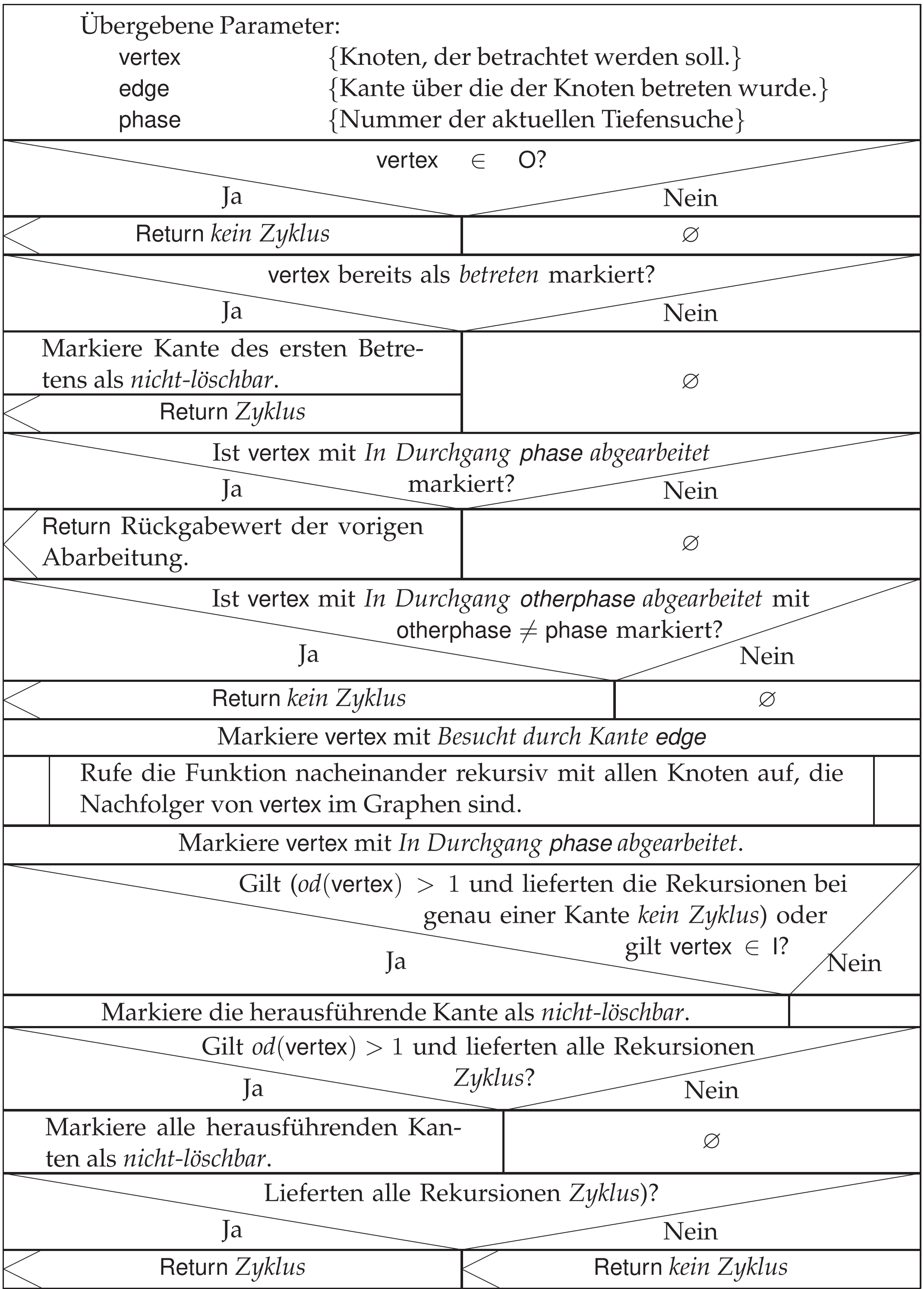


Abbildung 4.13: Der Teilalgorithmus zur *Kennzeichnung nicht-löschbarer Kanten* ist eine rekursive Funktion, die alle Kanten markiert, die entweder in einen Zyklus hinein- oder aus einem herausführen.

4.2.9 randomCrossover

Der *Crossover*-Operator bietet als einziger genetischer Operator die Möglichkeit, den genetischen Code zweier Individuen miteinander zu rekombinieren. Hierzu wird ein Teilgraph des GP-Graphen des zweiten Individuums durch einen Teilgraphen des ersten Individuums ersetzt. Das erste Individuum ist jenes, welches bei der Turnierselektion besser abgeschnitten hat. Der zugehörige GP-Graph wird ab jetzt G_1 genannt. Der Graph des zweiten Individuums wird mit G_2 bezeichnet.

Bei der Rekombination muss berücksichtigt werden, dass der resultierende Graph ebenfalls der Definition 3.1 eines GP-Graphen genügen muss und außerdem die maximale Graphgröße nicht überschreiten darf. Des Weiteren muss beachtet werden, dass die durch die zugeordnete Semantik vorgegebenen Eingangs- und Ausgangsgrade der Knoten erhalten bleiben.

Der implementierte Algorithmus arbeitet in vier Schritten. Zuerst werden in beiden Graphen mehrere Teilgraphen bestimmt, die für einen Austausch in Frage kommen. Als nächstes wird untersucht, welcher der Teilgraphen von G_2 durch welchen von G_1 unter Berücksichtigung von Definition 3.1 ersetzt werden kann. Danach wird ein Teilgraphenpaar (T_1, T_2) ausgewählt, wobei T_1 ein Teilgraph von G_1 ist und T_2 zu G_2 gehört. Es wird festgelegt, welche freigewordenen Kanten von G_2 nach der Entfernung des Teilgraphen T_2 mit welchen von T_1 verbunden werden sollen. In diesem Schritt wird der alte Teilgraph durch eine Kopie des neuen ersetzt. Abschließend wird überprüft, ob der neu erzeugte Graph ein GP-Graph ist.

Der Algorithmus beachtet implizit, dass Eingangs- und Ausgangsgrade aller Knoten der zugeordneten Semantik des GP-Graphen entsprechen. Dies wird dadurch erreicht, dass im neuen Graphen nur Knoten vorhanden sind, die bereits Teil eines der beiden Eltergraphen waren. Bei diesen Knoten wird die Anzahl der Eingangs- und Ausgangskanten beibehalten.

Bei dem in Abbildung 4.14 vorgestellten Algorithmus können auch Graphen erzeugt werden, die keine GP-Graphen sind oder Zyklen enthalten, obwohl nur zyklenfreie Graphen in der Population erwünscht sind. In diesem Fall wird der Operator noch einmal auf die beiden ursprünglichen Graphen angewandt. Es hat sich gezeigt, dass für Problemstellungen mit zyklischen Graphen in den meisten Fällen fünf Versuche ausreichen, um einen geeigneten neuen GP-Graph zu erhalten. Zyklische Problemstellungen und eine dazu passende Modifizierung des Operators werden in Kapitel 6 untersucht.

Wenn der Operator nach insgesamt fünf Versuchen mit dem gleichen Elternpaar keinen gültigen GP-Graphen erzeugen konnte, wird statt dessen eine Replikation des Individuums mit der besseren Fitness durchgeführt – der Crossover-Versuch gilt als gescheitert.

4.2.9.1 Bestimmung von Teilgraphen

Damit ein GP-Graph nach Definition 3.1 entsteht, muss die Anzahl der Kanten, die vom Rest des Graphen G_1 in den Teilgraphen T_1 führen, gleich der Anzahl der entsprechenden Kanten zwischen G_2 und T_2 sein. Diese Anzahl nennt sich *Eingangsgrad des Teilgraphen*. Das

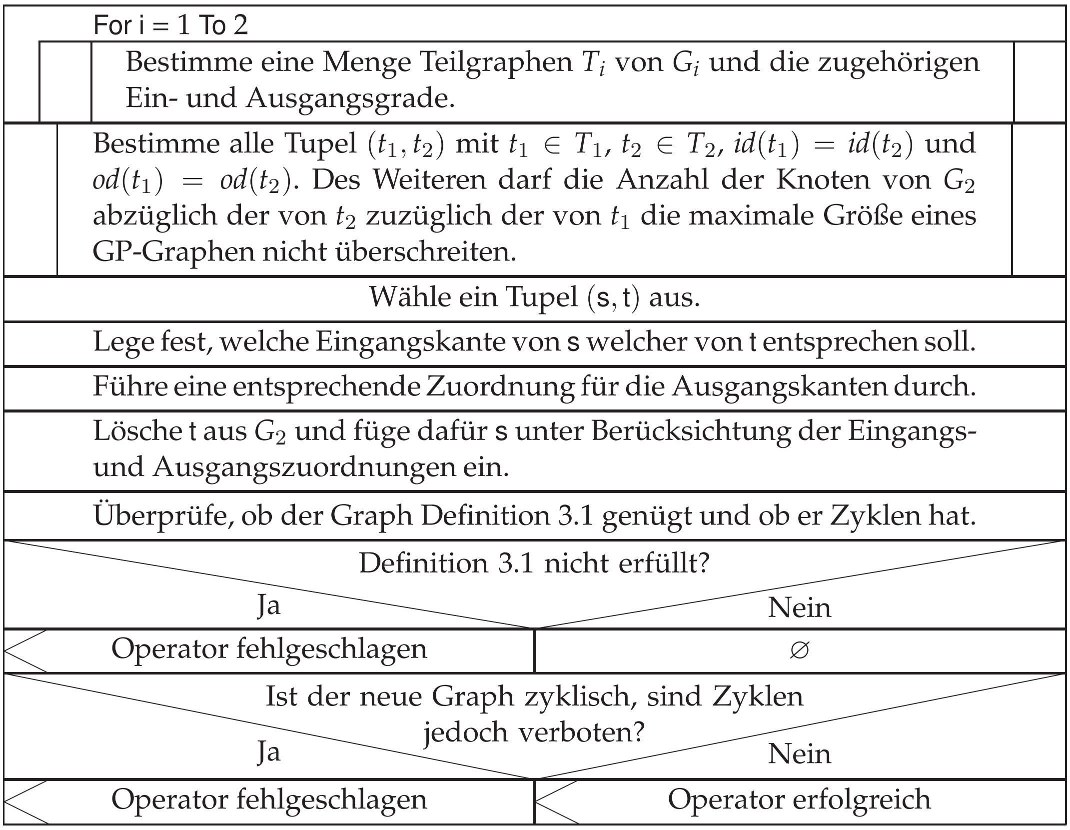


Abbildung 4.14: Der genetische Operator *Crossover* rekombiniert zwei GP-Graphen.

gleiche muss für die Kanten gelten die von T_1 nach G_1 bzw. von T_2 nach G_2 führen. Diese Anzahl wird als *Ausgangsgrad des Teilgraphen* bezeichnet.

Definition 4.3
Gegeben sei ein Graph (V, E) sowie ein Teilgraph (W, F) mit $W \subseteq V$ und $F \subseteq E$. Für die Kanten des Graphen gelte:

$$((v_1, v_2) \in E \wedge \{v_1, v_2\} \subseteq W) \Rightarrow (v_1, v_2) \in W$$

- *Der* Eingangsgrad des Teilgraphen *(W, F) bezeichne die Kardinalität der Menge $\{(v_1, v_2) | (v_1, v_2) \in E, v_1 \in V \setminus W, v_2 \in W\}$. Die Menge selbst heiße* Menge der Eingangskanten des Teilgraphen (W, F).
- *Der* Ausgangsgrad des Teilgraphen *(W, F) bezeichne die Kardinalität der Menge $\{(v_1, v_2) | (v_1, v_2) \in E, v_1 \in W, v_2 \in V \setminus W\}$. Die Menge selbst heiße* Menge der Ausgangskanten des Teilgraphen (W, F).

Sollten alle in Frage kommenden Crossover-Kombinationen möglicher Teilgraphen untersucht werden, müsste der Algorithmus von sämtlichen Teilgraphen der beiden Individuen den Ein- und Ausgangsgrad bestimmen. Die Anzahl der Teilgraphen eines GP-Graphen (I, O, V, E) entspricht der Kardinalität der Potenzmenge von V weniger eins (der leeren Menge)[1], also $2^{|V|} - 1$. Die Betrachtung aller Teilgraphen ist somit nicht möglich.

Voruntersuchungen haben ergeben, dass es ausreicht, pro GP-Graph so viele Teilgraphen zu untersuchen, wie er innere Knoten besitzt. Im Folgenden wird ein Algorithmus vorgestellt, der für einen GP-Graphen (I, O, V, E) für insgesamt $|V|$ Teilgraphen die Ein- und Ausgangsgrade berechnet. Die Graphen haben dabei die Größen von eins bis $|V|$.

Der Algorithmus beginnt mit einem Teilgraphen, bestehend aus einem zufällig ausgewählten Knoten aus V, dessen Ein- und Ausgangsgrad somit bekannt ist. Dieser Teilgraph wird nun schrittweise um jeweils einen weiteren Knoten ergänzt, indem ein weiterer Knoten aus V hinzugefügt wird. Es wird hierbei immer ein Knoten genommen, für den es eine Kante gibt, die ihn mit dem bestehenden Teilgraphen verbindet. Somit ist der Teilgraph immer zusammenhängend. Eine Ausnahme wird nur gemacht, wenn alle ein- und ausgehenden Kanten des Teilgraphen zu Eingangs- oder Ausgangsknoten des GP-Graphen führen. In diesem Fall wird aus den verbliebenen Knoten aus V einer zufällig ausgewählt.

Beim Hinzufügen eines weiteren Knotens zu dem Teilgraphen werden gleichzeitig alle weiteren Kanten aus E zur Kantenmenge des Teilgraphen hinzugefügt, die im GP-Graphen die Knoten des neuen Teilgraphen untereinander verbinden und noch nicht in seiner Kantenmenge sind. Aus diesen Kanten und dem Ein- und Ausgangsgrad des vorherigen Teilgraphen lassen sich leicht Ein- und Ausgangsgrad des neuen Teilgraphen berechnen (Abbildung 4.15).

4.2.9.2 Bestimmung von Teilgraph-Tupeln

Wurden für beide GP-Graphen $G_1 = (I_1, O_1, V_1, E_1)$ und $G_2 = (I_2, O_2, V_2, E_2)$ die Teilgraphmengen $\left\{T_1^1, \ldots, T_{|V_1|}^1\right\}$ und $\left\{T_1^2, \ldots, T_{|V_2|}^2\right\}$ bestimmt, müssen Paare (T_i^1, T_j^2) mit $T_i^1 = (V_i^1, E_i^1)$ und $T_j^2 = (V_j^2, E_j^2)$ gefunden werden, bei denen Eingangs- und Ausgangsgrad übereinstimmen und die die Gleichung $|V_2| + |I_2| + |O_2| + |V_i^1| - |V_j^2| \leq \mathit{maxsize}$ erfüllen, wobei *maxsize* die maximal erlaubte Größe eines GP-Graphen ist.

In der Implementierung werden alle Kombinationen einzeln geprüft, die sich aus den Teilgraphmengen ergeben. Daraus ergibt sich eine Laufzeit, die quadratisch zur maximalen Größe eines GP-Graphen ist. An dieser Stelle wären weitere Optimierungen der Laufzeit denkbar. So könnten die Teilgraphenmengen zuerst sortiert werden, wobei die lexikographische Ordnung auf den Tupeln aus Ein- und Ausgangsgrad als Sortierordnung benutzt werden könnte. Im Anschluss können passende Teilgraph-Tupel in einer Laufzeit gefunden werden, die linear zur Summe der Kardinalitäten der Teilgraphmengen ist. Es bleibt allerdings zu prüfen, ob die Reduzierung der nötigen Vergleichsoperationen gegenüber der Anzahl neu hinzukommender Initialisier-, Kopier- und Vergleichsoperationen für die Sortierung eine tatsächliche Reduzierung der Rechenzeit bewirkt.

[1] Hierbei wird vorausgesetzt, dass alle Kanten aus E, die in (I, O, V, E) Knoten des Teilgraphen verbinden, auch zum Teilgraphen gehören. Wäre dies nicht der Fall, würde die Anzahl der Teilgraphen gleich der Kardinalität der Potenzmenge von E weniger eins sein.

$knoten_0 = \emptyset$, $kanten_0 = \emptyset$, $kantenin_0 = \emptyset$, $kantenout_0 = \emptyset$, $kantenglobalin_0 = \emptyset$, $kantenglobalout_0 = \emptyset$

For i = 1 To |V|

$kantenin_{i-1} \cup kantenout_{i-1} = \emptyset$?

Ja

Wähle zufällig v mit $v \in V$ und $v \notin knoten_{i-1}$

$\emptyset$

Nein

Wähle zufällig $e = (v_1, v_2)$ mit $e \in kantenin_{i-1} \cup kantenout_{i-1}$

$e \in kantenin_{i-1}$?

Ja: $v = v_1$

Nein: $v = v_2$

$knoten_i = knoten_{i-1} + v$

$kantenin_i = kantenin_{i-1} + \{(w,v) | (w,v) \in E \wedge w \in V \wedge w \neq v\} - \{(u,w) | u,w \in knoten_i\}$

$kantenout_i = kantenout_{i-1} + \{(v,w) | (v,w) \in E \wedge w \in V \wedge w \neq v\} - \{(u,w) | u,w \in knoten_i\}$

$kantenglobalin_i = \{(u,w) | (u,w) \in E \wedge u \in I \wedge w \in knoten_i\}$

$kantenglobalout_i = \{(u,w) | (u,w) \in E \wedge u \in knoten_i \wedge w \in O\}$

$kanten_i = \{(u,w) | (u,w) \in E \wedge u,w \in V\}$

Abbildung 4.15: Der Algorithmus berechnet für einen GP-Graphen (I, O, V, E) insgesamt $|V|$ Teilgraphen $(knoten_1, kanten_1), \ldots, (knoten_{|V|}, kanten_{|V|})$. Die Mengen *kantenin* und *kantenout* enthalten jeweils die Kanten, die den Teilgraphen in (I, O, V, E) mit den restlichen Knoten aus V verbinden. Die Mengen *kantenglobalin* und *kantenglobalout* enthalten die Kanten zwischen den Teilgraphen und Ein- und Ausgangsknoten. Es gilt $id((knoten_i, kanten_i)) = |kantenin_i| + |kantenglobalin_i|$ und $od((knoten_i, kanten_i)) = |kantenout_i| + |kantenglobalout_i|$

Eine weitere Möglichkeit zur Geschwindigkeitssteigerung wäre die Verwendung von Hashing. Hierbei ist zu überprüfen, in welchem Verhältnis der Geschwindigkeitsgewinn beim Finden von Paaren zu einer Initialisierung der Hash-Tabellen stünde.

4.2.9.3 Zuordnung der Eingangs- und Ausgangskanten und Einbau des Teilgraphen

Wurde ein Tupel mit zwei Teilgraphen (T_1, T_2) ausgewählt, so kann festgelegt werden, wie der Teilgraph T_1 an Stelle des Teilgraphen T_2 in den GP-Graphen G_2 eingefügt werden soll.

Durch das Entfernen des Teilgraphen T_2 aus dem GP-Graphen G_2 wird für einige der Kanten in G_2 der Start- bzw. der Endknoten entfernt. Die Anzahl dieser Kanten entspricht jeweils dem Eingangs- und Ausgangsgrad des neu einzufügenden Teilgraphen. Somit ist es möglich, durch Verbinden des neuen Teilgraphen mit diesen Kanten einen GP-Graphen zu erzeugen, der den Punkten 1 bis 5 der GP-Graph-Definition 3.1 genügt. Aus den Graphen $G_1 = (I_1, O_1, V_1, E_1)$ und $G_2 = (I_2, O_2, V_2, E_2)$ sowie den beiden Teilgraphen $T_1 = (V_{T_1}, E_{T_1})$ und $T_2 = (V_{T_2}, E_{T_2})$ ergeben sich somit die folgenden vier Kantenmengen:

- $K_1 = \{(a, b)|a \in I_1 \cup O_1 \cup V_1 \setminus V_{T_1} \wedge b \in V_{T_1}\}$ mit $K_1 \subset E_1$
- $K_2 = \{(a, b)|a \in V_{T_1} \wedge b \in I_1 \cup O_1 \cup V_1 \setminus V_{T_1}\}$ mit $K_2 \subset E_1$
- $K_3 = \{(a, b)|a \in I_2 \cup O_2 \cup V_2 \setminus V_{T_2} \wedge b \in V_{T_2}\}$ mit $K_3 \subset E_2$
- $K_4 = \{(a, b)|a \in V_{T_2} \wedge b \in I_2 \cup O_2 \cup V_2 \setminus V_{T_2}\}$ mit $K_4 \subset E_2$

K_1 ist die Menge der Kanten, die in den Teilgraphen T_1 hineinführen und hat somit die gleiche Kardinalität wie die Menge K_3. Diese enthält entsprechend die Kanten, die in T_2 hineinführen. Derselbe Zusammenhang besteht zwischen K_2 und K_4. Diese Mengen enthalten die Kanten, die aus T_1 bzw. T_2 herausführen.

K_1 bzw. K_3 sind somit die Mengen der Eingangskanten der beiden Teilgraphen T_1 und T_2, die Mengen K_2 und K_4 sind entsprechend die Mengen der Ausgangskanten.

Zur Veränderung des Graphen G_2 werden alle in T_2 enthaltenen Knoten und Kanten aus G_2 entfernt. Der verbeibende Rest des Graphen wird *Hauptgraph* genannt. Die Kanten, von denen ein Endknoten durch das Entfernen von T_2 verloren gegangen ist, heißen *freie Kanten*. Die Knoten des Hauptgraphen, bei denen eine freie Kante beginnt, heißen *vordere Randknoten* und die Knoten des Hauptgraphen, bei denen eine freie Kante endet, heißen entsprechend *hintere Randknoten*.

Definition 4.4
Gegeben seien ein Graph (V, E) und ein zugehöriger Teilgraph (W, F), der dieselben Voraussetzungen erfüllt wie in Definition 4.3.

Die Kantenmenge E setze sich zusammen aus den Teilmengen $E_{main}, E_{sub}, E_{in}, E_{out}$ und W, mit

- $E_{main} = \{(a, b)|\{a, b\} \in V \setminus W\}$,
- $E_{sub} = \{(a, b)|\{a, b\} \in W\}$,
- $E_{in} = \{(a, b)|a \in V \setminus W, b \in W\}$ *und*
- $E_{out} = \{(a, b)|a \in W, b \in V \setminus W\}$.

1. *Die Knoten- und Kantenmenge (V_H, E_H) mit $V_H = V \setminus W$ und $E_H = E_{main} \cup E_{in} \cup E_{out}$ heiße Hauptgraph von (V, E).*
2. *Die Kantenmenge $E_{in} \cup E_{out}$ heiße freie Kanten von (V_H, E_H).*
3. *Die Knotenmenge $V_{vr} = \{v|v \in V_H \wedge \exists(v, e) \in E_{in}\}$ heiße Menge der vorderen Randknoten.*
4. *Die Knotenmenge $V_{hr} = \{v|v \in V_H \wedge \exists(e, v) \in E_{out}\}$ heiße Menge der hinteren Randknoten.*

Nach der Entfernung von T_2 werden alle freien Kanten von G_2 durch neue Kanten ersetzt, die den Hauptgraphen von G_2 mit den Eingangs- und Ausgangskanten von T_1 verbinden. Hierzu wird jeder Kante aus K_3 bijektiv eine Kante aus K_1 zugewiesen. Diese Zuordnung erfolgt in einer ersten Implementierung des *Crossover*-Operators zufällig. Auf die gleiche Weise wird jeder Kante aus K_4 zufällig eine aus K_2 zugewiesen. Für jedes dieser Paare wird eine neue Kante zu E_2 hinzugefügt:

- Sei $(a, b) \in K_1$ und $(c, d) \in K_3$ ein Kantenpaar. Ergänze E_2 durch $E_2 = E_2 \cup \{(c, b)\}$.
- Sei $(a, b) \in K_2$ und $(c, d) \in K_4$ ein Kantenpaar. Ergänze E_2 durch $E_2 = E_2 \cup \{(a, d)\}$.

Der Name *randomCrossover* des Operators wird von dieser zufälligen Kantenzuordnung abgeleitet. Die in den Abschnitten 6.2 und 6.3 vorgestellten *Crossover*-Operatoren unterscheiden sich nur durch andere Methoden der Kantenzuordnung.

In Kapitel 6 wird auf Grenzen der hier vorgestellten Implementierung eingegangen und wie sie durch eine Modifizierung des Algorithmus zu beheben sind.

In Punkt sechs der GP-Graph-Definition wird gefordert, dass sowohl die eingehenden als auch die ausgehenden Kanten eines Knotens eine Reihenfolge besitzen. Nach dem Verändern des Graphen G_2 werden die Ordnungen aller Knoten von V_2 und dem Teilgraphen T_1 übernommen. Für alle Kanten, die durch das Entfernen des alten Teilgraphen T_2 wegfallen, werden neue Kanten hinzugefügt. Diese nehmen in der Reihenfolge denselben Platz ein, wie die Kanten aus K_1 bis K_4, für die sie jeweils eingesetzt wurden. Somit ist über sämtliche Eingangs- und Ausgangskanten aller Knoten wieder eine Ordnung festgelegt.

4.2.9.4 Überprüfung weiterer GP-Graph-Eigenschaften

Durch die zufällige Bildung von Kantenpaaren aus K_1 und K_3 bzw. K_2 und K_4 ist nicht sichergestellt, dass der neue Graph die Punkte 7 und 8 der GP-Graph-Definition erfüllt. Durch die zufällige Wahl ist es außerdem möglich, dass der neue Graph Zyklen enthält, obwohl alle Individuen der Population zyklenfrei sein sollen.

Dies wird in einem abschließenden Graphdurchlauf untersucht. Von jedem Eingangsknoten und jedem (0,1)-Knoten ausgehend wird ein Tiefendurchlauf entlang des Graphen durchgeführt. Wird während eines Durchlaufs ein Knoten zum zweiten Mal erreicht, enthält der Graph einen Zyklus. Ist von einem der Eingangsknoten kein Ausgangsknoten erreichbar, so verstößt der Graph gegen Punkt 7 der GP-Graph-Definition 3.1. Ist von einem der (0,1)-Knoten kein Ausgangsknoten erreichbar, ist Punkt 8 der Definition verletzt.

Während der Durchläufe wird jeder innere Knoten als *besucht* gekennzeichnet. Nach den Tiefendurchläufen werden alle inneren Knoten auf die *besucht*-Markierung hin überprüft. Fehlt die Markierung bei einem der Knoten, so ist für diesen Punkt acht der GP-Graph-Definition verletzt. Es liegt ein verbotener Zyklus vor, wie er in Abbildung 4.10 dargestellt ist.

Der Algorithmus entspricht in weiten Teilen dem zur *Kennzeichnung nicht-löschbarer Kanten* in Abbildung 4.13 und wird deshalb nicht gesondert vorgestellt.

4.2.9.5 Verbesserungen des Crossover-Operators

Der zeitintensivste Teilalgorithmus des Crossovers ist die Bestimmung der Teilgraphen, die für einen Austausch in Frage kommen. Durch die – vor Einführung des Crossovers – gewählte Datenstruktur zur Implementierung des Graphen, ergibt sich beim sequenziellen Aufbau der Teilgraphen aus den vorher entwickelten eine kubische Laufzeit. Durch eine andere interne Implementierung des Graphen könnte diese auf quadratisches Verhalten verbessert werden.

In der Grundform wird zuerst der komplette Algorithmus des Operators durchgeführt. Anschließend wird überprüft, ob der erzeugte Graph alle Eigenschaften eines GP-Graphen erfüllt. Ist dies nicht der Fall, wird der gesamte Algorithmus wiederholt. Dieser Vorgang kann ebenfalls beschleunigt werden, indem

- entweder auf ein weiteres Tupel der bereits berechneten Teilgraphpaarungen zurückgegriffen wird
- oder das gleiche Teilgraphtupel noch einmal verwendet wird und nur andere Zuordnungen für Eingangs- und Ausgangskanten berechnet werden.

Der zweite Fall verringert die Laufzeit im Vergleich zum ersten nur marginal, da nur das Auswählen eines Tupels eingespart wird. Er kann jedoch mit dem Überprüfen des Graphen auf GP-Grapheigenschaften verknüpft werden. Auf diese Weise wird bereits vor dem Einfügen sichergestellt, dass der neue Graph die geforderten Eigenschaften erfüllt.

Hierzu müssen die Randknoten von G_2 näher betrachtet werden.

Vor der Zuordnung der Randknoten von G_2 zu den Ein- und Ausgangskanten des neuen Teilgraphen muss für diese Knoten festgestellt werden, welche von ihnen nach dem Entfernen von T_2 noch über Pfade zu den Eingangs- und Ausgangsknoten aus $I_2 \cup O_2$ verfügen. Für die Randknoten des Teilgraphen T_1 ist zu überprüfen, welche von ihnen untereinander Verbindungen besitzen, die komplett im Teilgraphen enthalten sind.

Diese beiden Untersuchungen können, wie bei der oben beschriebenen *Überprüfung der GP-Graph-Eigenschaften*, mit Tiefendurchläufen in linearer Laufzeit zur Anzahl der Kanten durchgeführt werden, da sämtliche Kanten des Hauptgraphen zu G_2 bzw. die Kanten des Teilgraphen T_1 genau einmal betrachtet werden müssen. Die Tiefendurchläufe starten jeweils bei den entsprechenden Randknoten.

Mit den Informationen über die internen Verbindungen von T_1 können die Eingangs- und Ausgangskanten so den Randknoten von G_2 zugewiesen werden, dass diese alle die Punkte sieben und acht der GP-Graph-Definition 3.1 erfüllen. Durch die Informationen über die Verbindungen der Randknoten von G_2 untereinander kann zudem bei zyklenfreien Graphen überprüft werden, ob der neue Graph ebenfalls zyklenfrei bleibt.

Es kann vorkommen, dass für eine GP-Graph/Teilgraph Kombination keine zyklenfreie Lösung existiert. In so einem Fall kann entweder ein neues Teilgraphenpaar ausgewählt werden oder es können komplett neue Teilgraphen errechnet werden.

Kapitel 5

Testprobleme

In diesem Kapitel werden die in der Arbeit verwendeten Testprobleme vorgestellt. Da alle bereits aus der Literatur bekannt sind, werden sie an dieser Stelle lediglich kurz mit den entsprechenden Verweisen eingeführt. Um eine Reproduktion der Ergebnisse zu ermöglichen, wird die gewählte Implementierung aller Benchmarks formal dargestellt und begründet.

Zum Vergleich der Ergebnisse wird das auf KOZAs Baumansatz basierende GP-System *lilgp* [92] verwendet, in dem viele der Benchmarks bereits implementiert sind oder einfach hinzugefügt werden konnten.

Zum Vergleich der Ergebnisse werden zwei verschiedene Größen verwendet. Besitzt ein Problem ein Optimum, das während der Evolution realisiert wird, so wird KOZAs *Effort* als Vergleichskriterium herangezogen [56]. Dieser Wert sagt aus, nach wie vielen Fitnessauswertungen mit 99-prozentiger Sicherheit das Optimum gefunden wird. Wie von CHRISTENSEN und OPPACHER in [21] bereits gezeigt wurde, ist die Berechnung dieses Wertes zwar als Vergleichskriterium wünschenswert, jedoch statistisch eher unsicher. Diesem Punkt wird in dieser Arbeit durch eine geänderte Berechnung des Wertes Rechnung getragen. Außerdem werden weitere Faktoren aufgezeigt, die bei der Interpretation des Wertes berücksichtigt werden müssen. Die Methodik zur Berechnung des *Efforts* sowie weitere Erkenntnisse zur Aussagekraft dieses Wertes befinden sich in Abschnitt 5.4. Die *Effort*-Werte, die in diesem Kapitel aufgeführt werden, sind alle nach der hier vorgestellten Methode berechnet. Werden andere in der Literatur veröffentlichte Ergebnisse vorgestellt, die auf KOZAs *Effort*-Berechnung beruhen, wird dies explizit ausgeführt. Aufgrund der statistischen Ungenauigkeit des Wertes sollten beim Vergleich zweier Ergebnisse neben dem *Effort*-Wert auch die Anzahl der erfolgreichen Versuche und die benötigten Fitnessauswertungen der einzelnen Läufe betrachtet werden.

Wenn die während der Evolution gefundene Lösung nur eine Annäherung an das Optimum ist, kann kein *Effort*-Wert berechnet werden. In solchen Fällen werden statt dessen die besten Individuen aller durchgeführten Läufe betrachtet und deren durchschnittliche Fitness berechnet.

Bei allen Testproblemen wird eine multikriterielle Fitnessfunktion verwendet. Das erste Kriterium ist immer problemspezifisch und ist in der Regel eine Formel, die den Abstand der gefundenen Lösung zum Optimum angibt. Das bedeutet, dass kleinere Werte immer für eine bessere Fitness stehen.

Das zweite Kriterium ist die Größe des Graphen. Diese wird nur zum Vergleich zweier Individuen herangezogen, die im Bezug auf das erste Kriterium dieselbe Fitness aufweisen. In diesem Fall hat das Individuum mit weniger Knoten die bessere Fitness. Dies soll einerseits dazu führen, dass Introns vermieden werden, andererseits soll die Fähigkeit zur Generalisierung der Individuen gefördert werden. Je größer ein Individuum ist, desto wahrscheinlicher wird ein auswendig lernen der Testwerte, das so genannte *Overfitting*. Durch die Unterstützung kleinerer Graphen soll dies vermieden werden.

In diesem Kapitel werden lediglich die Probleme vorgestellt, die in der Arbeit häufiger vorkommen. Wenn an einer Stelle ein neuer Benchmark verwendet wird, der sich mit einer speziellen Problemstellung oder einer speziellen Variante beschäftigt, wird er an geeigneter Stelle eingeführt.

Zu jedem der Benchmarks werden Testläufe mit dem in Kapitel 4 beschriebenen GP-System *GGP* durchgeführt. Die Ergebnisse werden in den folgenden Kapiteln als Vergleichswerte für die dort vorgestellten Neuerungen dienen. Parallel zu diesen Testläufen werden alle Probleme mit dem baumbasierten GP-System *lilgp* untersucht, das in der verwendeten Form KOZAs ursprünglichem GP-System entspricht.

Die Vergleiche zwischen den Ergebnissen der beiden Systeme dienen ausschließlich der Einordnung des nicht-optimierten Graph-GP-Systems bezüglich eines weit verbreiteten GP-Systems, dessen Ergebnisse für andere leicht reproduzierbar sind. Für alle Testprobleme gibt es für Baum-GP mittlerweile Parametrisierungen oder neue GP-Techniken, die zu besseren Ergebnissen führen.

5.1 Symbolische Regression

Die symbolische Regression ist eines der ersten Gebiete, auf denen die Genetische Programmierung eingesetzt wurde. Neben der Evolution trigonometrischer Funktionen wird hier das *Two-Boxes*-Problem vorgestellt.

5.1.1 Ein einfaches Polynom

Die Funktion $x^4 + x^3 + x^2 + x$ wird in der Literatur immer wieder verwendet und findet als Standardtestproblem auch in dieser Arbeit Berücksichtigung. Aufgabe des GP-Systems ist es, eine Funktion $\hat{f}$ zu ermitteln, die das Polynom $f : x \mapsto x^4 + x^3 + x^2 + x$ im Intervall $[-1, 1]$ möglichst genau annähert. Es gilt also $\forall x \in [-1, 1] : |f(x) - f(\hat{x})| \leq \epsilon$, mit möglichst kleinem Wert ϵ.

Zur Realisierung des Problems wird eine funktionale Semantik verwendet. Die Menge der Grundfunktionen besteht aus den Grundrechenarten, einer Konstanten *Const*, einer Verzweigung *Branch*, einer *No-Operation*-Funktion *Nop* und einer Funktion *Inp*, die dem aktuellen x-Wert entspricht:

$$F = \{Add, Sub, Mul, Div, Const, Branch, Nop, Inp\} \qquad \text{mit}$$

- $Add : \mathbb{R}^2 \times \varnothing \to \mathbb{R}$ und $Add : (x_1, x_2)^T \mapsto x_1 + x_2$
- $Sub : \mathbb{R}^2 \times \varnothing \to \mathbb{R}$ und $Sub : (x_1, x_2)^T \mapsto x_1 - x_2$
- $Mul : \mathbb{R}^2 \times \varnothing \to \mathbb{R}$ und $Mul : (x_1, x_2)^T \mapsto x_1 x_2$
- $Div : \mathbb{R}^2 \times \varnothing \to \mathbb{R}$ und $Div : (x_1, x_2)^T \mapsto \begin{cases} \text{MAXFLOAT} & \text{falls} \quad x2 = 0 \\ x1/x2 & \text{sonst} \end{cases}$
- $Const : \varnothing \times [-1, 1] \to \mathbb{R}$ und $Const : p \mapsto p$
- $Branch : \mathbb{R} \times \varnothing \to \mathbb{R}^2$ und $Branch : x_1 \mapsto (x_1, x_1)^T$
- $Nop : \mathbb{R} \times \varnothing \to \mathbb{R}$ und $Nop : x \mapsto x$
- $Inp : \varnothing \times \varnothing \to \mathbb{R}$ und Inp : INPUTVAL

Die *Nop*-Funktion wird benötigt, da der Algorithmus zur Initialisierung der Graphen das Vorhandensein eines (1,1)-Knotens voraussetzt. Existiert dieser nicht, ist das Erstellen einer vollständigen Population nicht sichergestellt. Diese Knoten werden im Laufe der Evolution aus dem Genom eines Individuums entfernt, da sie nur Graphen vergrößern, ohne jedoch einen positiven Einfluss auf den Fitnesswert auszuüben. Der Unterschied zu einem Baumansatz besteht ansonsten nur in der Verwendung eines *Branch*-Knotens.

Zur Fitnessberechnung werden 51 äquidistant zwischen -1 und 1 verteilte Stützpunkte verwendet. Die Fitness eines Individuums mit der evolvierten Funktion $\hat{f}$ ergibt sich aus der quadratischen Abweichung zur Sollfunktion f:

$$fitness = \frac{1}{51} \sum_{x=-1; x=x+0.04}^{1} (f(x) - \hat{f}(x))^2$$

Um spätere Ergebnisse vergleichen zu können, wurden initial 200 GP-Läufe mit den Parametern aus Tabelle 5.1 durchgeführt. Der genetische Operator *Knoten löschen* wird mit einer sehr geringen Wahrscheinlichkeit von 2 Prozent verwendet, da er nur zum Entfernen der *Nop*-Knoten eingesetzt wird. Enthält ein Individuum keinen derartigen Knoten mehr, dient *Knoten löschen* als Replikationsoperator.

Die Parameter für die 200 Vergleichsläufe mit dem baumbasierten System wurden mit den in der *lilgp*-Distribution ausgelieferten Parametern durchgeführt. An dieser Stelle soll es nicht darum gehen, die beiden Systeme mit exakt derselben Initialisierung zu vergleichen, sondern vielmehr darum, dass beide Systeme gemäß ihren Anforderungen an die Problemkodierung sinnvoll arbeiten können. Um die Ergebnisse von *lilgp* nicht negativ zu beeinflussen, wurden daher die bereits erprobten Standardwerte verwendet, die von KOZA in [58] vorgeschlagen wurden.

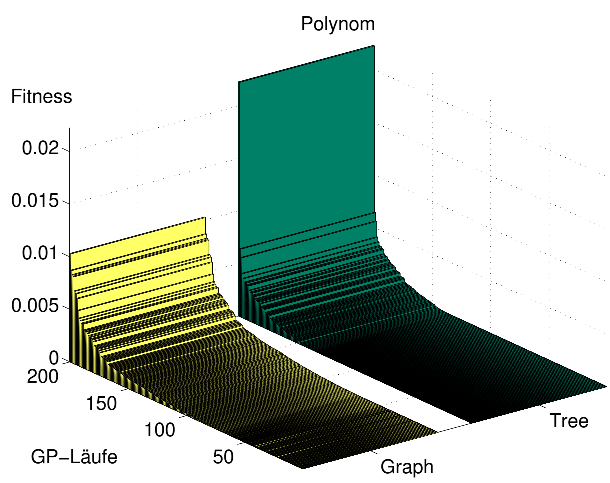
Parameter	Wert
GP-System	Graph
Populationsgröße	100
Fitnessauswertungen	200000
Turniergröße	4/2
max. Graphgröße	80
Knoten mutieren	24%
GP-Pfad löschen	24%
GP-Pfad einfügen	24%
Crossover	24%
Knoten löschen	2%
Läufe	200

Parameter	Wert
GP-System	Baum
Populationsgröße	5000
Fitnessauswertungen	200000
Selektion	Fitness-Proportional
max. Baumgröße	$2^{17}-1$
Crossover	90%
Läufe	200

Tabelle 5.1: Die Initialisierungswerte für das *Polynom*-Problem $x^4+x^3+x^2+x$

Ergebnisse

Nach Durchführung der jeweils 200 Läufe für graph- und baumbasiertes GP wurde aus jedem Lauf das beste Individuum ermittelt. Die Fitnesswerte dieser Individuen sind für beide Verfahren – jeweils nach den Fitnesswerten sortiert – in Abbildung 5.1 zusammengefasst.



	min	max	mean
Tree	0	0.02228	0.00056
Graph	0	0.01029	0.00092

Abbildung 5.1: Die Fitnesswerte des jeweils besten Individuums aller 400 Testläufe beim *Polynom*-Problem

Die rechte Tabelle der Abbildung enthält für beide Verfahren in der mit *min* überschriebenen Spalte die Fitness des besten Individuums aller Siegerindividuen der 200 Läufe, in der *max*-Spalte das schlechteste. Die letzte Spalte enthält die durchschnittliche Fitness der besten Individuen aller GP-Läufe eines Verfahrens. Bei beiden Verfahren gibt es Läufe in denen ein Individuum die optimale Fitness 0 erreicht. Das insgesamt schlechteste Ergebnis aller 400 Läufe trat beim baumbasierten GP-System auf, bei dem das beste Individuum nur eine Fitness von 0.02228 erreichte. Im Durchschnitt wurden allerdings mit dem baumbasierten System bessere Lösungen gefunden als mit dem graphbasierten.

5.1.2 Das *Two-Boxes*-Problem

Ein in der Literatur häufig anzutreffendes Problem ist das so genannte *Two-Boxes*-Problem [58]. Die Aufgabe besteht in der Ermittlung der Differenz von den Volumina zweier Quader. Gesucht wird also die Formel $l_1w_1h_1 - l_2w_2h_2$, wobei l, w und h jeweils für die Längen, Breiten und Höhen der Boxen bestehen.

Zur Implementierung der Problemstellung wird eine funktionale Semantik verwendet. Als Grundfunktionen werden die vier Grundrechenarten, ein *Branch*-, ein *No-Operation*-Befehl sowie eine Funktion *Inpv* verwendet, die – je nach Wert ihres Parameters p – für einen der sechs Eingabewerte stehen kann:

$$F = \{Add, Sub, Mul, Div, Branch, Nop, Inpv\} \qquad \text{mit}$$

- $Add : \mathbb{R}^2 \times \varnothing \to \mathbb{R}$ und $Add : (x_1, x_2)^T \mapsto x_1 + x_2$
- $Sub : \mathbb{R}^2 \times \varnothing \to \mathbb{R}$ und $Sub : (x_1, x_2)^T \mapsto x_1 - x_2$
- $Mul : \mathbb{R}^2 \times \varnothing \to \mathbb{R}$ und $Mul : (x_1, x_2)^T \mapsto x_1 x_2$
- $Div : \mathbb{R}^2 \times \varnothing \to \mathbb{R}$ und $Div : (x_1, x_2)^T \mapsto \begin{cases} \text{MAXFLOAT} & \text{falls} \quad x2 = 0 \\ x1/x2 & \text{sonst} \end{cases}$
- $Branch : \mathbb{R} \times \varnothing \to \mathbb{R}^2$ und $Branch : x_1 \mapsto (x_1, x_1)^T$
- $Nop : \mathbb{R} \times \varnothing \to \mathbb{R}$ und $Nop : x \mapsto x$
- $Inpv : \varnothing \times [0, 6[\to \mathbb{R} \quad \text{und} \quad Inpv : p \mapsto \begin{cases} l_1 & \text{falls} \quad p \in [0, 1[\\ w_1 & \text{falls} \quad p \in [1, 2[\\ h_1 & \text{falls} \quad p \in [2, 3[\\ l_2 & \text{falls} \quad p \in [3, 4[\\ w_2 & \text{falls} \quad p \in [4, 5[\\ h_2 & \text{falls} \quad p \in [5, 6[\end{cases}$

Die *Nop*-Funktion wird benötigt, da der Algorithmus zur Initialisierung der Graphen das Vorhandensein eines (1,1)-Knotens voraussetzt. Existiert dieser nicht, ist das Erstellen einer vollständigen Population nicht sichergestellt. Diese Knoten werden im Laufe der Evolution aus dem Genom eines Individuums entfernt, da sie nur Graphen vergrößern ohne einen positiven Einfluss auf den Fitnesswert auszuüben. Der Unterschied zu einem Baumansatz besteht ansonsten nur in der Verwendung eines *Branch*-Knotens.

Eine Schwierigkeit des Problems besteht darin, dass keine Funktion zur Verfügung steht, durch die direkt Konstanten vorgegeben sind. Eine Annäherung an das Optimum wird somit erschwert.

Die Fitness wird anhand von zehn Testdatensätzen berechnet (Tabelle 5.2):

$$\mathit{fitness} = \frac{1}{10}\sum(\hat{f}(l_1, w_1, h_1, l_2, w_2, h_2) - \mathit{value})^2$$

	l_1	w_1	h_1	l_2	w_2	h_2	value
1	3	4	7	2	5	3	54
2	7	10	9	10	3	1	600
3	10	9	4	8	1	6	312
4	3	9	5	1	6	4	111
5	4	3	2	7	6	1	-18
6	3	3	1	9	5	4	-171
7	5	9	9	1	7	6	363
8	7	2	9	9	9	2	-36
9	2	6	8	2	6	10	-24
10	1	10	7	5	1	45	-155

$$\mathit{value} = l_1 w_1 h_1 - l_2 w_2 h_2$$

Tabelle 5.2: Testdatensätze für das *Two-Boxes*-Problem

Um spätere Ergebnisse vergleichen zu können, wurden initial 200 GP-Läufe mit den Parametern aus Tabelle 5.3 durchgeführt. Der genetische Operator *Knoten löschen* wird mit einer sehr geringen Wahrscheinlichkeit von 2 Prozent verwendet, da er nur zum Entfernen der *Nop*-Knoten eingesetzt wird. Enthält ein Individuum keinen derartigen Knoten mehr, dient *Knoten löschen* als Replikationsoperator.

Die Parameter für die 200 Vergleichsläufe mit dem baumbasierten System wurden wiederum mit den in der *lilgp*-Distribution ausgelieferten Parametern durchgeführt, die den von KOZA in [58] vorgeschlagenen Werten entsprachen.

Ergebnisse

Aus den Ergebnispopulationen der jeweils 200 Läufe für graph- und baumbasiertes GP wurde jeweils das beste Individuum ermittelt. Abbildung 5.2 gibt eine Übersicht über die nach ihrem Wert sortierten Fitnesswerte. In der Tabelle stehen unter *succ.* die Anzahl der Läufe, in denen ein Individuum mit Fitness 0 gefunden wurde, in denen also eine korrekte Formel der Volumendifferenz evolviert wurde.[1] Die Spalte *impr.* gibt den Durchschnittswert

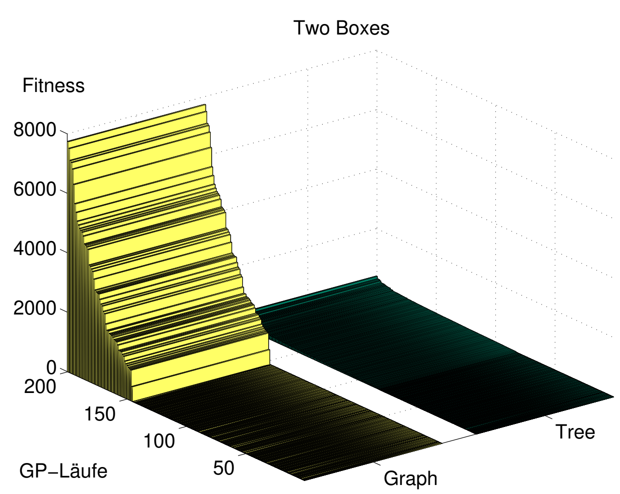
[1] Eine genauere Betrachtung der Bäume/Graphen dieser Individuen ergab, dass es sich tatsächlich immer um generalisierende Individuen handelte.

Parameter	Wert
GP-System	Graph
Populationsgröße	100
Fitnessauswertungen	200000
Turniergröße	4/2
max. Graphgröße	20
Knoten mutieren	24%
GP-Pfad löschen	24%
GP-Pfad einfügen	24%
Crossover	24%
Knoten löschen	2%
Läufe	200

Parameter	Wert
GP-System	Baum
Populationsgröße	4000
Fitnessauswertungen	200000
Selektion	Turnier (7/1)
max. Baumgröße	$2^{17} - 1$
Crossover	90%
Läufe	200

Tabelle 5.3: Die Initialisierungswerte für das *Two-Boxes*-Problem

an, nach wie vielen Fitnessberechnungen ein Individuum mit Fitness 0 gefunden wurde. In diese Zahl gehen somit nur die erfolgreichen Läufe ein. GP-Läufe, in denen kein Individuum mit Fitness 0 gefunden wurde, werden bei dieser Berechnung nicht berücksichtigt. In der Spalte *effort* steht der so genannte *Effort*-Wert. Näheres zur Berechnung und Aussagekraft dieses Wertes steht in Abschnitt 5.4.



	succ.	impr.
Tree	82	69366
Graph	145	30440

	effort (evals)
Tree	882167 (84000)
Graph	59793 (3226)

Abbildung 5.2: Die Fitnesswerte des jeweils besten Individuums aller 400 Testläufe für das *Two Boxes*-Problem

Die Grafik der Abbildung erweckt zunächst die Vorstellung, dass das baumbasierte System deutlich überlegen wäre. Diese Vermutung ist allerdings falsch. Während das graphbasierte System *GGP* bei 200 Läufen in 145 Fällen eine korrekte Formel für die Volumendifferenz finden konnte, schaffte es das baumbasierte GP-System nur in 82 Fällen. Die Größe der

Lösungsindividuen, die von beiden Verfahren erreicht wurden, war ebenfalls sehr unterschiedlich. Beim baumbasierten GP war sie sehr uneinheitlich und schwankte zwischen zwölf und 118 Knoten. Der Mittelwert war hierbei 37.7 bei einer Standardabweichung von 25.3. Bei den 145 Lösungen von *GGP* bestand ein Graph aus 20 und einer aus 14 Knoten. Die restlichen 143 Lösungen bestanden aus exakt zwölf Knoten und entsprechen damit der optimalen Kodierung der Formel $l_1 w_1 h_1 - l_2 w_2 h_2$ für dieses GP-System.

In Abbildung 5.2 ist zu sehen, dass alle mit *GGP* erzeugten Individuen, die nicht die gesuchte Formel repräsentieren, einen hohen Fitnesswert besitzen. Die Individuen der baumbasierten Läufe hingegen haben alle eine Fitness nahe null. Dies ist zu großen Teilen auf den unterschiedlichen Suchraum bei beiden Ansätzen zurückzuführen. Bei *GGP* ist der Suchraum durch die Beschränkung auf Graphen mit maximal 20 Knoten im Vergleich zu Bäumen der Tiefe 17 sehr klein. Durch den zusätzlichen Verzicht auf skalare Werte ist es nur schwer möglich, bei der vorgegebenen Knotenzahl Vorfaktoren für die Eingangsgrößen (z.B.$l_1/l_1 + l_1/l_1$) zu evolvieren. Beim baumbasierten Ansatz ist es jedoch möglich und kommt in den Individuen mit einer Fitness größer null entsprechend häufig vor.

Die Ergebnisse lassen darauf schließen, dass bei kleinen Graphen viele lokale Optima mit relativ schlechter Fitness existierten, die während der Evolution jedoch wieder verlassen werden können und die Suchstrategie also entsprechend global ist. Bei großen Bäumen hingegen werden oft lokale Optima mit relativ guter Fitness erreicht, die aber nicht mehr verlassen werden können. Die Suche ist demnach eher lokal.

Die Beschränkung des Suchraums auf 20 Knoten lässt sich nicht ohne weiteres auf das baumbasierte System übertragen. Zur Validierung wurden 200 weitere Läufe mit dem baumbasierten System und den Parametern aus Tabelle 5.2 durchgeführt, wobei die Anzahl der erlaubten Knoten auf 20 gesetzt wurde. In diesen Läufen konnten nur noch in 18 Fällen Lösungen gefunden werden.

Wie bereits geschildert, sind die gefundenen Lösungen des baumbasierten Systems wesentlich größer als die von *GGP*. Die Lösungen entsprachen zwar der gesuchten Formel, mussten jedoch noch gekürzt werden. Von den 82 Lösungen verwendeten 32 *ADF*s. Bei *GGP* lagen alle Lösungsformeln bereits in gekürzter Form vor. Ursache hierfür ist die Einbeziehung der Graphgröße in die Fitnessfunktion.

Die Vorteile von *GGP* gegen den ursprünglichen Baumansatz zeigen sich des Weiteren auch bei der Anzahl der durchschnittlich benötigten Fitnessauswertungen. Bei den erfolgreichen Läufen benötigte *GGP* durchschnittlich weniger als halb so viele Auswertungen wie der baumbasierte Ansatz.

Die *Effort*-Werte der beiden Verfahren zeigen ebenfalls eine deutliche Tendenz zu Gunsten von *GGP*. Diesen Werten sollte jedoch aufgrund der Ausführungen in [21] und Abschnitt 5.4 nicht zu viel Bedeutung zugesprochen werden.

5.1.3 Trigonometrische Funktionen

Als weiteres Testproblem zur symbolischen Regression wird die erste Periode der Sinusfunktion verwendet. Die Grundfunktionsmenge entspricht hierbei zu großen Teilen der aus Abschnitt 5.1.2. Lediglich die Funktion *Inpv* wurde durch die parameterlose Funktion *Inp* ersetzt. Zusätzlich stehen dem GP-System mit der Funktion *Const* Konstanten zur Verfügung:

$$F = \{Add, Sub, Mul, Div, Const, Branch, Nop, Inp\} \qquad \text{mit}$$

- $Const : \varnothing \times [-1, 1] \to \mathbb{R}$ und $Const : p \mapsto p$
- $Inp : \varnothing \times \varnothing \to \mathbb{R}$ und Inp : INPUTVAL

Als Stützstellen werden 51 äquidistant verteilte Werte aus dem Intervall $[0, 6.3]$ verwendet.

Die Sinusfunktion ist aus zwei Gründen schwieriger zu evolvieren als das Polynom: Zum einen verfügt die Funktion im betrachteten Bereich über mehr Extremstellen, zum anderen wird die richtige Verwendung von Vorfaktoren und Konstanten relevanter.

Es wurden erneut jeweils 200 Läufe mit dem graphbasierten GP-System und dem baumbasierten System durchgeführt. Die Initialisierungen erfolgten gemäß Tabelle 5.4.

Parameter	Wert
GP-System	Graph
Populationsgröße	100
Fitnessauswertungen	200000
Turniergröße	4/2
max. Graphgröße	80
Knoten mutieren	24%
GP-Pfad löschen	24%
GP-Pfad einfügen	24%
Crossover	24%
Knoten löschen	2%
Läufe	200

Parameter	Wert
GP-System	Baum
Populationsgröße	1000
Fitnessauswertungen	200000
Selektion	Fitness-Proportional
max. Baumgröße	$2^{17} - 1$
Crossover	90%
Läufe	200

Tabelle 5.4: Die Initialisierungswerte für das *Sinus*-Problem

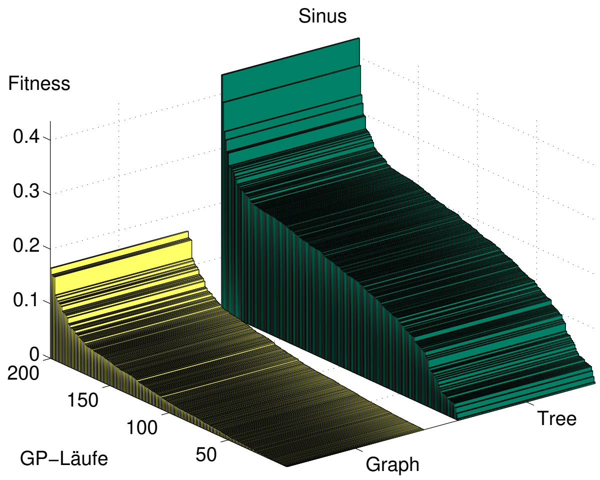
Da dieses Problem nicht Bestandteil der *lilgp*-Distribution ist, wurden in Vorversuchen Testläufe mit verschiedenen Parametersätzen durchgeführt und der beste für die eigentlichen Vergleiche ausgewählt.

Die verwendete Fitnessfunktion sah wie folgt aus:

$$fitness = \frac{1}{51} \sum_{x=0;x=x+0.126}^{6.3} (\sin(x) - \hat{f}(x))^2$$

Ergebnisse

Nach Durchführung der jeweils 200 Läufe für graph- und baumbasiertes GP wurde aus jedem Lauf das beste Individuum ermittelt. Die Fitnesswerte dieser Individuen sind für beide



	min	max	mean
Tree	0.00239	0.43506	0.16975
Graph	0.00019	0.16536	0.03071

Abbildung 5.3: Die Fitnesswerte des jeweils besten Individuums aller 400 Testläufe beim *Sinus*-Problem

Verfahren – jeweils nach den Fitnesswerten sortiert – in Abbildung 5.3 zusammengefasst. Der Aufbau der Tabelle entspricht dem aus Abbildung 5.1.

Die Fitnesswerte beim graphbasierten GP-System liegen zwischen 0.00019 und 0.16536 (bei einem Durchschnitt von 0.03071 und einem Standardfehler von 0.00227). Wie es bereits das Diagramm vermuten lässt, sind diese Werte auch nach einem Wilcoxon-Rangsummentest signifikant besser als die mit baumbasiertem GP erzielten Werte. Diese liegen zwischen 0.00239 und 0.43506. Der Durchschnitt ist 0.16975 und der Standardfehler 0.00489.

Die baumbasierten Individuen bestehen im Durchschnitt aus 140.84 Knoten. Die Extremwerte sind hierbei 5 und 629. Die durchschnittliche Graphgröße der besten Individuen bei *GGP* liegt bei 79.5 Knoten. Die vorgegebene maximale Größe von 80 Knoten wird demnach bei fast allen Individuen ausgenutzt.

5.2 Klassifikation

Neben der symbolischen Regression ist die Klassifikation ein weiteres klassisches Feld der Genetischen Programmierung. Aus diesem Bereich wird das von Brameier eingeführte Testproblem verwendet, bei dem die Punkte zweier Ringe jeweils einem der beiden zugeordnet werden müssen [14]. Abbildung 5.4 zeigt die beiden Ringe. Zu jedem Ring gehören jeweils 250 Punkte.

Die Menge der Grundfunktionen wurde im Vergleich zu den Problemen der symbolischen Regression um die trigonometrische Funktionen *sin* und *cos* sowie um eine Schwellwertfunktion *Threshold* erweitert. Die Implementierung dieser Funktion unterscheidet sich bei

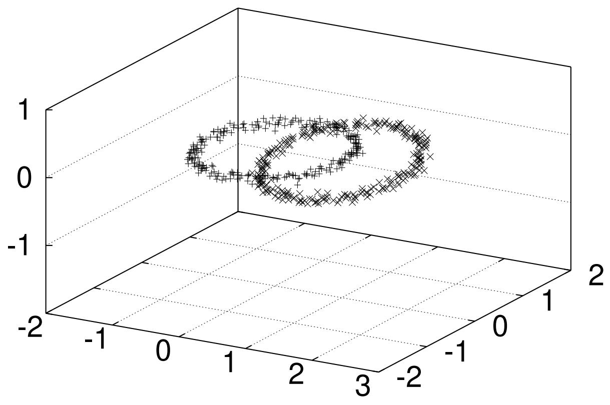


Abbildung 5.4: Der Datensatz zu den zwei ineinander verschränkten Ringen des *Klassifikations*-Problems

Graph- und Baum-GP: Beim Graph-GP wird sie mit einen (1,1)-Knoten mit einem Parameter realisiert. Wenn der Eingangswert des Knotens größer oder gleich dem Parameter ist, so ist der Ausgangswert 1. In allen anderen ist der Ausgangswert -1. Beim Baum-GP dient die Funktion statt dessen als Vergleich zwischen den Werten zweier Söhne. Ist der Wert des ersten Sohns größer oder gleich dem des zweiten, so erhält der Knoten den Wert 1. Ist dies nicht der Fall, so wird der Wert -1. Der zweite Sohn übernimmt somit die Rolle des Parameters.

Die Menge der Grundfunktionen ist gegeben durch:

$$F = \{Add, Sub, Mul, Div, Const, Branch, Inpv, Threshold, Sin, Cos\} \qquad \text{mit}$$

- $Const : \varnothing \times [0, 10] \to \mathbb{R}$ und $Const : p \mapsto p$
- $Inpv : \varnothing \times [0, 3[\to \mathbb{R} \quad \text{und} \quad Inpv : p \mapsto \begin{cases} x & \text{falls} \quad p \in [0, 1[\\ y & \text{falls} \quad p \in [1, 2[\\ z & \text{falls} \quad p \in [2, 3[\end{cases}$
- $Sin : \mathbb{R} \times \varnothing \to \mathbb{R}$ und $Sin : x \mapsto \sin(x)$
- $Cos : \mathbb{R} \times \varnothing \to \mathbb{R}$ und $Cos : x \mapsto \cos(x)$

Als Fitnessfunktion diente die Anzahl der falsch klassifizierten Punkte dividiert durch die Gesamtzahl der Punkte.

In diesem Benchmark werden mit den Funktionen *Sinus* und *Cosinus* erstmals (1,1)-Knoten im Graph verwendet, die für die Fitnessentwicklung eine Rolle spielen (im Gegensatz zur

Nop-Funktion). Aus diesem Grund werden nun auch die genetischen Operatoren *Knoten einfügen, löschen* und *verschieben* verwendet. Die Tabelle 5.5 gibt eine Übersicht über die verwendeten Parameter.

Parameter	Wert
GP-System	Graph
Populationsgröße	100
Fitnessauswertungen	200000
Turniergröße	4/2
max. Graphgröße	80
Knoten mutieren	14%
GP-Pfad löschen	14%
GP-Pfad einfügen	14%
Knoten löschen	14%
Knoten einfügen	14%
Crossover	14%
Knoten verschieben	14%
Läufe	200

Parameter	Wert
GP-System	Baum
Populationsgröße	500
Fitnessauswertungen	200000
Selektion	Fitness-Proportional
max. Baumgröße	$2^{17}-1$
Crossover	90%
Läufe	200

Tabelle 5.5: Die Initialisierungswerte für das *Klassifikations*-Problem

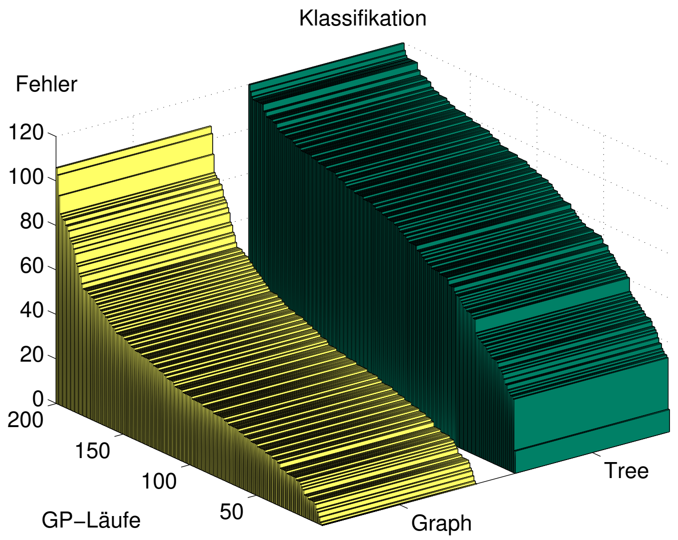
Ergebnisse

Nach der Durchführung der jeweils 200 Läufe für graph- und baumbasiertes GP wurde aus jedem Lauf das beste Individuum ermittelt. Die Fitnesswerte dieser Individuen sind für beide Verfahren – jeweils nach den Fitnesswerten sortiert – in Abbildung 5.5 zusammengefasst. Der Fitnesswert eines Individuums entspricht der Anzahl der falsch klassifizierten Punkte dividiert durch die Gesamtzahl der Klassifizierungen. Zur besseren Übersichtlichkeit werden in der Abbildung sowohl im Diagramm als auch in der Tabelle die Fehlklassifizierungen aufgezeigt. Dies entspricht jeweils dem Fitnesswert multipliziert mit 500. Die Tabelle enthält somit die Anzahl der Fehlklassifizierungen des besten und schlechtesten Individuums der jeweils 200 Läufe sowie den Durchschnittswert aller 200 Individuen.

Die besten Individuen, die beim graphbasierten System evolviert wurden, hatten zwischen 0 und 106 Punkte falsch klassifiziert – im Durchschnitt 35.2. Die Ergebnisse des baumbasierten Systems waren mit 10 bis 120 falschen Klassifikationen und einem Durchschnitt von 83.8 signifikant[1] schlechter.

Die Größen der Graphen beim *GGP*-System schwankten zwischen 16 und 79 Knoten. Im Gegensatz zum *Sinus*-Problem lag der Durchschnitt mit 44.2 Knoten allerdings deutlich unter der maximalen erlaubten Größe. Die Baumgrößen des GP-Systems nach KOZA lagen zwischen 5 und 263 Knoten. Der Durchschnitt lag hier mit 44.9 Knoten nahe bei dem des graphbasierten Systems.

[1]Es wurde ein Wilcoxon-Rangsummentest durchgeführt.



	min	max	mean
Tree	10	120	83.8
Graph	0	106	35.2

Abbildung 5.5: Die Fitnesswerte des jeweils besten Individuums aller 400 Testläufe beim *Klassifikations*-Problem

5.3 Algorithmische Probleme

Alle bisher vorgestellten Probleme basieren auf funktionalen Semantiken. Die größten Vorteile des graphbasierten Ansatzes sind jedoch bei Problemstellungen zu erwarten, bei denen sich Individuen besser durch Graphen als durch Bäume darstellen lassen. Dies ist bei Problemen mit algorithmischer Semantik zu erwarten, bei denen Zyklen erlaubt sind. Aus dieser Problemklasse werden zwei Benchmarks betrachtet, das *Artificial Ant*-Problem in Abschnitt 5.3.1 sowie das *Lawnmower*-Problem in Abschnitt 5.3.2.

5.3.1 Das *Artificial Ant*-Problem

Bei *Artificial Ant*-Problemen wird ein Algorithmus für eine Ameise gesucht, die auf einem Spielfeld Nahrungsstücke einsammeln muss. Das Spielfeld wird als $n \times n$ Felder umfassender Torus modelliert, die Felder, auf denen sich ein Nahrungsstück befindet, werden entsprechend gekennzeichnet. Die Ameise kann sich ein Feld vorwärts bewegen und sich um 90 Grad nach rechts oder links drehen. Des Weiteren existiert eine Abfragefunktion, ob sich auf dem Feld vor der Ameise ein Nahrungsstück befindet. Die Ameise hat keine Funktionen zur Verfügung, die ihr Informationen über den bereits zurückgelegten Weg, die Größe des Torus oder die Anzahl der verbleibenden Nahrungsstücke liefern.

Als Verteilung der Nahrungsstücke wird in dieser Arbeit der so genannte *Santa Fé*-Pfad verwendet. Bei diesem befinden sich 89 Nahrungsstücke auf einem 32×32 Felder umfassenden Spielfeld. Die Positionierung der Nahrung ist Abbildung 5.6 zu entnehmen. Die Ameise startet in der linken oberen Ecke und zeigt zu Beginn nach rechts.

Die Ameise darf insgesamt 400 Schritte ausführen. Hierzu zählen alle Vorwärts- und Drehbewegungen. Die verbliebenen Nahrungsstücke, die Stellung der Ameise und die Anzahl der noch erlaubten Schritte werden in der Parametermenge $P = (\mathit{food}, \mathit{ant}, \mathit{cmd})$ kodiert. Bei dem 3-Tupel steht *food* für eine Teilmenge aller Koordinaten, auf denen sich noch Nahrung befindet, es gilt also $\mathit{food} \subseteq \{(0,0)\ldots(0,31),(1,0)\ldots(1,31),\ldots,(31,0)\ldots(31,31)\}$. Der Wert *ant* steht für zwei 2-Tupel, bei dem das erste für die Koordinaten der Ameise und das zweite für die Blickrichtung steht: $(31,0)$ - Nord, $(1,0)$ - Süd, $(0,31)$ - West, $(0,1)$ - Ost. Der Wert $\mathit{cmd} \in \mathbb{N}_0$ steht für die Anzahl der Schritte, die der Ameise noch zur Verfügung stehen.

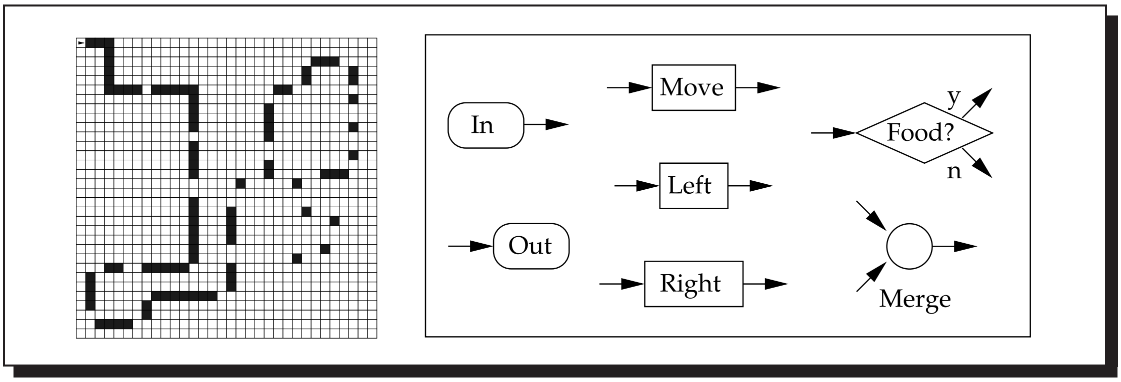


Abbildung 5.6: Der *Santa Fé*-Pfad und die für das Problem verwendeten Grundfunktionen.

Die Parametermenge P_0 ist in Tabelle 5.6 abgebildet.

$$\begin{aligned}
P_0 = (\{&(0,1),(0,2),(0,3),(1,3),(2,3),(3,3),(4,3),(5,3),(5,4),(5,5),(5,6),(5,8),(5,9),\\
&(5,10),(5,11),(5,12),(6,12),(7,12),(8,12),(9,12),(11,12),(12,12),(13,12),\\
&(14,12),(17,12),(18,12),(19,12),(20,12),(21,12),(22,12),(23,12),(24,11),\\
&(24,10),(24,9),(24,8),(24,7),(24,4),(24,3),(25,1),(26,1),(27,1),(28,1),\\
&(30,2),(30,3),(30,4),(30,5),(29,7),(28,7),(27,8),(27,9),(27,10),(27,11),\\
&(27,12),(27,13),(27,14),(26,16),(25,16),(24,16),(21,16),(20,16),(19,16),\\
&(18,16),(15,17),(14,20),(13,20),(10,20),(9,20),(8,20),(7,20),(5,21),(5,22),\\
&(4,24),(3,24),(2,25),(2,26),(2,27),(3,29),(4,29),(6,29),(9,29),(12,29),\\
&(14,28),(14,27),(14,26),(15,23),(18,24),(19,27),(22,26),(23,23)\},\\
&((0,0),(0,1)),400)
\end{aligned}$$

Tabelle 5.6: Die Parametermenge, mit der jede Fitnessauswertung initialisiert wird.

Die Grundfunktionen $F = \{\text{Food, Move, Left, Right, Merge}\}$ mit

$$\begin{aligned}
\text{Food}: &\quad P \to P \times \{1,2\} \times \{1\} \times \{2\} \\
\text{Left}: &\quad P \to P \times \{1\} \times \{1\} \times \{1\} \\
\text{Right}: &\quad P \to P \times \{1\} \times \{1\} \times \{1\} \\
\text{Move}: &\quad P \to P \times \{1\} \times \{1\} \times \{1\} \\
\text{Merge}: &\quad P \to P \times \{1\} \times \{2\} \times \{1\}
\end{aligned}$$

sind folgendermaßen definiert:

$$\text{Food}: \quad (food, (pos, dir), cmd) \mapsto (food, (pos, dir), cmd) \times j \times 1 \times 2$$

$$\text{mit} \quad j = \begin{cases} 1 & \text{falls } ((pos + dir) \ mod \ 32) \in \text{food}^1 \\ 2 & \text{sonst} \end{cases}$$

$$\text{Left}: \quad (food, (pos, dir), cmd) \mapsto (food, (pos, \widehat{dir}), \widehat{cmd}) \times 1 \times 1 \times 1$$

$$\widehat{dir} = \begin{cases} (31,0) & \text{falls } dir = (0,1) \\ (0,31) & \text{falls } dir = (31,0) \\ (1,0) & \text{falls } dir = (0,31) \\ (0,1) & \text{falls } dir = (1,0) \end{cases} \qquad \widehat{cmd} = cmd - 1$$

$$\text{Right}: \quad (food, (pos, dir), cmd) \mapsto (food, (pos, \widehat{dir}), \widehat{cmd}) \times 1 \times 1 \times 1$$

$$\widehat{dir} = \begin{cases} (1,0) & \text{falls } dir = (0,1) \\ (0,1) & \text{falls } dir = (31,0) \\ (31,0) & \text{falls } dir = (0,31) \\ (0,31) & \text{falls } dir = (1,0) \end{cases} \qquad \widehat{cmd} = cmd - 1$$

$$\text{Move}: \quad (food, (pos, dir), cmd) \mapsto (\widehat{food}, (\widehat{pos}, dir), \widehat{cmd}) \times 1 \times 1 \times 1$$

$$\widehat{pos} = (pos + dir) \ mod \ 32, \quad \widehat{food} = food \setminus \widehat{pos}, \quad \widehat{cmd} = cmd - 1$$

$$\text{Merge}: \quad (food, (pos, dir), cmd) \mapsto (food, (pos, dir), cmd) \times 1 \times 2 \times 1$$

Bei den Graphen, durch welche die Algorithmen modelliert werden, sind Zyklen erlaubt. Bei einer Fitnessauswertung wird der Graph so lange durchlaufen, bis die Anzahl der verbliebenen Schritte auf null gesunken ist. Wird während der Ausführung des Algorithmus der *Out*-Knoten erreicht, so wird die Auswertung beim *In*-Knoten fortgesetzt.

Der Fitnesswert eines Individuums setzt sich – neben der Größe des Graphen als sekundäres Fitnesskriterium – aus zwei Werten zusammen. Zum einen ist dies die Anzahl der nach 400 Schritten nicht gefundenen Nahrungsstücke, zum anderen die Nummer des Schritts, in dem das letzte gefundene Nahrungsstück aufgenommen wurde, geteilt durch 1000. Eine Fitness von 1.354 bedeutet beispielsweise, dass nach 400 Schritten ein Nahrungsstück liegen geblieben ist und das letzte der 88 gefundenen Stücke mit dem Schritt Nummer 354 aufgenommen wurde.

Tabelle 5.7 zeigt die gewählten Parametersätze der Testläufe.

[1] Die Modulo-Rechnung wird auf beiden Komponenten des Tupels (pos, dir) einzeln durchgeführt

Parameter	Wert
GP-System	Graph
Populationsgröße	100
Fitnessauswertungen	200000
Turniergröße	4/2
max. Graphgröße	25
Knoten mutieren	12%
GP-Pfad löschen	12%
GP-Pfad einfügen	12%
Zyklus	12%
Knoten löschen	12%
Knoten einfügen	12%
Crossover	12%
Knoten verschieben	12%
Läufe	200

Parameter	Wert
GP-System	Baum
Populationsgröße	1000
Fitnessauswertungen	200000
Selektion	Overselection
max. Baumgröße	$2^{17} - 1$
Crossover	90%
Läufe	200

Tabelle 5.7: Die Initialisierungswerte für das *Artificial Ant*-Problem

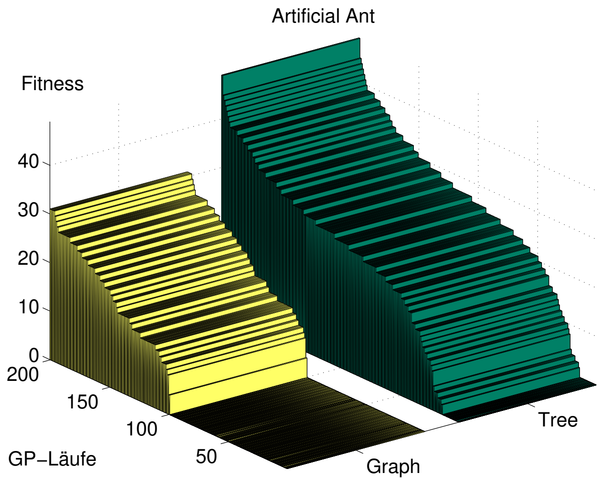
Am Beispiel des *Artificial Ant*-Problems zeigt sich deutlich der Vorteil der gewählten Graph-Struktur: Jeder evolvierte Graph entspricht direkt dem Flussdiagramm des Algorithmus, der durch ihn dargestellt wird. Im Fall der Baum-Kodierung nach *Koza* (siehe [58]) lässt sich jeder Baum zwar ebenfalls als Flussdiagramm darstellen, allerdings ist die Erzeugung von Zyklen nicht möglich. Somit ist es mit KOZAs Kodierung nicht möglich, eine Schleife als Kontrollstruktur zu evolvieren.

Ergebnisse

Nach der Durchführung der jeweils 200 Läufe für graph- und baumbasiertes GP wurde aus jedem Lauf das beste Individuum ermittelt. Die Fitnesswerte dieser Individuen sind für beide Verfahren – jeweils nach den Fitnesswerten sortiert – in Abbildung 5.7 zusammengefasst. Die Tabelle ist wie die in Abbildung 5.2 aufgebaut.

Es ist deutlich zu sehen, dass *GGP* dem ursprünglichen Baumansatz in diesem Beispiel überlegen ist. Das baumbasierte System konnte nur in 14 Fällen ein Programm für die Ameise erzeugen, bei dem alle Nahrungstücke aufgesammelt wurden. *GGP* war dazu in 99 Fällen in der Lage.

Die Graphen bzw. Bäume der Lösungen sind bei *GGP* deutlich kleiner als beim baumbasierten System, was durch die Begrenzung der Knotenzahlen zu erwarten war. Die Graphgröße liegt bei *GGP* zwischen 17 und 25 Knoten, im Durchschnitt bei 23.5 (Standardabweichung 1.8). Die evolvierten Bäume bewegen sich hingegen zwischen 31 und 238 Knoten, im Schnitt bei 128 (Standardabweichung 71.9). Es zeigt sich somit auch in diesem Fall wieder, dass *GGP* Lösungen in kleineren Suchräumen finden kann. Zur Validierung wurden weitere 200 Läufe mit dem baumbasierten System durchgeführt, bei denen die Knotenobergrenze eben-



	succ.	impr.
Tree	14	103071
Graph	99	46980

	effort (evals)
Tree	8315529 (56000)
Graph	443321 (40000)

Abbildung 5.7: Die Fitnesswerte des jeweils besten Individuums aller 400 Testläufe beim *Artificial Ant*-Problems

falls auf 25 festgesetzt wurde. Auf diese Weise wurden in 38 Läufen Lösungen gefunden. Dies zeigt, dass bei baumbasiertem GP ebenfalls eine Beschränkung des Suchraums sinnvoll ist, aber nicht zu vergleichbaren Ergebnissen wie beim graphbasierten Ansatz führt.

Die Tabelle aus Abbildung 5.7 zeigt des Weiteren, dass *GGP* Lösungen im Durchschnitt mit weniger als halb so vielen Fitnessauswertungen finden konnte wie das baumbasierte System. Zusammen mit der deutlich höheren Anzahl gefundener Lösungen führte dies somit zu einem kleineren *Effort*-Wert beim graphbasierten System.

5.3.2 Das *Lawnmower*-Problem

Das *Lawnmower*-Problem kann als eine Variante des *Artificial Ant*-Problems gesehen werden. Ziel der Problemstellung ist es, mit einem Rasenmäher den Rasen auf einem Torus zu mähen. Setzt man den Rasenmäher mit einer Ameise des *Artificial Ant*-Problems gleich, so ergeben sich folgende Unterschiede:

- Während beim *Artificial Ant*-Problem nur eine Teilmenge der Felder des Torus betreten werden müssen, sind beim *Lawnmower* alle Felder anzusteuern.
- Beim *Artificial Ant*-Problem gibt es mit der *Food*-Funktion eine Möglichkeit, auf den aktuellen Zustand des Torus zu reagieren. Beim *Lawnmower*-Problem ist dies nicht vorgesehen.

Die gesuchten Lösungen der beiden Probleme müssen demnach unterschiedliche Ansätze verfolgen. Beim *Artificial Ant*-Problem muss ein Programm evolviert werden, das die Umgebung der Ameise nach einem Nahrungsstück systematisch absucht. Beim *Lawnmower*-Problem hingegen muss ein Algorithmus gefunden werden, der die Fortbewegung nur in Abhängigkeit der Feldgröße festlegt.

Die Realisierung des *Lawnmower*-Problems innerhalb des GP-Systems entspricht zu großen Teilen der des *Artificial Ant*-Problems. Die Parametermenge P besteht ebenfalls aus einem 3-Tupel mit $P = \{lawn, mower, cmd\}$, wobei *lawn* der Menge *food* und *mower* dem 2-Tupel *ant* entspricht und *cmd* bei beiden Problemen für die Anzahl der Schritte steht (vergl. 5.3.1). Die Menge *lawn* enthält in der initialen Parametermenge alle Koordinaten des Torus. Die Startkoordinate des Rasenmähers ist (0,0) mit Blick nach Osten und die Anzahl der erlaubten Schritte ist mit 100 vorgegeben.

Die Menge der Grundfunktionen *F* umfasst die Funktionen *Left*, *Mow*, *Frog*, *Merge* und *Loop* mit

$$\begin{aligned}
\text{Mow}: &\quad P \to P \times \{1\} \times \{1\} \times \{1\} \\
\text{Left}: &\quad P \to P \times \{1\} \times \{1\} \times \{1\} \\
\text{Frog}: &\quad P \times \{0, \ldots, 7\}^2 \to P \times \{1\} \times \{1\} \times \{1\} \\
\text{Loop}: &\quad P \to P \times \{1\} \times \{1\} \times \{2\} \\
\text{Merge}: &\quad P \to P \times \{1\} \times \{2\} \times \{1\}
\end{aligned}$$

Mow entspricht dem Befehl *Move* bei *Artificial Ant*-Problem, *Left* einer Linksdrehung des Rasenmähers und *Merge* ist bei beiden Problemen identisch. *Frog* stellt eine weitere Möglichkeit der Fortbewegung zur Verfügung. Gemäß der zwei lokalen Parameter aus der Menge $\{0, \ldots, 7\}$ wird der Rasenmäher um die entsprechende Zahl von Feldern vorwärts und nach links versetzt. Die Anzahl der Schritte wird nur von den Funktionen *Mow* und *Frog* um eins erniedrigt. *Left* hat auf diesen Wert keinen Einfluss.

Das *Lawnmower*-Problem wird häufig als Beispiel für den sinnvollen Einsatz von *ADF*s angeführt, da ein baumbasiertes GP-System ohne diese Technik nur in den seltensten Fällen eine Lösung für das Problem findet. Aus diesem Grund wurde die im Programmpaket *lilgp* als Standard vorgegebene Benutzung von *ADF*s bei den Testläufen beibehalten. Für das graphbasierte System wurde als Ausgleich die zusätzliche Grundfunktion *Loop* eingeführt, mit der die Erzeugung einer einfachen Kontrollstruktur innerhalb eines Programms möglich ist. Beim *Artificial Ant*-Problem wurde der Graph nach Erreichen eines *Out*-Knotens wieder von vorne ausgeführt. Dies ist beim *Lawnmower*-Problem nicht vorgesehen. Stattdessen entspricht die *Loop*-Funktion einem *Goto*-Befehl, mit dem diese sonst implizite Schleife in den Graphen verschoben wird. Der *Loop*-Befehl erzeugt somit eine Endlosschleife, die nur durch das Aufbrauchen der verfügbaren Schritte beendet wird.[1]

$$\text{Loop}: \quad P \mapsto P \times 1 \times 1 \times 1$$

Tabelle 5.8 zeigt die gewählten Parametersätze der Testläufe.

[1]Bei den Testläufen hat sich gezeigt, dass der *Loop*-Befehl bei fast allen richtigen Lösungen auch als *While (cmd* $\neq 0$*)* interpretiert werden kann, da der zweite Ausgang immer direkt mit dem *Out*-Knoten des Graphen verbunden ist.

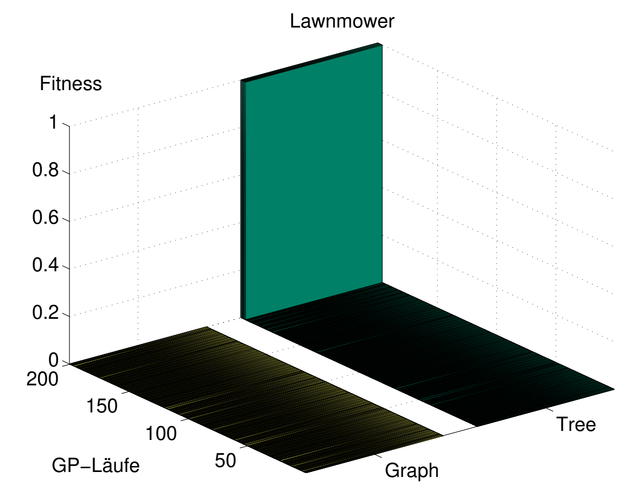
Parameter	Wert
GP-System	Graph
Populationsgröße	100
Fitnessauswertungen	200000
Turniergröße	4/2
max. Graphgröße	40
Rasengröße	8×8
Knoten mutieren	12%
GP-Pfad löschen	12%
GP-Pfad einfügen	12%
Zyklus	12%
Knoten löschen	12%
Knoten einfügen	12%
Crossover	12%
Knoten verschieben	12%
Läufe	200

Parameter	Wert
GP-System	Baum
Populationsgröße	300
Fitnessauswertungen	200000
Selektion	Turnier (7/1)
max. Baumgröße	$2^{17} - 1$
Besonderheiten	*ADFs*
Rasengröße	8×8
Crossover	90%
Läufe	200

Tabelle 5.8: Die Initialisierungswerte für das *Lawnmower*-Problem

Ergebnisse

Abbildung 5.8 fasst die Ergebnisse der Testläufe zusammen. Der Aufbau von Diagramm und Tabelle entsprechen dem von Abbildung 5.2. Mit Ausnahme von vier Läufen des baumbasierten GP-Systems konnte in allen Versuchen eine Lösung des Problems gefunden werden.



	succ.	impr.
Tree	196	7171
Graph	200	1744

	effort (evals)
Tree	14565 (12600)
Graph	5190 (2997)

Abbildung 5.8: Die Fitnesswerte des jeweils besten Individuums aller 400 Testläufe beim *Lawnmower*-Problem

Die Größen der Lösungsindividuen waren wiederum erwartungsgemäß sehr unterschiedlich. Während sie bei *GGP* zwischen 12 und 16 Knoten lagen, war der kleinste Baum einer baumbasierten GP-Laufs 48 Knoten groß, der größte sogar 463. Der Durchschnitt lag bei diesem Ansatz bei 159 Knoten mit einer Standardabweichung von 68.6.

Es zeigt sich somit, dass KOZAs System dazu neigt, selbst sehr einfache Algorithmen (*gehe achtmal vorwärts und springe danach in eine andere Zeile*) durch unlesbar große Bäume zu kodieren. Durch die Einbeziehung der Knotenzahl als sekundäres Kriterium in die Fitnessberechnung wird dieser *Bloat* im graphbasierten Ansatz vermieden.

Die *success*- und *improvement*-Spalten der Tabelle in Abbildung 5.8 zeigen erneut, dass *GGP* auch hier schneller eine Lösung findet als das ursprüngliche Baum-GP-System. Bei *GGP* werden im Durchschnitt nur ein viertel der Fitnessauswertungen vom baumbasierten System benötigt.

5.4 Berechnung des *Effort*-Wertes

Bei mehreren Ergebnissen zu den Testproblemen in diesem Abschnitt wurde unter anderem ein *Effort*-Wert aufgeführt. Es handelt sich um einen, von KOZA in [57] eingeführten statistischen Wert, mit dem die Ergebnisse verschiedener GP-Ansätze verglichen werden können. Es wurde bereits aufgezeigt, dass dieser Wert nur eine geringe Aussagekraft hat und in diesem Zusammenhang auf eine Veröffentlichung von CHRISTENSEN und OPPACHER verwiesen. An diesem Abschnitt werden weitere Probleme des *Effort*-Wertes aufgezeigt.

Die hier vorgestellten Beobachtungen sind in [81] veröffentlicht.

In [57] stellt KOZA eine Methode vor, mit der die Ergebnisse unterschiedlicher evolutionärer Verfahren bzw. verschiedener Modifikationen von *GP* miteinander verglichen werden können. Hierbei wird berechnet, wie viele Fitnessauswertungen mit dem Verfahren durchgeführt werden müssen, um mit 99 Prozent Wahrscheinlichkeit eine Lösung für das untersuchte Problem zu finden. Diesen Wert bezeichnet er als *Computational Effort*. Er wird in dieser Arbeit als *Effort*-Wert bezeichnet.

Da bei der Genetischen Programmierung viele Läufe gar nicht oder zu sehr unterschiedlichen Zeiten das Optimum zu einer Aufgabenstellung finden, kann diese Zahl nur durch empirisches Vorgehen berechnet werden und beruht dann auf relativen Häufigkeiten, die als Schätzer für die tatsächlichen Wahrscheinlichkeiten einer Lösungsfindung nach einer bestimmten Anzahl von Fitnessauswertungen verwendet werden. Die Formel, die KOZA zur Berechnung des Wertes vorschlägt, lautet:

$$I(M,z) = \min_i Mi \left\lceil \frac{\ln(1-z)}{\ln(1-P(M,i))} \right\rceil \tag{5.1}$$

Hierbei steht M für die Populationsgröße und i für die Anzahl der berechneten Generationen. Der Wert Mi ist somit die Anzahl der Fitnessauswertungen, die in einem Lauf durchgeführt werden. Der Wert $P(M,i)$ steht für die relative Häufigkeiten, mit denen die durchgeführten Läufe bei einer Populationsgröße von M nach i Generationen eine Lösung gefunden haben. Der Wert z steht für das Konfidenzniveau, mit dem eine Lösung gefunden wird. Sie wird wie bei KOZA im Folgenden immer auf $z = 99\%$ gesetzt.

In [21] zeigen CHRISTENSEN und OPPACHER, dass nach KOZA berechnete Werte bei üblichen GP-Problemen teilweise über 25 Prozent vom tatsächlichen Wert entfernt liegen können. Als Gründe hierfür werden unter anderem das Aufrunden des Bruches und die Verwendung des Minimumoperators aufgeführt.

In diesem Abschnitt wird zunächst eine leicht variierte Berechnung des *Effort*-Wertes vorgestellt, die den durch Turnierselektion hervorgerufenen *Steady-State*-Charakter des hier vorgestellten GP-Systems berücksichtigt und dabei die Kritikpunkte von CHRISTENSEN und OPPACHER etwas abschwächt.

Im Anschluss werden weitere Probleme des *Effort*-Wertes dargelegt, die auch die hier verwendete Berechnung des *Efforts* betreffen und zu einer sehr vorsichtigen Bewertung der Aussagekraft dieser Größe führen sollten.

5.4.1 Der *Effort*-Wert für *Steady-State*-Algorithmen

Werden bei KOZAs generationsbasierten GP-System n Läufe durchgeführt, so ergeben sich die relativen Häufigkeiten $P(M,i)$ jeweils als k/n, wobei k die Anzahl der Läufe ist, bei denen nach i Generationen eine Lösung gefunden wurde und n die Gesamtzahl der durchgeführten Läufe. Betrachtet man die Erhöhung von $P(M,i)$ über die Anzahl der Fitnessauswertungen, so treten die Erhöhungen von P nur alle M Fitnessauswertungen auf und stehen dann für alle Läufe, die irgendwann innerhalb der letzten M Auswertungen ihre Lösung gefunden haben. Durch einen *Steady-State*-Ansatz kann die relative Häufigkeit nun genau auf die Fitnessauswertung angepasst werden, bei der im entsprechenden Lauf die Lösung gefunden wurde. Die in dieser Arbeit verwendete Methode zur Berechnung des Effort-Wertes verzichtet als Folge auf den Generationenbegriff und erlaubt, Läufe nach jeder beliebigen Fitnessauswertung abzubrechen. Auf diese Weise kann auf die Aufrundung des Bruches bei der *Effort*-Wert-Berechnung verzichtet werden.

Zur Berechnung der Anzahl benötigter Fitnessauswertungen wird folgende Formel zu Grunde gelegt:

$$1 - (1 - P\,(\mathit{eval}))^{\frac{\mathit{effort}}{\mathit{eval}}} \geq 0.99 \tag{5.2}$$

Hierbei steht *effort* für die gesuchte Gesamtzahl der Fitnessauswertungen, *eval* für die Anzahl der Auswertungen, nach denen ein Lauf abgebrochen wird und $P\,(\mathit{eval})$ für die Wahrscheinlichkeit, dass ein Lauf nach maximal *eval* Fitnessauswertungen eine Lösung gefunden hat. Da diese Wahrscheinlichkeit nicht bekannt ist, wird sie durch die relativen Häufigkeiten $\hat{P}\,(\mathit{eval})$ geschätzt, die auf den empirischen Daten der durchgeführten Testläufe beruhen. Auf die Größe der Fehler, die durch die Unterschiede zwischen P und $\hat{P}$ entstehen, wird in [21] eingegangen.

Durch Umformung ergibt sich aus (5.2):

$$\mathit{effort} = \min_{\mathit{eval}}\ \mathit{eval}\frac{\ln 0.01}{\ln\left(1 - \hat{P}\,(\mathit{eval})\right)} \tag{5.3}$$

Die Unterschiede zu Formel (5.1) liegen einerseits in der nicht mehr benötigten oberen Gauss-Klammer, anderseits darin, dass das Minimum nun über die Fitnessauswertungen und nicht über die Anzahl der Generationen berechnet wird.

5.4.2 Weitere statistische Probleme des *Effort*-Wertes

In [21] werden einige Ungenauigkeiten aufgezeigt, die sich bei der Berechnung des *Effort*-Wertes ergeben. In diesem Abschnitt werden zwei weitere Gesichtspunkte betrachtet, durch die der errechnete *Effort*-Wert zum Teil sehr stark vom theoretischen abweichen kann. Zum einen soll der Einfluss der Anzahl der Läufe untersucht werden, die in den durchgeführten Fitnessauswertungen zu keiner Lösung führten. Zum anderen soll der Einfluss der Verteilung der Anzahl der Fitnessauswertungen untersucht werden, die in den einzelnen Läufen zum Auffinden einer Lösung benötigt wurden.

Zur Veranschaulichung wird eine Schar imaginärer Optimierungsprobleme mit zwei freien Parametern, P_{fail} und sd betrachtet. Von den Problemen ist bekannt, dass sie jeweils mit einer Wahrscheinlichkeit von P_{fail} keine Lösung finden. Die Anzahl der Fitnessauswertungen, nach denen in den verbliebenen Läufen eine Lösung gefunden wird, sei $(100000, sd)$-normalverteilt. Die Annahme einer Normalverteilung wird bei vielen GP-Problemen nicht zutreffen, aber bereits anhand dieser relativ einfach zu untersuchenden Verteilung ergeben sich Probleme, die den *Effort*-Wert beeinflussen können. Da als Anzahl für Fitnessauswertungen nur natürliche Zahlen in Frage kommen, werden durch die Verteilung erhaltene Zahlen aufgerundet bzw. im negativen Fall verworfen und durch eine weitere Zufallszahl ersetzt.

Im Folgenden werden die theoretischen *Effort*-Werte für die Optimierungsprobleme berechnet. Hierbei wird der Übergang zu natürlichen Zahlen vernachlässigt, da sich empirisch ergeben hat, dass die dadurch verursachten Unterschiede unter 0.001% liegen.

Zunächst wird gezeigt, welchen Einfluss verschiedene Werte für P_{fail} und sd auf den *Effort*-Wert eines Problems haben können. Für P_{fail} werden die Werte $\{0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$ untersucht und für sd die Werte $\{1000, 2000, 5000, 10000, 11000, 20000, 30000\}$.

Für jede sich so ergebende Kombination wird der theoretische *Effort*-Wert mit Hilfe von Gleichung (5.3) und der Dichtefunktion einer $(100000, sd)$-Normalverteilung berechnet:

$$P(eval) = (1 - P_{fail}) \frac{1}{sd\sqrt{2\pi}} \int_0^{eval} \exp \frac{-(x - 100000)^2}{2sd^2} dx \tag{5.4}$$

Die Ergebnisse sind in Abbildung 5.9 aufgetragen.

Den geringsten *Effort*-Wert hat die Kombination (P_{fail} = 0.2, sd = 1000) mit 295701 Fitnessauswertungen, den größten die Kombination (P_{fail} = 0.8, sd = 30000) mit 3214423 Auswertungen, also mehr als dem Zehnfachen. Der Einfluss von P_{fail} ist dabei wesentlich höher, als jener der Standardabweichung. Dies ergibt sich ebenfalls direkt aus der Formel (5.3). Der Wert $P_{fail} < 1$ geht direkt in den Wert $\hat{P}$, der sich im Nenner der Formel logarithmisch auf den *Effort*-Wert auswirkt, so dass das Produkt für Werte nahe Eins entsprechend größer

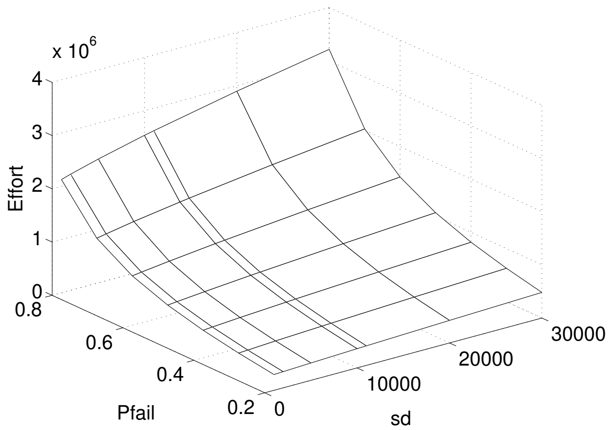


Abbildung 5.9: Die *Effort*-Werte für die 49 synthetisch erzeugten Optimierungsprobleme

wird. Die Veränderung der Standardabweichung wirkt sich hingegen in den meisten Fällen hauptsächlich auf eine Veränderung des linearen *eval*-Terms aus. So schwankt der Abbruchwert *eval* beispielsweise bei allen Problemen mit $P_{fail} = 0.2$ zwischen 103041 und 153192, wohingegen der zugehörige Wert $\hat{P}(eval)$ immer zwischen 0.769 und 0.799 liegt. Dies führt trotz der Logarithmierung nur zu relativ kleinen Änderungen des zweiten Faktors führt.

5.4.3 Der Einfluss von P_{fail} auf den *Effort*-Wert

Abbildung 5.9 zeigte bereits, dass unterschiedliche Werte von P_{fail} bei sonst gleicher Verteilung der Anzahlen der benötigten Fitnessauswertungen zu extrem unterschiedlichen *Effort*-Werten führen können. Hieraus ergibt sich ein Problem, wenn zur Berechnung der relativen Häufigkeiten nur eine sehr kleine Anzahl von Testläufen durchgeführt wird, zum Beispiel 50. In diesem Fall kann sich der Anteil der erfolgreichen Läufe zum Teil stark von der tatsächlichen Wahrscheinlichkeit $1 - P_{fail}$ unterscheiden. Entsprechend kann sich dann der *Effort*-Wert in Abbildung 5.9 entlang der P_{fail}-Achse verschieben.

Zur empirischen quantitativen Analyse dieses Effektes wurden für alle Werte von P_{fail} mit einer Standardabweichung von $sd = 1000$ jeweils 500 Läufe ausgeführt, die mit einer Wahrscheinlichkeit von P_{fail} zu keiner Lösung führen. Für die restlichen Läufe wurde jeweils die Anzahl der benötigten Fitnessberechnungen mit Hilfe einer (100000,*sd*)-Normalverteilung bestimmt. Die Werte wurden zur nächsten natürlichen Zahl aufgerundet, negative Zahlen wurden durch eine nochmalige zufällige Auswahl ausgetauscht. Für jeden dieser Läufe wird ein empirischer *Effort*-Wert $\widehat{effort}$ berechnet. Dieses Experiment wurde für jeden Wert P_{fail} tausendmal wiederholt.

Die Ergebnisse hierzu sind in der Tabelle 5.9 zusammengefasst. In der zweiten Spalte befindet sich der theoretische *Effort*-Wert. In der dritten steht der Durchschnittswert aller 1000 empirischen $\widehat{effort}$-Werte sowie das prozentuale Verhältnis zum theoretischen. In den letzten drei Spalten stehen der prozentuale Anteil des kleinsten und größten $\widehat{effort}$-Wertes im Vergleich zum theoretischen sowie die entsprechende Standardabweichung aller eintausend neuen Prozentwerte.

P_{fail}	*effort*	$\sum \widehat{effort}/1000$		min	max	sd
0.2	295701	294566	(99.6 %)	85.1 %	119.6 %	5.56
0.3	394995	393505	(99.6 %)	83.5 %	121.1 %	5.71
0.4	518746	515964	(99.5 %)	81.3 %	117.2 %	5.95
0.5	685480	683113	(99.7 %)	80.9 %	120.5 %	6.35
0.6	929849	927447	(99.7 %)	77.7 %	123.6 %	6.68
0.7	1331376	1330270	(99.9 %)	71.8 %	126.0 %	8.08
0.8	2127615	2128350	(100.0 %)	72.2 %	136.6 %	9.91

Tabelle 5.9: Werden statt der zuerst vorgegebenen 100000 Läufe nur 500 durchgeführt, liegt der entsprechende neue Effort-Wert $\widehat{effort}$ bei häufiger Wiederholung des Experiments nahe am ursprünglichen Wert, kann aber im Einzelfall deutlich abweichen.

Wie zu erwarten war, liegt der durchschnittliche $\widehat{effort}$-Wert, gemittelt über alle eintausend Testreihen zu je 500 Läufen, sehr nahe am theoretischen. Die letzten Spalten zeigen allerdings, dass es einzelne Testreihen gibt, die den *Effort*-Wert um bis zu 28 Prozent unter- oder um bis zu 36 Prozent überschätzen (beide Werte für $P_{fail} = 0.8$). Je größer P_{fail} wird, desto weiter liegt der empirische $\widehat{effort}$-Wert durchschnittlich vom theoretischen entfernt.

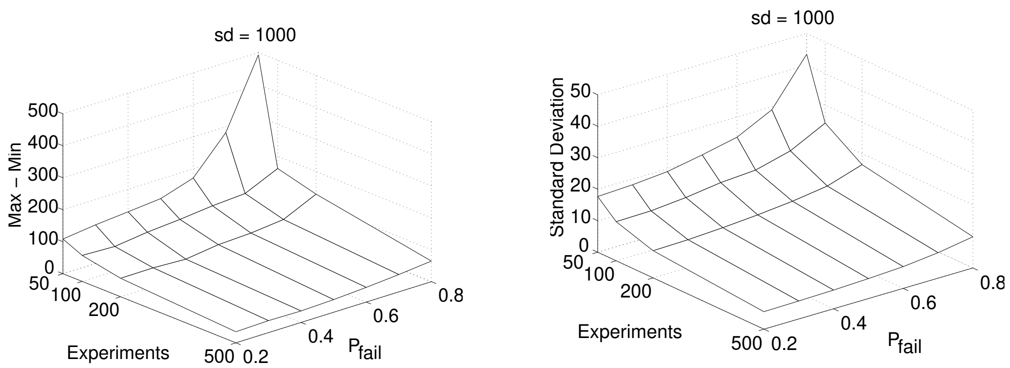


Abbildung 5.10: Die prozentuale Abweichung der mit wenigen Testläufen ermittelten $\widehat{effort}$-Werte vom theoretischen.

Noch deutlicher werden die Ergebnisse, wenn die Anzahl der Läufe in einer Testreihe reduziert wird. Die Experimente wurden mit 50, 100 und 200 Läufen pro Testreihe wiederholt und wiederum tausendmal durchgeführt. Die Ergebnisse sind in den Graphen der Abbildung 5.10 zusammengefasst. Auf der z-Achse des linken Diagramms sind hier die Differenzen der Prozentzahlen der Testreihe mit dem höchsten und niedrigsten $\widehat{effort}$-Wert abgetragen. Die vordere Kante parallel zur $P_{fail} - Achse$ entspricht somit den Differenzen aus der vierten und fünften Spalte der Tabelle 5.9.

Im rechten Diagramm sind die Standardabweichungen der einzelnen Testreihen zum durchschnittlichen $\widehat{effort}$-Wert eingezeichnet, die vordere Kante parallel zur P_{fail}-Achse entspricht somit der sechsten Spalte aus Tabelle 5.9.

Während der Graph bei 500 Testläufen relativ flach ist und die *Effort*-Ergebnisse der Stichprobe somit auch relativ nah am theoretischen Wert liegen, werden die Werte für geringere Laufzahlen deutlich schlechter. Der Effekt ist hierbei umso größer, je höher P_{fail} ist.

Bei 50 Testläufen und einer Fehlerwahrscheinlichkeit von $P_{fail} = 0.8$ ergab sich sogar in einem Fall ein $\widehat{effort}$-Wert von 11345515, dies sind mehr als 500 Prozent des theoretischen *Effort*s, also mehr als das Fünffache. In dieser Testreihe führten nicht zehn Läufe – wie es für $P_{fail} = 0.8$ wünschenswert wäre – zum Ziel, sondern nur zwei. Diese geringe relative Häufigkeit geht in der Formel (5.3) logarithmisch in den Nenner ein und bewirkt somit den hohen Wert.

Alle bisher vorgestellten Testreihen gingen von einer normalverteilten Grundgesamtheit der benötigten Fitnessauswertungen bei den erfolgreichen Läufen mit einer Standardabweichung von $sd = 1000$ aus. Die Experimente wurden für die Werte $sd = 2000, 5000, 10000, 11000, 20000$ und 30000 wiederholt und es zeigte sich, dass sich die Ergebnisse nicht signifikant von denen für $sd = 1000$ unterschieden. Einzelne Maximal- oder Minimalwerte diverser Testreihen können zwar von den Werten für $sd = 1000$ abweichen, die Standardabweichungen der Unterschiede zwischen empirischen $\widehat{effort} - Werten$ und den theoretischen unterscheiden sich jedoch nur geringfügig. Die Hauptursache für die abweichenden $\widehat{effort}$-Werte liegt somit in dem Unterschied zwischen den relativen Erfolgshäufigkeiten und dem tatsächlichen P_{fail}-Wert und nicht in der Verteilung der Anzahl benötigter Fitnessauswertungen.

5.4.4 Der Einfluss von sd auf den *Effort*-Wert

In Abschnitt 5.4.3 wurde gezeigt, welchen Einfluss die Differenz zwischen der tatsächlichen Erfolgswahrscheinlichkeit $1 - P_{fail}$ bei einem Testproblem und den in einer Testreihe gemessenen relativen Erfolgshäufigkeiten auf den gemessenen *Effort*-Wert haben kann. In diesen Abschnitt wird nun gezeigt, dass der in einer Testreihe gemessene $\widehat{effort}$-Wert auch vom theoretischen abweichen kann, wenn die relativen Häufigkeiten exakt der Erfolgswahrscheinlichkeit entsprechen.

An dieser Stelle soll empirisch der Einfluss der Verteilung untersucht werden, die sich durch die Anzahl der zum Finden einer Lösung benötigten Fitnessauswertungen in den einzelnen Läufen ergibt.

Es wird wieder die in Abschnitt 5.4.2 beschriebene Schar imaginärer Optimierungsprobleme mit den beiden Parametern $P_{fail} \in \{0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$ und $sd \in \{1000, 2000, 5000, 10000, 11000, 20000, 30000\}$ benutzt. Für jede Kombination werden erneut

Testreihen mit 50, 100, 200 und 500 Läufen durchgeführt. Dieses Mal wird der Erfolg einzelner Läufe nicht mit Hilfe der Wahrscheinlichkeit P_{fail} festgelegt. Die Anzahl der erfolgreichen Läufe bei $runs \in \{50, 100, 200, 500\}$ Experimenten wird vielmehr auf genau $(1 - P_{fail}) \times runs$ gesetzt. Die Anzahl der benötigten Fitnessauswertungen in den verbliebenen erfolgreichen Läufen wird über eine (100000,sd)-Normalverteilung bestimmt.

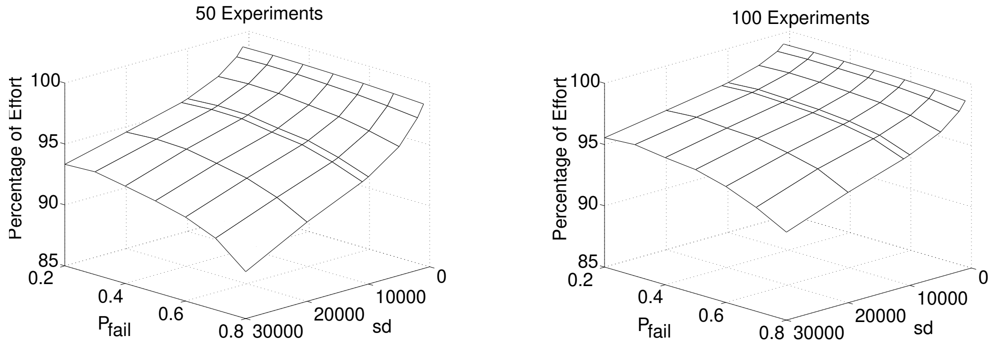


Abbildung 5.11: Der prozentuale Anteil der mit 50 bzw. 100 Testläufen ermittelten $\widehat{effort}$-Werte im Vergleich zu den theoretischen.

Zu jeder der genannten $(P_{fail}, sd, runs)$-Kombinationen wurde ein empirischer $\widehat{effort}$ berechnet. Dies wurde tausendmal wiederholt und aus diesen tausend neuen Werten wurde der Durchschnitt gebildet. In den Abbildungen 5.11 und 5.12 sind die Ergebnisse dieser Versuchsreihen zu sehen. Für die unterschiedlichen Laufanzahlen ist jeweils ein eigenes Diagramm abgebildet, die X-Achse und Y-Achse repräsentieren die Werte von *sd* und P_{fail}. Auf der Z-Achse ist der Durchschnittswert der zu einer Parameterkombination gehörenden tausend $\widehat{effort}$-Werte als Prozentangabe gegenüber dem theoretischen *Effort*-Wert angegeben.

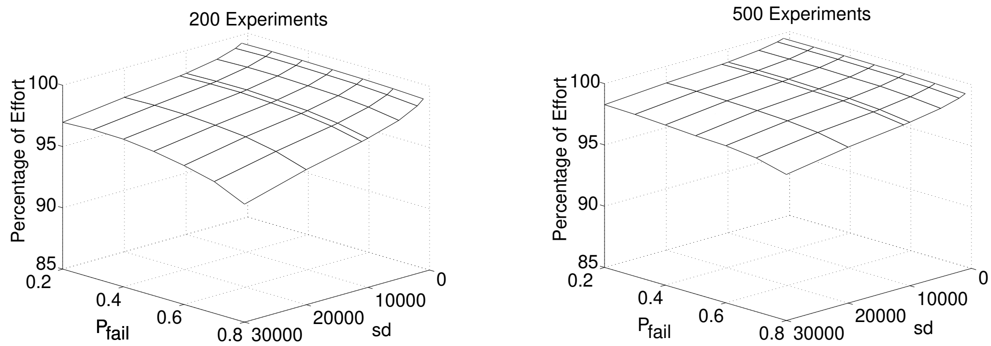


Abbildung 5.12: Der prozentuale Anteil der mit 200 bzw. 500 Testläufen ermittelten $\widehat{effort}$-Werte im Vergleich zu den theoretischen.

Der*Effort* bei dieser Problemstellung wird im Durchschnitt leicht unterschätzt. Diese Ungenauigkeit wird umso deutlicher, je größer *sd* und P_{fail} werden. Während für kleinere *sd*-Werte P_{fail} nur geringe Auswirkung auf das Resultat hat, wird dieser Einfluss mit wachsendem *sd* größer. Je mehr Läufe gemacht werden, desto weniger wird der Wert unterschätzt.

Die Ursache für die Unterschätzung liegt darin, dass bei der benutzten Normalverteilung der Fitnessauswertungszahlen in Formel (5.3) das Minimum in den meisten Fällen für den größten aller *eval*-Werte einer Testreihe angenommen wird.[1] Im Nenner des Bruchs steht in diesem Fall für $(1-\hat{P}(eval))$ der Wert P_{fail}. Bei der Berechnung des theoretischen Wertes wird ein ähnlich großer Wert im Logarithmus des Nenners erst für größere *eval*-Werte erreicht. Für kleinere *eval*-Werte wirkt sich die Änderung des Nenners von Formel (5.3) stärker aus als der Term *eval* selbst, so dass diese Werte bei der Minimumberechnung in fast allen Fällen keine Rolle spielen. Durch die diskrete Anzahl der Experimente ist $\hat{P}(eval)$ für den größten vorliegenden *eval*-Wert durchschnittlich höher als der über Formel (5.4) zugehörige theoretische Wert, was zu einem geringeren empirischen *Effort* führt. Je geringer die Anzahl der Experimente, desto größer wird im Durchschnitt der Unterschied zwischen $\hat{P}(eval)$ und $P(eval)$.

5.4.5 Zusammenfassung

In diesem Abschnitt wurde die Berechnung des *Effort*-Wertes beschrieben und gezeigt, wie sie für das in dieser Arbeit benutzte GP-System abgewandelt werden kann.

Bei der Berechnung des *Efforts* kann es zu verschiedenen, statistischen Ungenauigkeiten kommen. Es wurden zwei Ursachen für die Berechnung falscher Werte vorgestellt.

Das erste Beispiel zeigte, dass es durch den Unterschied zwischen Wahrscheinlichkeiten P_{fail} und den als Schätzer benutzten relativen Häufigkeiten $\hat{P}_{fail}$ bei Testreihen mit wenigen Experimenten zu großen Unterschieden zwischen theoretischem *Effort* und empirisch ermitteltem Wert kommen kann.

Im zweiten Beispiel wurde gezeigt, dass die Verteilung der Ergebnisse einer Testreihe grundsätzlich zu einem falschen *Effort*-Wert führen kann. Wenn der *Effort*-Wert einer Testreihe mit dem einer anderen verglichen werden soll, kann eine unterschiedliche Verteilung der benötigten Funktionsauswertungsanzahlen bereits zu divergierenden Ergebnissen führen, selbst wenn die Problemstellungen, die den beiden Testreihen zu Grunde liegen, denselben theoretischen *Effort*-Wert haben. Dieser Unterschied lässt sich auch nicht durch mehrmalige Wiederholung der Testreihen ausschließen.

Der Aussagekraft des *Effort*-Wertes sollte demnach nicht zu hoch bewertet werden (siehe auch [21]) und auf keinen Fall als ausschließliches Kriterium für die Bewertung eines Verfahrens herangezogen werden. Weitere Kriterien sollten immer auch die relative Anzahl der gescheiterten Läufe $\hat{P}_{fail}$ sowie die Art der Verteilung bei den erfolgreichen Läufen sein.

Die Testreihen haben gezeigt, dass bei nur 50 Testläufen der Effort-Wert zu stark von statistischen Ungenauigkeiten abhängt. Oftmals lässt sich ein Vergleich mittels *Effort*-Wert nicht vermeiden, da in vielen bereits erschienenen Puplikationen keine anderen Vergleichsstatistiken veröffentlicht wurden. In solchen Fällen sollte darauf geachtet werden, dass die Anzahl der Testläufe bei allen Verfahren ungefähr gleichhoch ist und deutlich über 50 liegt.

[1] Für die Testreihen mit 50 Läufen und $sd < 30000$ gab es im Durchschnitt immer weniger als einen *eval*-Wert, der größer als jener war, bei dem das Minimum angenommen wurde.

In dieser Arbeit werden zu jedem Problem mindestens zweihundert Läufe durchgeführt. Trotzdem wird vermieden, ein Verfahren nur aufgrund eines etwas niedrigeren *Effort*-Wertes gegenüber einem anderen als überlegen zu bezeichnen. Ausschlaggebend hierfür wäre vielmehr eine deutlich höherer Prozentsatz von erfolgreichen Testläufen, die ein Optimum finden können. Bei ähnlichem Prozentsatz wäre von zwei Verfahren dasjenige besser, welches im Durchschnitt wesentlich weniger Funktionsauswertungen benötigt als das andere.

Kapitel 6

Untersuchungen zum Crossover

In Abschnitt 4.2.9.3 wird beschrieben, wie bei *randomCrossover* die Ein- und Ausgangskanten des neu einzufügenden Teilgraphen zufällig mit den entsprechenden freien Kanten des Hauptgraphen (siehe Definition 4.4) kombiniert werden. Bei diesem Ansatz kann es passieren, dass ungültige Graphen erzeugt werden, die entweder nicht der Definition von GP-Graphen genügen oder – bei vorausgesetzten azyklischen Graphen – Zyklen beinhalten.

In den folgenden Abschnitten wird analysiert, wo die Grenzen dieses Ansatzes liegen. Zusätzlich werden Verbesserungen des Zuordnungsalgorithmus vorgestellt, mit denen das Fehlverhalten kompensiert werden kann.

6.1 Vorgehensweise

Ziel eines *Crossover*-Operators ist es, aus zwei GP-Graphen einen neuen Graphen zu erzeugen, der der GP-Graph-Definition 3.1 genügt, zur Semantik des Problems passt und zusätzliche Voraussetzungen der Problemstellung erfüllt. Dies kann zum Beispiel die Forderung nach azyklischen Graphen sein.

Je öfter ein *Crossover*-Operator dieses Ziel erfüllen kann, desto geeigneter ist er für den Einsatz im GP-System *GGP*. Der in Abschnitt 4.2.9 vorgestellte Operator *randomCrossover* erreicht dieses Ziel nicht in allen Fällen, was in einer ersten Verbesserung dazu führte, fünf Crossover-Versuche mit denselben Graphen zuzulassen, bevor die Operation als gescheitert gilt.

Die Qualität des *randomCrossover*-Operators wurde empirisch untersucht, indem er auf viele zufällig erzeugte GP-Graphen angewandt wurde. Es wurde gezählt, wie viele dieser Anwendungen erfolgreich waren. Hierbei wurde darauf geachtet, dass die Diversität der getesteten GP-Graphen möglichst hoch ist. So wurden mehrere verschiedene Graphgrößen betrachtet und zwischen azyklischen und zyklischen Graphstrukturen sowie Baumstrukturen unterschieden.

Die Experimente haben gezeigt, dass der ursprüngliche Operator – je nach Problemklasse und Graphgröße – in bis zu 80 Prozent aller Fälle beim ersten Rekombinationsversuch keinen Graphen erzeugen kann, welcher der Aufgabenstellung entspricht. Die Fehlerrate steigt dabei mit der Größe der Graphen an. Wurden des Weiteren azyklische Graphen gefordert, führte dies zu einer weiteren Erhöhung der Fehlerrate.

Aufgrund dieser Erkenntnisse wurde zuerst ein neuer *Crossover*-Operator entwickelt, der bereits bei der Verknüpfung eines Teilgraphen mit dem Hauptgraphen des anderen Individuums konstruktiv auf die Einhaltung der geforderten Bedingungen achtet. Danach folgte ein weiterer Operator, der speziell im azyklischen Fall für gute Ergebnisse sorgt, indem er primär auf Zyklenvermeidung achtet.

Im Folgenden werden zunächst die neuen *Crossover*-Varianten eingeführt. Daran anschließend werden in Abschnitt 6.4 die experimentellen Analysen aller Operatoren gemeinsam vorgestellt. Hierbei wird deutlich, wo die Unterschiede zwischen den einzelnen Operatoren liegen und welche für unterschiedliche Problemtypen primär eingesetzt werden sollten.

6.2 Ein allgemeiner, konstruktiv arbeitender Operator – *generalCrossover*

Der Operator *randomCrossover* verbindet durch eine randomisierte Zuordnung die freien Kanten des Hauptgraphen mit den Eingangs- und Ausgangskanten des Teilgraphen. Für den neu entstandenen Graphen muss untersucht werden, ob er die geforderten Bedingungen aus Definition 3.1 erfüllt und, falls gefordert, zyklenfrei ist. Ist dies nicht der Fall, wird die ganze Rekombination wiederholt. Passiert dies fünf Mal, wird statt eines Crossovers eine Replikation des Individuums mit der besseren Fitness durchgeführt.

Die Analysen in Abschnitt 6.4 zeigen, dass diese Implementierung für Baumstrukturen immer und für zyklische Graphen in einer ausreichenden Zahl von Fällen zu GP-Graphen führt. Bei Problemen mit azyklischer Graphstruktur werden bei größer werdenden Graphen jedoch immer öfter keine verwendbaren Graphen gefunden, so dass ein alternativer Ansatz gefunden werden muss.

In diesem Abschnitt wird ein neuer sowohl auf zyklischen als auch auf azyklischen Graphen anwendbarer *Crossover*-Operator vorgestellt. Der einzige Unterschied zu *randomCrossover* liegt in einer anderen Art der Kantenzuordnung der freien Kanten des Hauptgraphen zu denen des Teilgraphen. Bei der Zuordnung wird besonders beachtet, dass die Erfüllung von Definition 3.1 forciert wird. Da der neue Operator sowohl für azyklische als auch für zyklische Problemstellungen eingesetzt werden kann, wird er *generalCrossover* genannt.

Die Menge aller GP-Graphen, die *generalCrossover* erzeugen kann, ist eine Teilmenge der Graphen von *randomCrossover*. Somit sind alle GP-Graph-Eigenschaften, die durch *randomCrossover* implizit erfüllt werden, auch bei der Verwendung von *generalCrossover* gegeben.

Zusätzlich müssen neue Graphen die Punkte 7 und 8 der GP-Graph-Definition erfüllen. Diese Eigenschaften werden zunächst für bestimmte Knoten des Hauptgraphen untersucht. Es wird sich zeigen, dass die beiden Punkte für den gesamten neuen GP-Graphen erfüllt sind, wenn sie für die Randknoten des Hauptgraphen (Definition 4.4) nachgewiesen werden können.

Satz 6.1
Nach Entfernen eines Teilgraphen aus einem GP-Graphen (I, O, V, E) *gilt für alle verbliebenen Knoten* $\hat{V} \subset V$*:*

1. *Zu jedem Element* $v \in \hat{V}$ *existiert ein Verbindung zu einem Element* $o \in O$ *oder zu einer freien Kante, die in den entfernten Teilgraphen führte.*
2. *Zu jedem Element* $v \in \hat{V}$ *existiert mindestens eine Verbindung von einem Knoten* $i \in I \cup V$ *mit Eingangsgrad 0 oder von einer freien Kante, die aus dem entfernten Teilgraphen herausführte.*

Beweis
Nach Punkt 7 der Definition 3.1 führt vor dem Entfernen eines Teilgraphen von jedem Knoten $v \in V$ ein GP-Pfad zu einem Knoten $o \in O$. Wird dieser Pfad durch das Entfernen eines Teilgraphen unterbrochen, so kann dieser nur über eine der nun freien Kanten in den Teilgraph geführt haben. Gleiches gilt für Verbindungen von Eingangs- bzw. (0,1)-Knoten zu den einzelnen Knoten des Hauptgraphen. ■

Das Ziel einer konstruktiven Kantenzuordnung lässt sich somit auf folgende Weise formulieren: Allen Knoten des Hauptgraphen müssen durch Einfügen des neuen Teilgraphen wieder über die oben beschriebenen Verbindungen verfügen. Für azyklische Problemstellungen muss weiterhin das Verhindern von Zyklen beachtet werden.

Das Problem der fehlenden Verbindungen von beliebigen Knoten des verbliebenen Hauptgraphen lässt sich auf die fehlenden Verbindungen der Randknoten reduzieren.

Satz 6.2
Sind die Eigenschaften sieben und acht von Definition 3.1 nach dem Einfügen des neuen Teilgraphen für alle Randknoten erfüllt, so sind sie es für alle Knoten des ehemaligen Hauptgraphen.

Beweis
Verfügt ein Knoten des Hauptgraphen über keine Verbindung zu einem Ausgangsknoten, so weist er nach Satz 6.1 statt dessen eine Verbindung zu einer freien Kante und somit zu einem vorderen Randknoten des Hauptgraphens auf. Wenn dieser nach Einsetzen des neuen Teilgraphen über eine Verbindung zu einem Ausgangsknoten verfügt, gilt dies automatisch auch für den betrachteten Knoten. Die gleiche Argumentation lässt sich analog auf fehlende Verbindungen von Eingangs-/(0,1)-Knoten anwenden. ■

Wird durch das Einfügen des neuen Teilgraphen ein vorderer Randknoten mit einem hinteren verbunden, bedeutet dies, dass der vordere Randknoten genau dann eine Verbindung zu einem Ausgangsknoten erhält, wenn der hintere Randknoten bereits über eine verfügte. Andererseits wird eine Verbindung von einem (0,1)-Knoten oder einem Eingangsknoten zu dem hinteren Randknoten erzeugt, wenn der vordere Randknoten über eine solche Verbindung verfügte. Zusätzlich kann eine Verbindung von einem (0,1)-Konten zum hinteren Randknoten erzeugt werden, wenn innerhalb des einzufügenden Teilgraphen ein (0,1)-Knoten existiert, der eine Verbindung zu der Ausgangskante des Teilgraphen hat, die mit dem hinteren Randknoten verbunden wird.

Ein vorderer Randknoten v_v ohne Verbindung von einem Eingangs- oder (0,1)-Knoten kann nur dann eine solche erhalten, wenn eine Verbindung von einem hinteren Randknoten v_h zu diesem führt und v_h über einen neuen Kantenzug durch den Teilgraphen, eine Verbindung von einem Eingangs- oder (0,1)-Knoten zugeordnet wird. Der Knoten v_h kann vorher noch keine Verbindung dieser Art besessen haben, da sonst v_v automatisch eine entsprechende gehabt hätte.

Analog kann ein hinterer Randknoten nur eine neue Verbindung zu einem Ausgangsknoten erhalten, wenn eine Verbindung zu einem vorderen Randknoten v_v existiert und diesem eine entsprechende Verbindung zugeordnet wird.

Hinreichende Bedingung für die Erfüllung der Punkte sieben und acht der GP-Graph-Definition 3.1 ist somit, dass alle vorderen Randknoten eine Verbindung zu einem Ausgangsknoten erhalten und alle hinteren Randknoten eine Verbindung von einem Eingangs- oder (0,1)-Knoten. Die restlichen notwendigen Verbindungen ergeben sich somit automatisch.

Außerdem muss neben den Verbindungen der Randknoten bekannt sein, zwischen welchen Ein- und Ausgangskanten des einzufügenden Teilgraphen Verbindungen existieren. Anhand des folgenden Beispiels soll die hinreichende Bedingung verdeutlicht werden.

Zur Nutzung der Erkenntnisse aus Satz 6.2 bei der Erzeugung eines GP-Graphen müssen somit folgende Informationen zur Verfügung stehen:

- Welche Randknoten des Hauptgraphen haben Verbindungen von (0,1)-Knoten oder Eingangsknoten?
- Welche Randknoten des Hauptgraphen haben Verbindungen zu Ausgangsknoten?
- Welche Verbindungen existieren zwischen hinteren und vorderen Randknoten des Hauptgraphen?
- Welche Verbindungen existieren zwischen Eingangskanten und Ausgangskanten des neuen Teilgraphen?

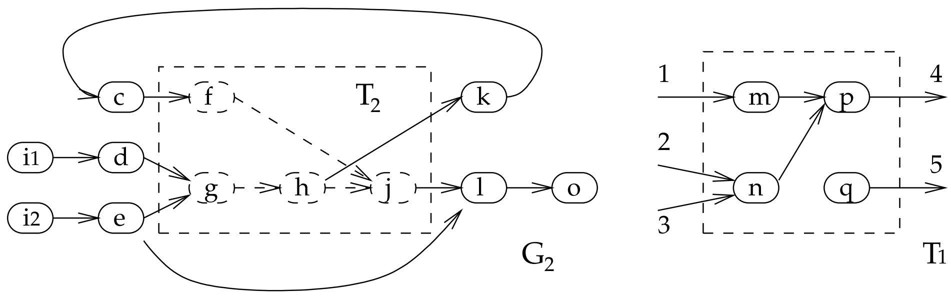


Abbildung 6.1: Der Teilgraph T_1 soll an Stelle des Teilgraphs T_2 so in G_2 eingefügt werden, dass Definition 3.1 erfüllt ist.

Beispiel 6.1
Abbildung 6.1 zeigt einen GP-Graph G_2, dessen Teilgraph T_2 gegen den Teilgraph T_1 ausgetauscht werden soll.

Für die Randknoten $\{c, d, e, k, l\}$ des Graphen ergeben sich nach der Entfernung von T_2 die in Tabelle 6.1 aufgeführten Verbindungen zu Eingangs- und Ausgangsknoten und untereinander. Die Kante (e, l) spielt für das Propagieren von Ein- und Ausgangsverbindungen keine Rolle und wird deshalb in der Tabelle nicht berücksichtigt.

Knoten	Ein	Aus	Rand
c			
d	ja		
e	ja	ja	
k			c
l	ja	ja	

	4	5
1	+	
2	+	
3	+	

Tabelle 6.1: Die linke Tabelle gibt die Verbindungen der Randknoten von G_2 aus Abb. 6.1 wieder. Die Spalte *Ein* sagt aus, ob eine Verbindung von einem Eingangsknoten oder einem (0,1)-Knoten zum Knoten der ersten Spalte besteht. *Aus* zeigt eine Verbindung zu einem Ausgangsknoten an und *Rand* gibt an, zu welchen anderen Randknoten eine Verbindung existiert. Da Verbindungen von vorderen zu hinteren Randknoten keine Rolle spielen, werden sie nicht mit aufgeführt. Die rechte Tabelle zeigt, zwischen welchen Eingangs- und Ausgangskanten von T_1 Verbindungen bestehen.

Aus der Tabelle ergibt sich, dass c, d und k eine Verbindung zu einem Ausgangsknoten benötigen. Gemäß der hinreichenden Bedingung genügt es, bei c und d auf entsprechende Verbindungen zu achten, da eine Verbindung für c automatisch durch die Kante (k, c) eine Verbindung für k zur Folge hat. Andererseits muss eine Verbindung von i_1 oder i_2 zum hinteren Randknoten k hergestellt werden. Hierdurch erhält automatisch der vordere Randknoten c eine entsprechende Verbindung.

Die gestellten Forderungen können erfüllt werden, wenn die Ausgangskante 5 des Teilgraphen mit Knoten k und die Kante 4 mit dem Knoten l verbunden werden. Die Zuordnung der Eingangskanten zu den freien Kanten der vorderen Randknoten ist beliebig.

Wenn bekannt ist, welche Randknoten über Verbindungen zu Eingangs-/(0,1)-Knoten und Ausgangsknoten verfügen und zwischen welchen Eingangs- und Ausgangskanten des neuen Teilgraphen Verbindungen existieren, kann eine Zuordnung zwischen den Eingangs- und Ausgangskanten des Teilgraphen und den freien Kanten des Hauptgraphen erfolgen.

Hierzu wird zuerst ein hinterer Randknoten ausgesucht, der noch über freie Kanten verfügt. Im zweiten Schritt wird diesem Knoten eine Ausgangskante des Teilgraphen zugeordnet. Diese Auswahl unterliegt bestimmten Regeln, die im Folgenden näher erläutert werden. Anschließend wird der Vorgang alternierend für vordere und hintere Randknoten wiederholt. Wenn nur noch vordere oder hintere Randknoten mit freien, noch nicht zugeordneten Kanten übrig sind, werden diese nacheinander abgearbeitet.

Im nächsten Abschnitt wird aufgeführt, nach welchen Regeln Knoten und Kanten ausgewählt werden.

6.2.1 IN- und OUT-Flags

Um die Regeln textlich übersichtlicher zu gestalten, werden sowohl allen Randknoten als auch den Ein- und Ausgangskanten des Teilgraphen die zwei Flags IN und OUT zugeordnet. Das IN-Flag steht für eine vorhandene Verbindung von einem Eingangs- oder einem (0,1)-Knoten, das OUT-Flag für eine Verbindung zu einem Ausgangsknoten. Zu Beginn der Knotenauswahl werden folgende Flags gesetzt:

- Das IN-Flag eines Randknotens wird gesetzt, wenn eine Verbindung von einem Eingangsknoten oder einem (0,1)-Knoten des Hauptgraphen zu diesem besteht, die nicht durch den entfernten Teilgraphen unterbrochen wurde.
- Das OUT-Flag eines Randknotens wird gesetzt, wenn von diesem eine Verbindung zu einem Ausgangsknoten des Hauptgraphen existiert, die nicht durch den entfernten Teilgraphen unterbrochen wurde.
- Das IN-Flag einer Ausgangskante des Teilgraphen wird gesetzt, wenn innerhalb des neuen Teilgraphen ein Kantenzug von einem (0,1)-Knoten zu diesem führt.

Anhand der Flags der Randknoten ist somit ersichtlich, über welche Verbindungen sie bereits verfügen und welche noch hergestellt werden müssen. Besitzt zum Beispiel eine Ausgangskante ein IN-Flag, so erhält sowohl eine freie Kante, die dieser zugewiesen wird, als auch der zu dieser Kante gehörende hintere Randknoten das IN-Flag.

6.2.2 Propagieren der Flags

Wird ein vorderer Randknoten mit IN durch das Einbinden des neuen Teilgraphen mit einem hinteren Randknoten ohne IN, dafür aber mit OUT-Flag verbunden, so erhält der hintere Randknoten ein IN-Flag und der vordere ein OUT-Flag. Existiert nämlich eine Verbindung von einem vorderen Randknoten zu einem hinteren und von diesem eine Verbindung zu einem Ausgangsknoten, so besitzt auch der vordere Randknoten aufgrund der Transitivität eine Verbindung zu diesem Ausgangsknoten – das OUT-Flag wird gesetzt. Analoges gilt für das IN-Flag.

Im Einzelnen wird ein OUT-Flag eines hinteren Randknoten zuerst an eine Ausgangskante des Teilgraphen weitergegeben, die einer freien Kante dieses Knotens zugeordnet wird. Hierdurch wird das OUT-Flag an alle Eingangskanten des Teilgraphen weitergereicht, von denen eine Verbindung innerhalb des Teilgraphen zu dieser Ausgangskante führt. Wird einer dieser Eingangskanten eine freie Kante des Hauptgraphen zugeordnet, erhält der zugehörige vordere Randknoten ebenfalls das OUT-Flag.

Bei einem IN-Flag eines vorderen Randknotens ist der Weg genau entgegengesetzt.

Bezogen auf das Beispiel 6.1 ergibt sich, dass für den hinteren Randknoten l das OUT-Flag gesetzt ist. Durch Zuordnung der Ausgangskante 4 zu der freien Kante von l erhalten 4 und damit auch (nach Tabelle 6.1 rechts) 1, 2 und 3 das OUT-Flag. Durch die Zuordnung dieser vorderen Eingangskanten zu den freien Kanten von c, d und e, besitzen nun auch die vorderen Randknoten das OUT-Flag und somit eine Verbindung zu einem Ausgangsknoten des Graphen.

6.2.3 Prioritäten bei der Auswahl der Randknoten

Die nachfolgende Aufzählung gibt an, in welcher Reihenfolge die Randknoten bearbeitet werden. Die Liste wird vom ersten bis zum letzten Punkt durchgegangen, bis ein Randknoten gefunden wurde, der die geforderten Flags besitzt. Die Aufzählung bezieht sich auf die **hinteren** Randknoten. Das Vorgehen für vordere Randknoten ist entsprechend. Hierbei sind die Aussagen bezüglich der IN- und OUT-Flags zu vertauschen.

1. Zuerst werden hintere Randknoten mit OUT und ohne IN gesucht.

2. Als zweites werden hintere Randknoten mit OUT und IN verwendet.

3. Wurde zu beiden Punkten kein passender hinterer Randknoten gefunden, wird statt dessen ein vorderer Randknoten mit den entsprechenden ersten beiden Prioritätsstufen gesucht.

4. Falls sowohl bei den hinteren als auch bei den vorderen Randknoten keine Knoten gefunden wurden, die den beiden Prioritätsstufen (1) oder (2) entsprechen, wird ein hinterer Randknoten ohne OUT und ohne IN benutzt.

Motiviert wird die Auswahlreihenfolge durch folgende Überlegungen:

Es ist einleuchtend, zuerst hintere Randknoten mit OUT-Flag zu verwenden, da die primäre Aufgabe darin besteht, diese Flags an die vorderen Randknoten weiterzugeben, die noch keine Verbindung zu einem Ausgangsknoten haben. Das OUT-Flag eines hinteren Randknotens kann nur dann an einen vorderen übertragen werden, wenn der hintere Randknoten über eine freie Kante mit einer Ausgangskante des Teilgraphen verbunden wird, die wiederum eine Verbindung von einer Eingangskante besitzt. Diese Eingangskante muss einer freien Kante des vorderen Randknoten zugeordnet sein.

Hintere Randknoten können nur dann ein fehlendes OUT-Flag erhalten, wenn von ihnen eine Verbindung zu einem vorderen Randknoten führt. Sobald dieser ein OUT-Flag erhält, wird es über die Verbindung zum hinteren Randknoten propagiert. So kann das Verbinden von hinteren Randknoten mit OUT-Flag mit dem Teilgraphen und somit auch mit den vorderen Randknoten sowohl bei den vorderen als auch bei den verbliebenen hinteren Randknoten zum Setzen von weiteren OUT-Flags führen.

Hintere Randknoten ohne IN haben eine höhere Priorität als Randknoten mit IN. Da beim Auswählen der vorderen Randknoten solche mit IN-Flag bevorzugt werden, verfügen die meisten Ausgangskanten des Teilgraphen, die über die Eingangskanten bereits mit dem Hauptgraphen verbunden sind, über ein IN-Flag. Es liegt somit nahe, diese Kanten mit Randknoten zu verbinden, die genau dieses Flag benötigen.

Es kann trotzdem zu Fällen kommen, in denen ein hinterer Randknoten verbunden werden muss, obwohl er noch über kein OUT-Flag verfügt. Abbildung 6.2 zeigt eine entsprechende Situation. Für diese Ausnahmen ist der vierte Punkt der Prioritätenliste vorhanden.

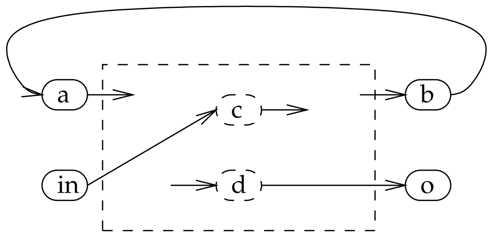


Abbildung 6.2: Der Graph kann nur zu einem GP-Graphen verbunden werden, wenn der hintere Randknoten *b* ausgewählt wird, obwohl er kein OUT-Flag besitzt hat.

6.2.4 Auswahl einer zum Randknoten passenden Ausgangskante

Sobald feststeht, welcher Randknoten als nächstes mit dem neuen Teilgraphen verbunden werden soll, wird eine passende Eingangs- oder Ausgangskante des Teilgraphen ausgewählt. Die Auswahl wird durch die Flags bestimmt, die der Randknoten bereits besitzt.

Wie schon bei der Auswahl der Randknoten werden auch hier Kanten, die bestimmte Flags besitzen, bevorzugt. In Tabelle 6.2 ist aufgelistet, welche Flagkombinationen bei den Kanten für die verschiedenen Randknoten favorisiert werden.

Randknoten	Priorität 1	Priorität 2	Priorität 3
hinten OUT ~~IN~~	Aus verbunden IN	Aus ~~verbunden IN~~	Aus ~~verbunden~~ IN
hinten OUT IN	Aus verbunden ~~IN~~	Aus ~~verbunden IN~~	Aus verbunden IN
hinten ~~OUT IN~~	Aus verbunden IN	Aus ~~verbunden~~ IN	
vorne IN ~~OUT~~	Ein verbunden OUT	Ein ~~verbunden~~	
vorne IN OUT	Ein verbunden ~~OUT~~	Ein ~~verbunden~~	Ein verbunden OUT
vorne ~~OUT IN~~	Ein verbunden OUT		

Tabelle 6.2: Auswahlpräferenzen für Eingangs- und Ausgangskanten

Die erste Spalte gibt jeweils an, auf welche Randknoten sich die entsprechende Zeile bezieht. Steht hier zum Beispiel *hinten* OUT ~~IN~~, so gilt die Zeile für einen hinteren Randknoten, der ein OUT-Flag, aber kein IN-Flag besitzt. In den folgenden Spalten einer Zeile stehen die Flag-Kombinationen, die eine Kante aufweisen muss, um ausgewählt zu werden. Zuerst wird demnach eine Kante gesucht, die der Beschreibung in der Spalte *Priorität 2* entspricht. Existiert keine solche Kante, wird nach einer Kante gesucht, die zur Spalte *Priorität 3* passt, usw. Wurde bis zur letzten Prioritätstufe keine passende Kante gefunden, gilt der Graphaufbau als gescheitert, der *Crossover*-Operator hat keinen GP-Graphen erzeugt.

Die Abkürzungen in der Tabelle haben folgende Bedeutung:

hinten/vorne: hinterer/vorderer Randknoten

OUT/~~OUT~~: OUT-Flag gesetzt/nicht gesetzt

IN/~~IN~~: IN-Flag gesetzt/nicht gesetzt

Aus/Ein: Gesucht wird eine Ausgangskante/Eingangskante

verbunden: Die gesuchte Kante wurde bisher noch keiner freien Kante des Hauptgraphen zugeordnet. Allerdings wurde bereits eine Kante am anderen Ende des Teilgraphen einer freien Kante des Hauptgraphen zugeordnet und zwischen diesen beiden existiert eine Verbindung innerhalb des Teilgraphen. Kurz zusammengefasst bedeutet dies, dass durch die Verwendung einer Kante mit der Eigenschaft *verbunden* im Hauptgraphen eine Verbindung zwischen (mindestens) zwei Randknoten erzeugt wird. Abbildung 6.3 verdeutlicht dies.

~~verbunden~~: Alle Kanten am anderen Ende des Teilgraphen, die eine Verbindung zur angegebenen Kante haben, wurden noch nicht zugeordnet.

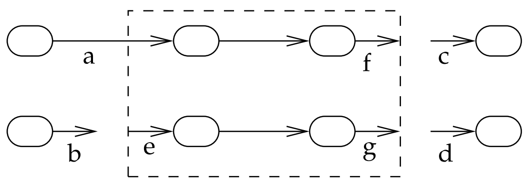


Abbildung 6.3: Die noch nicht zugeordnete Ausgangskante f des Teilgraphen besitzt die Eigenschaft *verbunden*, da sie über die Eingangskante a bereits eine Verbindung zum Hauptgraphen besitzt. Die Ausgangskante g hingegen gilt als *~~verbunden~~*, also nicht verbunden, da alle Eingangskanten des Teilgraphen, die zu g führen, ebenfalls noch keine Verbindung vom Hauptgraphen besitzen.

Die Eigenschaft *verbunden*/*~~verbunden~~* wird berücksichtigt, um eine passendere Zuordnung der Kanten zu ermöglichen. Dies soll an einem Beispiel für die zweite Zeile der Tabelle 6.2 näher erläutert werden.

Beispiel 6.2
Ausgehend von der Wahl eines hinteren Randknotens mit OUT*-Flag und* IN*-Flag, wird zunächst eine Ausgangskante gesucht, die schon mit mindestens einem vorderen Randknoten verbunden ist, aber über kein* IN*-Flag verfügt. Da der Ausgangsknoten über ein* IN*-Flag verfügt, wäre es unüberlegt, eine Kante mit* IN*-Flag zu verwenden, da diese später für einen Knoten ohne* IN *genutzt werden kann.*

Wurde keine nach Spalte 1 geeignete Kante gefunden, wird eine Ausgangskante ohne IN *gesucht, die noch nicht mit dem Hauptgraphen verbunden ist. Auf diese Weise wird ebenfalls kein* IN*-Flag unnötig verwendet, das für andere hintere Randknoten in Frage käme. Des Weiteren wird das* OUT*-Flag an mindestens eine Eingangskante des Teilgraphen weitergegeben.*

Konnte keine geeignete Kante ohne IN*-Flag gefunden werden, wird eine verbundene Ausgangskante mit* IN *benutzt. Dieses* IN *steht in diesem Fall nicht mehr für andere hintere Randknoten zur Verfügung.*

Neben den Bedingungen aus Tabelle 6.2 müssen bei jeder Kantenauswahl noch weitere Kriterien überprüft werden.

- Werden in der Problemstellung explizit azyklische GP-Graphen verlangt, so muss bei der Auswahl der Kanten darauf geachtet werden, dass durch die Zuordnung einer Kante des Teilgraphen zu einer freien Kante des Hauptgraphen in diesem kein Zyklus entsteht. Hierzu wird während der Erstellung des neuen GP-Graphen eine Adjazenzmatrix für die transitive Hülle aller Randknoten und ihrer Verbindungen mitgeführt. Wenn zwischen den Randknoten i und j durch die Zuordnung der Kanten eine Verbindung erzeugt werden soll, muss somit nur sichergestellt sein, dass noch keine Verbindung von j nach i existiert.

- Es können Situationen auftreten, in denen das letzte IN- oder OUT-Flag durch das Zuordnen einer Kante aufgebraucht würde, obwohl noch weitere Randknoten dieses Flag benötigen. In diesem Fall könnte – bei beliebigen weiteren Zuordnungen – kein GP-Graph mehr erzeugt werden. Aus diesen Grund wird bei jeder Auswahl einer Kante überprüft, ob nach der Zuordnung noch alle benötigten Flags verfügbar sind.

6.2.5 Nutzen des *generalCrossover*-Operators

Zum Testen des Operators wurden zuerst drei verschiedene Benchmarks verwendet – jeweils ein Benchmark mit Bäumen, azyklischen Graphen und zyklischen Graphen als zugrunde liegender Datenstruktur. Während der GP-Läufe wurde mitprotokolliert, wie oft der *Crossover*-Operator aufgerufen wurde und wie oft er dabei keine GP-Graphen erzeugen konnte, die alle Forderungen erfüllten[1]. Während in Abschnitt 6.4 die genaue Vorgehens-

[1] Wie auch der *randomCrossover*-Operator wird bei einem Scheitern im ersten Versuch der Operator insgesamt fünf Mal mit denselben Eltern aufgerufen. Konnte in keinen dieser Versuche ein geforderter GP-Graph erzeugt werden, gilt der Crossover-Versuch als gescheitert.

weise und die einzelnen Ergebnisse beschrieben werden, soll hier nur eine Zusammenfassung der Resultate gegeben werden, die den Entwurf des spezialisierten *acycCrossover*-Operators motiviert, der in Abschnitt 6.3 vorgestellt wird.

Für den Benchmark, bei dem der Crossover auf Bäumen arbeitet, haben sowohl *randomCrossover* mit zufälliger Kantenzuordnung als auch *generalCrossover* in allen Fällen korrekt GP-Graphen erzeugt. Dabei gab es keine signifikanten Unterschiede der Laufzeiten.

Bei dem Benchmark mit azyklischen Graphen ist der neue Algorithmus jedoch deutlich besser. Während mit *randomCrossover*, je nach Graphgröße, in bis zu 65 Prozent aller Fälle auch nach fünfmaliger Ausführung kein gültiger GP-Graph erzeugt wurde, lag der entsprechende Wert für *generalCrossover* bei 4 Prozent.

Bei der zyklischen Problemstellung war der Unterschied zwischen den Ergebnissen wesentlich kleiner, da hier die Erfolgshäufigkeiten für ein gelungenes Crossover nach maximal fünf Versuchen bei beiden Algorithmen bei über 99 Prozent liegen. Diese Zahlen sprechen dafür, dass beide Operatoren in einem GP-Lauf eingesetzt werden können.

Die Ergebnisse des azyklischen Benchmarks für einzelne GP-Läufe zeigen, dass die Erfolgshäufigkeit des Operators *generalCrossover* für eine korrekte Rekombination im ersten Versuch von Lauf zu Lauf stark schwanken kann. So wurden Ergebnisse zwischen 99 und 41 Prozent ermittelt. Die Häufigkeit bei *randomCrossover* variierte in einzelnen Läufen sogar zwischen 95 und 3 Prozent. Dies deutet darauf hin, dass die Operatoren auf bestimmten Elter-Graphen besser arbeiten als auf anderen und dass in einzelnen GP-Läufen, je nachdem wie sich die Population auf ein Optimum zu bewegt, verschiedene Graphtypen bevorzugt in der Population vorhanden sind.

Wie in Abschnitt 6.4 gezeigt wird, liegen die Schwächen des neuen Operators darin begründet, dass er im azyklischen Fall zu wenig auf die Vermeidung von Zyklen achtet. Folgende Verbesserungen des Operators wären denkbar:

1. Wenn beim Auswählen einer Kante keine mehr vorhanden ist, die den Anforderungen aus Tabelle 6.2 entspricht, bricht der Algorithmus ab. In so einem Fall konnte der *Crossover*-Operator keinen gültigen GP-Graphen erstellen. Die Erfolgshäufigkeit des Operators ließe sich erhöhen, wenn die letzte Kantenzuordnung vor dem Scheitern zurückgenommen würde und statt dessen eine andere Kante ausgewählt wird, die ebenfalls den Bedingungen aus Tabelle 6.2 entspräche. Wenn dieser Vorgang zu einem kompletten Backtracking ausgedehnt würde, wäre die Wahrscheinlichkeit deutlich höher, einen GP-Graphen zu finden.

   Allerdings würde diese Verbesserung mit einer exponentiellen Verlängerung der Laufzeit einhergehen. Wenn den n Eingangskanten und den m Ausgangskanten eines Teilgraphen entsprechende freie Kanten des Hauptgraphen zugeordnet werden müssen, kann es im worst-case zu $n!m!$ Zuordnungen kommen. In Voruntersuchungen wurde versucht, das Backtracking nach einer vorgegebenen Anzahl von Rücknahmen zugeordneter Kanten abzubrechen. Der Nutzen konnte allerdings nie die Laufzeiteinbußen rechtfertigen.

2. In den beiden bisher beschriebenen *Crossover*-Operatoren wird sehr viel Wert auf die Erfüllung der GP-Graph Definition gelegt, hingegen wird die Vermeidung von Zyklen bei azyklischen Graphen nur als Zusatzbedingung gefordert. Eine weitere Verbesserung für azyklische Systeme besteht darin, die Kantenzuordnungen so vorzunehmen, dass die Wahrscheinlichkeit von Zyklen herabgesetzt wird. In Abschnitt 6.3 wird ein *Crossover*-Operator vorgestellt, der diesen Ansatz verfolgt und damit eine sehr hohe Erfolgshäufigkeit auf allen getesteten azyklischen GP-Graphen erreicht.

6.3 Ein *Crossover*-Operator für azyklische Graphen – *acycCrossover*

Der im letzten Abschnitt vorgestellte *generalCrossover*-Operator funktioniert zwar auf den vorgegebenen azyklischen Testproblemen ausreichend gut, konnte aber auf Problemstellungen mit zyklischen Graphen keine wesentlichen Verbesserungen gegenüber *randomCrossover* erreichen. Die Ergebnisse der Analyse in Abschnitt 6.4 zeigen außerdem, dass die Resultate bei azyklischer Problemstellung von GP-Lauf zu GP-Lauf stark variieren. Der Prozentsatz der erfolgreichen Anwendungen schwankt zwischen 41 und 100 Prozent. Hieraus lassen sich zwei Schlüsse ziehen:

1. Die starke Abweichung in den Ergebnissen verdeutlicht, dass der neue Operator auf verschiedenen azyklischen Problemen durchaus unterschiedlich gute Resultate erzielen kann. Aus diesem Grund kann er nicht bedenkenlos in das GP-System integriert werden.

2. Die Arbeitsweise eines *Crossover*-Operators betrifft die Syntax der betroffenen Graphen und die Typisierung der einzelnen Knoten. Die Zielfunktion des GP-Problems sollte keinerlei Einfluss auf ein erfolgreiches Ausführen des Operators haben. Die Ergebnisse zeigen allerdings, dass genau dieses eintritt. Durch den Verlauf der Evolution bei den Testproblemen werden nicht immer neue zufällige Graphen erzeugt, sondern immer wieder Variationen der Graphen, die zu den Individuen mit den besten Fitnesswerten gehören. Diese sind sich somit ähnlicher als Graphen, die jedes Mal zufällig erzeugt würden.

Um die Wirkung der Operatoren für allgemeinere Graphen untersuchen zu können, wurden zwei neue Testprobleme eingeführt. In diesen hat der Graph keine semantische Bedeutung im Bezug auf eine Fitnessfunktion. Dies wurde dadurch erreicht, dass die Fitness eines Individuums nicht durch Auswertung des Graphen berechnet wird, sondern sich aus der Anzahl der noch verbliebenen Fitnessauswertungen ergibt. Auf diese Weise ist sichergestellt, dass die Diversität in der Population ausreichend hoch ist und der *Crossover*-Operator während eines GP-Laufs tatsächlich mit unterschiedliche Graphen getestet wird.

Der Unterschied zwischen den beiden neuen Testproblemen besteht lediglich darin, dass im ersten Fall ausschließlich (1,1)-, (2,1)- und (1,2)-Knoten als innere Knoten verwendet wurden, während im zweiten Fall ebenfalls (0,1)-Knoten benutzt werden durften.

Bei beiden Testproblemen erzielten die bisherigen *Crossover*-Operatoren ähnlich gute Resultate. Da sich das erste Testproblem als das schwierigere erwiesen hat, werden nur dessen Ergebnisse vorgestellt. Die Resultate vom zweiten Problem entsprechen in ihrer Tendenz den hier vorgestellten und werden ausführlich in Abschnitt 6.4 besprochen. Bei kleinen Graphen mit bis zu 20 Knoten erstellten die Operatoren im ersten Versuch noch in etwa 80 Prozent aller Fälle allen Anforderungen genügende Graphen. Diese Häufigkeit sank allerdings bei 50 Knoten auf etwa 45 Prozent, bei GP-Graphen mit 150 Knoten sogar auf 20 Prozent.

Das unbefriedigende Verhalten beider Operatoren führte zur Entwicklung eines neuen *Crossover*-Operators. Da beide bisherigen Operatoren sowohl bei Problemstellungen für zyklische GP-Graphen als auch auf Bäumen akzeptable Ergebnisse erzielen, wird der neue Operator ausschließlich für azyklische Problemstellungen eingesetzt und kann somit bestimmte Eigenschaften dieser Graphen ausnutzen.

6.3.1 Vorüberlegungen

Der neue Operator *acycCrossover* unterscheidet sich nur durch seine Kantenzuordnung von *generalCrossover* und *randomCrossover*.

Entscheidend für den Operator ist, dass in einem azyklische Graphen alle Knoten jeweils Verbindungen von einem Eingangs- oder (0,1)-Knoten und zu einem Ausgangsknoten haben. Die Bedingungen 7 und 8 der Definition 3.1 sind somit für azyklische Graphen, die die restlichen Bedingungen erfüllen, ebenfalls gegeben.

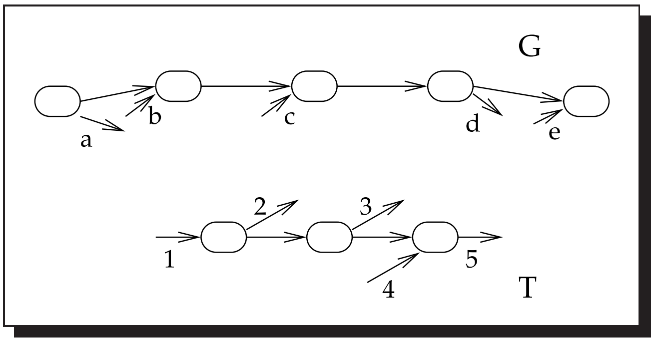
Der neue Operator *acycCrossover* braucht deshalb bei der Zuordnung der Kanten nur darauf zu achten, dass keine Zyklen erzeugt werden. Ist dies der Fall, liegt automatisch ein gesuchter GP-Graph vor.

Um die Wahrscheinlichkeit von Zyklen bei der Kantenzuordnung zu verringern, kann die Lage der Randknoten im Hauptgraphen und die Lage der Ein- und Ausgangskanten im Teilgraphen berücksichtigt werden. Dies soll an folgendem Beispiel verdeutlicht werden.

Beispiel 6.3
Der Teilgraph T aus Abbildung 6.4 soll in den Hauptgraphen G eingefügt werden. Hierzu sind die fünf Eingangs- und Ausgangskanten von T den freien Kanten von G zuzuordnen.

Eine Zuordnung, die einen azyklischen Graphen zur Folge hat, ist ($a-1, b-2, c-3, d-4, e-5$)*. Wird allerdings die vordere freie Kante* a *der Eingangskante* 4 *zugeordnet, so lässt sich ein Zyklus nicht mehr vermeiden. Nach den nun erzwungenen Zuordnungen* $d-1$ *und* $e-5$ *würde sich durch die Zuordnung der Ausgangskante* 2 *zu einer der beiden verbliebenden hinteren freien Kanten* b *oder* c *automatisch ein Zyklus ergeben.*

Für den Graphen aus Abbildung 6.4 ist die Zuordnung $a-1$ naheliegend. Einer, im Hauptgraphen sehr weit links liegenden freien Kante eines vorderen Randknotens, wird eine Eingangskante des Teilgraphen zugeordnet, die ebenfalls weit links liegt und sich somit am Anfang des Teilgraphen befindet. Eine Zuordnung der Kante 1 zur freien Kante d hätte eine



Kante	a	d	b	c	e
Verbindungen	0	2	1	1	0

Kante	1	4	2	3	5
Verbindungen	3	1	1	1	2

Abbildung 6.4: Einfügen des Teilgraphen T in den Hauptgraphen G

Verbindung vom linken Knoten des Hauptgraphen bis zum rechten Knoten des Teilgraphen zur Folge. Zur Zyklenvermeidung müsste nun die Ausgangskante 5 mit einem Knoten des Hauptgraphen verbunden werden, der rechts vom Randknoten zur freien Kante *d* liegt.

Algorithmisch lässt sich dieses Zuordnungsprinzip am einfachsten formulieren, wenn beim Hauptgraphen nicht die Lage der freien Kanten zu den Eingangsknoten betrachtet wird, also *wie weit links liegt die Kante in der Abbildung des Graphen, wenn alle Eingangsknoten links außen liegen*, sondern die Position der freien Kanten relativ zueinander. Zyklen entstehen genau dann, wenn über den neuen Teilgraph eine Verbindung von einer vorderen freien Kante des Hauptgraphen zu einer hinteren freien Kante hergestellt wird und diese beiden Kanten im Hauptgraphen bereits eine Verbindung von der hinteren zur vorderen besitzen. Wenn zum Beispiel in Abbildung 6.4 von der vorderen freien Kante *d* über den Teilgraph eine Verbindung zur Kante *b* führen würde, ergäbe sich ein Zyklus.

Werden die hinteren freien Kanten absteigend nach der Anzahl ihrer Verbindungen zu vorderen freien Kanten sortiert und die vorderen freien Kanten aufsteigend nach der Anzahl der Verbindungen, die von hinteren freien Kanten zu diesen führen, so ergibt sich jeweils eine Reihenfolge, die der Lage der Kanten im Graphen bezogen auf ihre relative Position zueinander entspricht. In Abbildung 6.4 ist dies in der oberen Tabelle zusammengefasst.

Auf die gleiche Weise werden die Eingangs- bzw. Ausgangskanten des Teilgraphen sortiert. Die Eingangskanten werden nach der Zahl ihrer Verbindungen zu Ausgangskanten absteigend sortiert und die Ausgangskanten nach der Zahl ihrer Verbindungen von Eingangskanten aufsteigend. Für das Beispiel stehen die entsprechenden Zahlen in der unteren Tabelle von Abbildung 6.4.

6.3.2 Die Kantenzuordnung

Die Zuordnung der freien Kanten des Hauptgraphen zu den Eingangs- und Ausgangskanten des Teilgraphen wurde gegenüber dem Algorithmus zu *generalCrossover* stark vereinfacht. Zuerst werden die Ausgangskanten des Teilgraphen den hinteren freien Kanten zugeordnet. Hierzu wird die hintere freie Kante mit den meisten Verbindungen zu vorderen freien Kanten der Ausgangskante zugewiesen, welche die meisten Verbindungen von

Eingangskanten hat. Die hintere freie Kante mit den zweitmeisten Verbindungen wird der Ausgangskante mit den zweitmeisten Verbindungen von Eingangskanten zugeordnet, usw.

Für die Zuordnung der vorderen Randkanten zu den Eingangskanten des Teilgraphen werden diese beiden Kantenmengen, ebenfalls sortiert. Die vorderen freien Kanten des Hauptgraphen werden aufsteigend nach der Anzahl der Verbindungen, die von hinteren freien Kanten zu ihnen führen sortiert, die Eingangskanten absteigend nach der Anzahl der Verbindungen, die sie jeweils zu Ausgangskanten des Teilgraphen besitzen.

Ausgehend von diesen Reihenfolgen wird jeder vorderen freien Kante des Hauptgraphen eine Eingangskante des Teilgraphen zugeordnet, indem für jede freie Kante die erste Eingangskante genommen wird, die noch keiner anderen freien Kante zugeordnet wurde und durch deren Zuordnung kein Zyklus entsteht.

Beispiel 6.4
In diesem Beispiel wird die Kantenzuordnung zum Graphen aus Abbildung 6.4 durchgeführt.

Zuerst werden die hinteren freien Kanten des Graphen G und die Ausgangskanten von T sortiert und einander zugeordnet. Somit ergeben sich gemäß den rechten Tabellen der Abbildung nacheinander die Paare $(b, 2)$, $(c, 3)$ *und* $(e, 2)$.

Im nächsten Schritt werden die vorderen freien Kanten von G aufsteigend sortiert. Es ergibt sich die Reihenfolge Kante a, Kante d. *Für die Eingangskanten ergibt sich* Kante 1, Kante 4.

Nun wird nacheinander zu den beiden vorderen freien Kanten eine passende Eingangskante gesucht. Zunächst wird die Zuordnung $(a-1)$ *getestet. Da sich durch eine entsprechende Verbindung zwischen dem Haupt- und dem Teilgraphen kein Zyklus ergibt, wird der freien Kante* d *die Eingangskante* 4 *zugeordnet. In diesem Fall entsteht ebenfalls kein Zyklus. Somit ist der Teilgraph in den Hauptgraphen eingefügt.*

Es ergibt sich die Zuordnung ($a-1$, $b-2$, $c-3$, $d-4$, $e-5$)*, die bereits in Beispiel 6.3 als naheliegend betrachtet wurde.*

Die Überprüfung auf Zyklenfreiheit erfolgt auf die gleiche Art wie beim *generalCrossover*-Operator. Es wird eine Matrix *connect* angelegt, in der Verbindungen zwischen den einzelnen Randknoten eingetragen werden. Die Matrix hat die Form:

$$connect(i,j) = \begin{cases} 1, \text{ falls Verbindung von Randknoten i zu Randknoten j existiert} \\ 0, \text{ sonst} \end{cases}$$

Eine neue Verbindung zwischen den Randknoten *i* und *j* führt somit zu einem Zyklus, wenn $connect(j,i) = 1$ gilt.

Soll eine neue Verbindung in die Matrix eingetragen werden, ist es nicht ausreichend, an der entsprechenden Stelle der Matrix den Wert von 0 auf 1 zu ändern. Vielmehr müssen auch alle Verbindungen eingetragen werden, die durch die Kombination der neuen Verbindung mit der alten entstehen, die bei den beiden Randknoten bereits vorher begannen oder endeten. Die Aktualisierung erfolgt mit dem Algorithmus aus Abbildung 6.5.

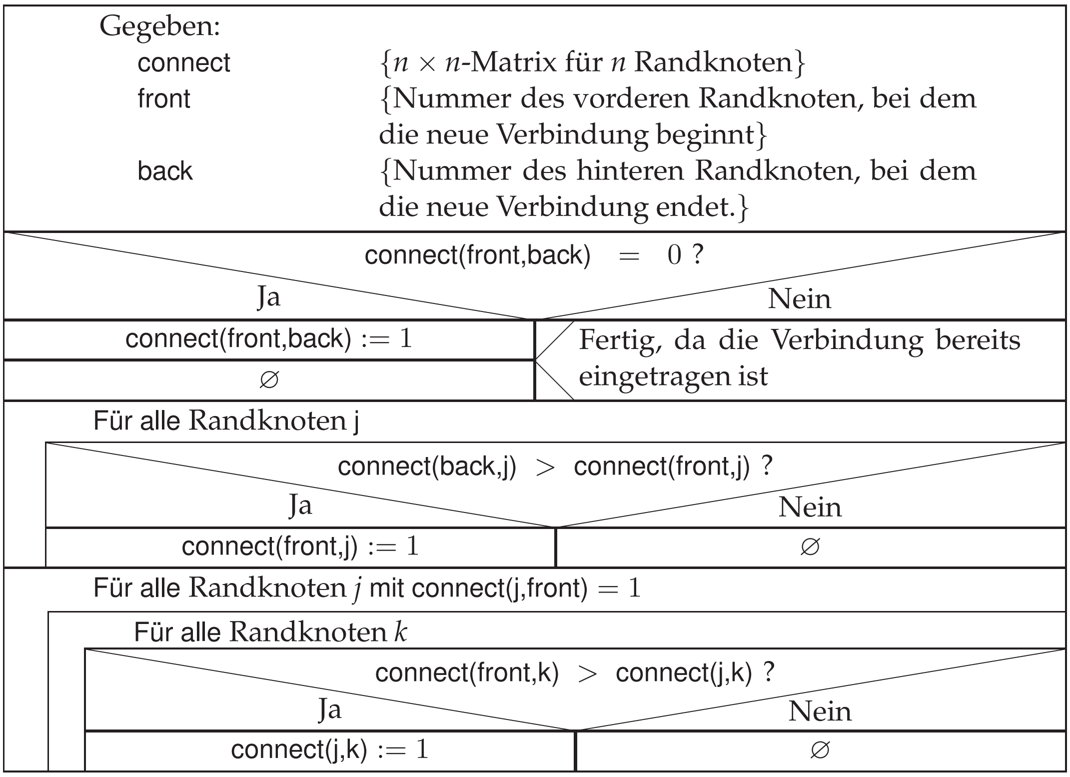


Abbildung 6.5: Algorithmus zur Aktualisierung der Verbindungsmatrix *connect*

6.3.3 Ergebnisse des *acycCrossover*-Operators

Zum Testen des Operators wurden alle azyklischen Testprobleme eingesetzt, die auch bei den anderen beiden *Crossover*-Operatoren benutzt wurden. Während die ausführliche Analyse in Abschnitt 6.4 erfolgt, werden hier die wichtigsten Ergebnisse zusammengefasst.

Der neue Operator war *randomCrossover* und *generalCrossover* bei allen Benchmarks bezüglich der Erfolgshäufigkeit deutlich überlegen. Für *acycCrossover* lag beim ursprünglichen Benchmark, bei dem noch nicht auf eine graphbezogene Fitnessfunktion verzichtet wurde, der Anteil der im ersten Versuch erfolgreichen Rekombinationsversuche – unabhängig von der Graphgröße – bei mindestens 91 Prozent. Dieser Wert betrug bei den anderen beiden Operatoren ab einer Graphgröße von 50 Knoten maximal 22 Prozent (*randomCrossover*) bzw. maximal 73 Prozent (*generalCrossover*). Die Standardabweichung dieser Zahl, bezogen auf die Unterschiede zwischen verschiedenen GP-Läufen, lag hier bei maximal 6 Prozent, während sie bei den anderen beiden Verfahren bis auf 30 bzw. 15 Prozent stieg.

Beim neuen Benchmark ohne Einfluss des Graphen auf die Fitnessfunktion, der ohne (0,1)-Knoten durchgeführt wurde, lag der Anteil der im ersten Versuch erfolgreichen Rekombinationen bei *acycCrossover* bei mindestens 60 Prozent. Bei den anderen beiden *Crossover*-

Operatoren lag der Anteil teilweise unter 25 Prozent.

Das neue Verfahren ist der einzige *Crossover*-Operator, der bei allen azyklischen Graphproblemen in mehr als 98 Prozent aller Fälle nach spätestens 5 Versuchen einen korrekten GP-Graphen erzeugen konnte. Diese Zahl liegt für *randomCrossover* bei 33 Prozent und für *generalCrossover* bei 29.

Die höhere Erfolgshäufigkeit von *acycCrossover* wird allerdings von einer Verlängerung der Laufzeit auf bis zu 146 Prozent im Vergleich zu *randomCrossover* begleitet.

6.4 Analyse der *Crossover*-Operatoren

Das Ziel eines *Crossover*-Operators ist, aus zwei vorgegebenen GP-Graphen durch Rekombination einen neuen GP-Graphen zu erstellen. Hierbei sind zusätzlich mehrere Randbedingungen einzuhalten:

- Die Mengen der Eingangs- und Ausgangsknoten müssen jeweils denen der beiden Elter-Graphen entsprechen.

  Beide GP-Graphen haben die gleiche Anzahl von Eingangs- und Ausgangsknoten. Diese Zahl muss auch beim neuen GP-Graphen erhalten bleiben. Dies wird bei allen verwendeten *Crossover*-Operatoren dadurch erreicht, dass die auszutauschenden Teilgraphen nur aus inneren Knoten der GP-Graphen bestehen.

- Die Typisierung der einzelnen Knoten durch die Semantik muss erhalten bleiben.

  Durch die Semantik eines GP-Graphen wird jedem Knoten eine Funktion zugeordnet. Durch diese Funktion ist der Eingangs- und der Ausgangsgrad des Knotens festgelegt. Beide für die Rekombination gewählten Graphen benutzen dieselbe Semantik, der der neue GP-Graph ebenfalls genügen muss. Das bedeutet, dass Eingangs- und Ausgangsgrade aller Knoten erhalten bleiben müssen.

  Alle Knoten des neuen Graphen sind in einem der beiden vorgegebenen GP-Graphen enthalten und der Eingangs- und Ausgangsgrad der Knoten wird durch die Rekombination nicht verändert. Die Operatoren erfüllen diesen Punkt somit implizit.

- Der neue Graph darf eine bestimmte Größe nicht überschreiten.

  Bei der Initialisierung eines GP-Laufs wird eine maximale Größe für die einzelnen Individuen und somit für die GP-Graphen vorgegeben. Wird aus einem GP-Graphen ein Teilgraph entfernt und durch einen größeren Teilgraphen des anderen GP-Graphen ersetzt, könnte dies dazu beitragen, dass der neu entstandene Graph zu groß wird.

  Dieser Punkt wird bei der Auswahl der auszutauschenden Teilgraphen bereits untersucht, so dass alle drei *Crossover*-Operatoren bereits zu diesem Zeitpunkt scheitern können. Diese Fehlerquelle ist bei allen Operatoren gleich groß, da für alle der gleiche Auswahlalgorithmus eingesetzt wird.

- Wenn Probleme azyklische Graphen voraussetzen, muss der neue GP-Graph ebenfalls azyklisch sein.

  Viele Testprobleme setzen azyklische Graphen voraus. Dies trifft hauptsächlich auf Probleme mit funktionalen Semantiken zu, kann aber auch für algorithmische Semantiken gefordert werden.

Während die ersten beiden Punkte durch die Algorithmen der *Crossover*-Operatoren automatisch befolgt werden, müssen die letzten beiden Punkte sowie die Bedingung, überhaupt einen GP-Graphen zu erzeugen, nicht erfüllt sein.

Um *randomCrossover* mit den beiden Weiterentwicklungen *generalCrossover* aus Abschnitt 6.2 und *acycCrossover* aus Abschnitt 6.3 vergleichen zu können, muss bewertet werden, wie gut ein Operator ist. Die Güte richtet sich primär danach, wie oft bei einer Rekombination die oben genannten Bedingungen erfüllt sind, also ein weiter verwendbares Individuum entstanden ist. Weitere Gütekriterien sind die Laufzeit des Crossover-Operators und die Anzahl verschiedener Nachkommen, die er aus einem Elternpaar erzeugen kann.

Zum Testen eines Operators werden zwei GP-Graphen erzeugt, bei denen die Knoten typisiert sind und bei denen somit über die Typisierung jeweils Eingangs- und Ausgangsgrad aller Knoten festgelegt ist.

Es wird nun überprüft, ob der *Crossover*-Operator bei einmaliger Anwendung einen neuen GP-Graphen erzeugt, bei dem die obigen Bedingungen erfüllt sind. Ist dies nicht der Fall, wird der Operator weitere vier Mal auf dieselben beiden Elter-Graphen angewandt und jedesmal überprüft, ob ein den Forderungen genügender GP-Graph erzeugt wurde.

Wird dieser Test sehr oft mit unterschiedlichen Graphen wiederholt, kann die Häufigkeit, mit der ein Operator im ersten bzw. spätestens im fünften Versuch einen geforderten GP-Graphen erzeugt, als Gütekriterium benutzt werden.

Zur Durchführung dieser Tests wurde das GP-System *GGP* benutzt. Es liegt die Überlegung zu Grunde, dass während eines GP-Laufs viele verschiedene Graphen erzeugt werden, die als Grundlage für den Crossovertest dienen können.

Bei der Bewertung wurden zunächst drei Graphklassen unterschieden:

1. zyklische GP-Graphen,
2. azyklische GP-Graphen und
3. GP-Graphen, die Baumform haben.

Diese Klassen sind nicht disjunkt. Die Praxis hat allerdings gezeigt, dass Probleme, die zyklische Graphen zulassen, während eines GP-Laufs nur in den seltensten Fällen azyklische Graphen erzeugen und andererseits Probleme mit azyklischen Graphen so gut wie nie Bäume erzeugen, wenn die entsprechenden Möglichkeiten zu allgemeineren Graphen gegeben sind.

Für jede dieser Graphklassen wurde zuerst ein Testproblem aus Kapitel 5 verwendet. Bei den zyklischen Graphen wurde das '*Artificial Ant/Santa Fé Trail*'-Problem und bei den anderen beiden Klassen jeweils das *Klassifikations*-Problem aus Abschnitt 5.2 mit 500 Testdaten benutzt.

In Tabelle 6.3 befindet sich die Auswahl der erlaubten Knotentypen für die drei Testprobleme, wobei die Zahlen in Klammern Eingangs- und Ausgangsgrad des jeweiligen Knotens wiedergeben.

Artificial Ant	*Klassifikation* – azyklisch	*Klassifikation* – Baum
IfFoodAhead (1,2)	C (0,1)	C (0,1)
Left (1,1)	Inpv (0,1)	Inpv (0,1)
Right (1,1)	P (1,1)	P (1,1)
Move (1,1)	Add (2,1)	Add (2,1)
Merge (2,1)	Sub (2,1)	Sub (2,1)
	Mul (2,1)	Mul (2,1)
	Div (2,1)	Div (2,1)
	Threshold (1,1)	Threshold (1,1)
	Sin (1,1)	Sin (1,1)
	Cos (1,1)	Cos (1,1)
	Branch (1,2)	

Tabelle 6.3: Übersicht der Funktionen, die in den ersten drei Benchmarks benutzt wurden

In Tabelle 6.4 sind die Wahrscheinlichkeiten der genetischen Operatoren angegeben, die während der Benchmarks benutzt werden. Beim *Artificial Ant*-Problem wird, neben den üblichen Operatoren, auch *Zyklus* verwendet. Da bei der Initialisierung von GP-Graphen immer azyklische Graphen erzeugt werden, ist dies für das GP-System die erste Möglichkeit zur Erzeugung von Zyklen. Auf der anderen Seite wurde die Nebenbedingung der *Crossover*-Operatoren – zyklenfreie Graphen zu erzeugen – an die Wahrscheinlichkeit des Operators *Zyklus* gekoppelt. Genau dann, wenn $P(\text{Zyklus}) \neq 0$ durch den Initialisierungswert für *Zyklus* erfüllt ist, darf der jeweilige *Crossover*-Operator ebenfalls Zyklen erzeugen.

Tabelle 6.5 gibt eine Übersicht über die restlichen für einen GP-Lauf benötigten Initialisierungen. Jeder der drei *Crossover*-Operatoren wurde mit vier verschiedenen maximalen Graphgrößen getestet, zu jeder dieser Kombinationen wurden 60 Läufe durchgeführt. Somit wurden für jedes der Testprobleme für alle Graphgrößen jeweils etwa 840000 *Crossover*-Versuche berechnet. Der Operator *acycCrossover* wurde für das *Artificial Ant*-Problem nicht eingesetzt, da er nur für azyklische Graphen verwendbar ist.

6.4.1 Das azyklische *Klassifikations*-Problem

Abbildung 6.6 zeigt die Ergebnisse, die sich beim azyklischen *Klassifikations*-Problem mit jeweils 60 Läufen bei den vier verschiedenen maximalen Graphgrößen ergaben. Jedes Tor-

Operator	*Artificial Ant*	*Klassifikation*
Knoten mutieren	12 %	14 %
Knoten einfügen	12 %	14 %
Knoten löschen	12 %	14 %
Knoten verschieben	12 %	14 %
Pfad einfügen	12 %	14 %
Pfad löschen	12 %	14 %
Zyklus	12 %	0 %
Crossover	14 %	14 %

Tabelle 6.4: Die Wahrscheinlichkeiten der genetischen Operatoren, die bei den ersten drei Benchmarks benutzt werden. Die Spalte *Klassifikation* gilt sowohl für den azyklischen Fall als auch für die Bäume

Parameter	Wert
Populationsgröße	100
Fitnessauswertungen	100000
Turniergröße	4/2
Graphgröße	20, 50, 100, 150
Crossover	*random*, *general*, (*acyc*)
Läufe	für jede Kombination 60

Tabelle 6.5: Weitere Initialisierungen zu den ersten drei Benchmarks

tendiagramm steht hierbei für die über 60 Läufe gemittelten Häufigkeiten für einen *Crossover*-Operator bei einer Graphgröße. In jedem Diagramm werden die prozentuale Häufigkeit des korrekten Grapherzeugens im ersten Versuch sowie das Scheitern in fünf aufeinanderfolgenden Versuchen dargestellt. Der zu vollen 100 Prozent fehlende Teil stellt somit dar, wie oft ein Operator im zweiten bis fünften Versuch erfolgreich war.

Die Zahlen verdeutlichen, dass der Operator *randomCrossover* unzureichende Ergebnisse produziert. Die anderen beiden Operatoren hingegen konnten in den meisten Fällen nach spätestens fünf Versuchen einen GP-Graphen erzeugen, der den Forderungen entspricht.

Die Überlegenheit von *acycCrossover* gegenüber *generalCrossover* zeigt sich allerdings, wenn die Erfolge im ersten Versuch betrachtet werden. Während dieser Wert bei *generalCrossover* – je nach Graphgröße – bis auf 73 Prozent fällt, bleibt er bei *acycCrossover* auf über 90 Prozent.

Die Unterschiede dieser Ergebnisse werden beim Betrachten der Standardabweichungen bezüglich der einzelnen Läufe noch deutlicher. Abbildung 6.7 zeigt exemplarisch die Häufigkeiten für das Erhalten korrekter Graphen im ersten Versuch in den einzelnen GP-Läufen bei Graphgrößen von maximal 20 und 50 Knoten. Jeder Punkt in den Graphen steht für die genannte Häufigkeit innerhalb eines GP-Laufs. Zur übersichtlicheren Gestaltung der Diagramme sind die einzelnen Läufe für jeden der Operatoren nach der Erfolgshäufigkeit sortiert.

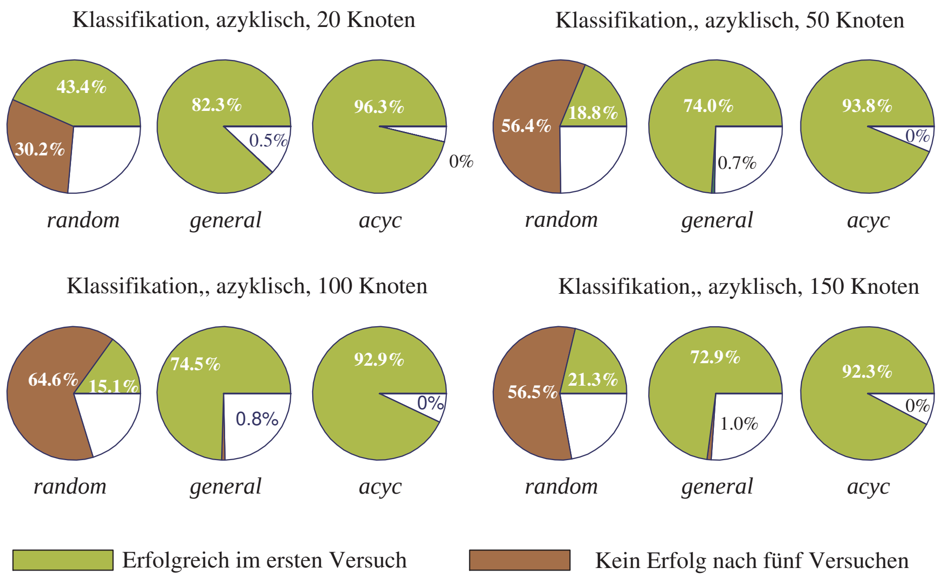


Abbildung 6.6: Erfolgshäufigkeiten der Crossover-Operatoren beim azyklischen *Klassifikations*-Problem gemittelt über jeweils 60 Läufe.

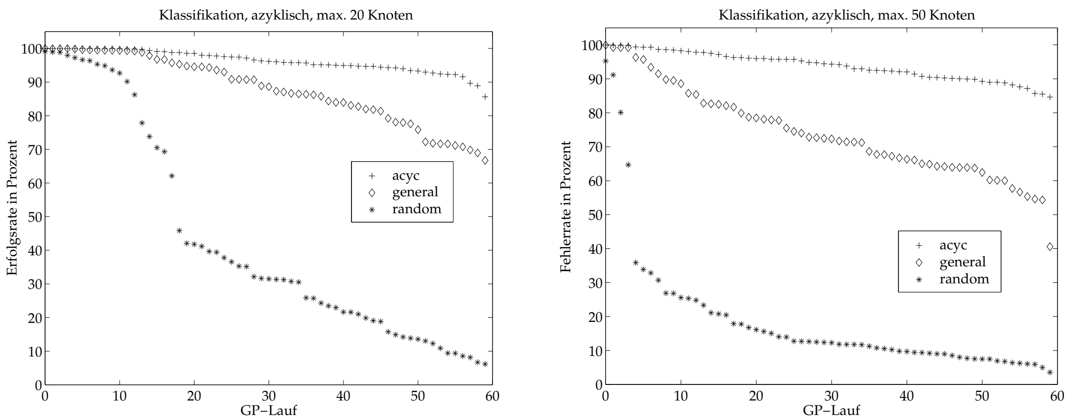


Abbildung 6.7: Ergebnisse der prozentualen Erfolgshäufigkeiten im ersten Versuch bei den einzelnen GP-Läufen beim azyklischen *Klassifikations*-Problem mit maximal 20 bzw. 50 Knoten im Graphen

Es zeigt sich, dass die Werte für *randomCrossover* und *generalCrossover* für einzelne Läufe stark variieren, während sie bei *acycCrossover* relativ nah beieinander liegen. Tabelle 6.6 fasst die Ergebnisse zusammen und enthält zusätzlich die Werte für die GP-Läufe mit größeren Graphen. Da deren Diagramme qualitativ denen aus Abbildung 6.7 entsprechen, wurde auf den Abdruck verzichtet.

Sowohl bei *randomCrossover* als auch bei *generalCrossover* gibt es Läufe, innerhalb derer in fast jedem Crossover-Versuch direkt ein geforderter GP-Graph erzeugt werden konnte. Al-

randomCrossover				
Size	20	50	100	150
Max.	99.1%	95.2%	73.4%	95.1%
Min.	6.2%	3.6%	3.2%	3.6%
Avg.	43.4%	18.4%	15.1%	21.3%
Sd.	31.9	19.1	16.8	23.7

generalCrossover				
Size	20	50	100	150
Max.	99.5%	99.9%	99.6%	99.3%
Min.	66.7%	40.6%	44.6%	41.6%
Avg.	87.9%	74.0%	74.5%	72.9%
Sd.	10.1	13.2	14.8	15.2

acycCrossover				
Size	20	50	100	150
Max.	100%	100%	100 %	100 %
Min.	85.6%	84.6%	80.6%	74.3%
Avg.	96.3%	93.8%	92.9%	92.3%
Sd.	3.2	4.1	4.9	6.0

Legende

Size:	maximale Graphgröße
Min:	kleinste Erfolgshäufigkeit
Max:	größte Erfolgshäufigkeit
Avg:	durchschnittliche Erfolgshäufigkeit
Sd:	Standardabweichung

Tabelle 6.6: Die Erfolgshäufigkeit der Operatoren im ersten Versuch – bezogen auf einzelne GP-Läufe beim azyklischen *Klassifikations*-Problem

lerdings gibt es bei *randomCrossover* auch Läufe, in denen so gut wie kein Crossover auf Anhieb gelingt. In einigen Läufen von *generalCrossover* wurde ebenfalls in etwa 40 Prozent aller Fälle kein geforderter GP-Graph im ersten Versuch erzeugt. Entsprechend hoch sind in beiden Fällen die Standardabweichungen.

Bei *acycCrossover* liegen die Häufigkeiten immer zwischen 78 und 100 Prozent, die Standardabweichungen sind demnach relativ klein.

Diese Ergebnisse lassen die Schlussfolgerung zu, dass sowohl der Erfolg von *randomCrossover* als auch der von *generalCrossover* stark von der Struktur der Graphen abhängt. Durch den Verlauf der Evolution werden offensichtlich jeweils Graphen erzeugt, die einer bestimmten Struktur gehorchen. Diese können dann durch die beiden Operatoren unterschiedlich gut verarbeitet werden.

Der Operator *acycCrossover* hat offensichtlich weniger Probleme, geeignete GP-Graphen zu erzeugen. Da die Ergebnisse bei allen Läufen ähnlich ausfielen, scheint der Operator auf relativ vielen verschiedenen azyklischen Graphstrukturen gute Ergebnisse zu erzielen.

6.4.2 Weitere azyklische Graphprobleme

Die Ergebnisse des vorigen Abschnitts legen einerseits die Vermutung nahe, dass *randomCrossover* und *generalCrossover* auf verschiedenen azyklischen Graphen unterschiedlich gut funktionieren. Andererseits wird klar, dass durch die Verwendung von den bekannten Testproblemen innerhalb eines GP-Laufs oft Graphen erzeugt werden, die eine ähnliche Struktur haben. Die Vermutung, dass innerhalb eines GP-Laufs möglichst viele unterschiedliche Graphen erzeugt werden, trifft also nicht zu.

Aus dieser Überlegung heraus wurden zwei weitere Testprobleme für azyklische Graphen erzeugt. Dabei wurde ausschließlich Wert darauf gelegt, dass während eines GP-Laufs möglichst viele verschiedene Graphen erzeugt wurden. Hierzu wurde die Fitness eines Individuums von der Struktur des Graphen entkoppelt, so dass nicht die Graphen von Individuen mit einer guten Fitness die gesamte Population beherrschen können.

Die Fitnessfunktion bei beiden Testproblemen lautet:

$$fitness = \begin{cases} \text{Evals}_{allowed} - \text{Evals}_{used} & \text{, falls } size \geq 0.75 \times maxsize \\ 2(\text{Evals}_{allowed} - \text{Evals}_{used}) & \text{, sonst} \end{cases} \tag{6.1}$$

mit

$\text{Evals}_{allowed}$:	Anzahl der insgesamt erlaubten Fitnessberechnungen,
Evals_{used}:	Nummer der aktuellen Fitnessberechnung,
size:	Anzahl der Knoten im GP-Graphen und
maxsize:	Maximal erlaubte Knotenzahl im Graphen.

Je kleiner der Fitnesswert ist, desto besser ist das Individuum. Die Funktion wurde aus zwei Gründen gewählt:

1. Ist die Größe eines GP-Graphen kleiner als 75 Prozent der erlaubten Größe, wird das Individuum bestraft. Auf diese Weise soll ausgeschlossen werden, dass eine Drift zu kleineren Individuen entstehen kann. Dies würde bedeuten, dass die Graphen wesentlich kleiner als die festgelegte maximale Größe wären.

2. Die Fitness eines Individuums ist umso besser, je neuer das Individuum ist. So wird verhindert, dass dasselbe Individuum häufig zur Rekombination herangezogen wird und somit mehrere ähnliche Graphen erzeugen könnte.

Für beide Testprobleme wurde eine funktionale Semantik benutzt, bei der die Grundfunktionen lediglich die Aufgabe haben, den Eingangs- und Ausgangsgrad der assoziierten Knoten festzulegen. Ansonsten liefern alle Funktionen – unabhängig von den Eingangsparametern – die nach Formel 6.1 berechnete Fitness des Individuums.

Tabelle 6.7 gibt eine Übersicht über die Knotentypen, die durch die Wahl der Grundfunktionen in den beiden neuen Testproblemen benutzt werden. Die Benchmarks heißen *Nofit 1* und *Nofit 2*.

Der einzige Unterschied zwischen den beiden Benchmarks ist die Verwendung des Knotentyps *ZeroOne* bei *Nofit 2*. Während bei *Nofit 1* alle Knoten mit mehreren Ein- oder Ausgängen potentielle Gefahren für eine Zyklenentstehung darstellen, kann ein *ZeroOne*-Knoten nie Teil eines Zyklus sein. Daher ist das Risiko, bei *Nofit 2* einen Zyklus zu erzeugen, geringer als bei *Nofit 1*. Es sind somit größere Erfolgshäufigkeiten zu erwarten.

In Abbildung 6.8 sind – analog zu Abbildung 6.6 – die Erfolgshäufigkeiten der drei Crossover-Operatoren beim Testproblem *Nofit 1* dargestellt. Für die Läufe wurden dieselben Initialisierungen aus den Tabellen 6.4 und 6.5 benutzt wie für das azyklische *Klassifikations*-Problem.

Funktionsname	Knotentyp	*Nofit 1*	*Nofit 2*
TwoOne	(2,1)	x	x
OneTwo	(1,2)	x	x
OneOne	(1,1)	x	x
ZeroOne	(0,1)		x

Tabelle 6.7: Die Funktionen für die Testprobleme *Nofit 1* und *Nofit 2*. wurden nach Eingangs- und Ausgangsgrad benannt. Die Kreuze in den Spalten geben an, welche Knotentypen für die Testprobleme benutzt wurden.

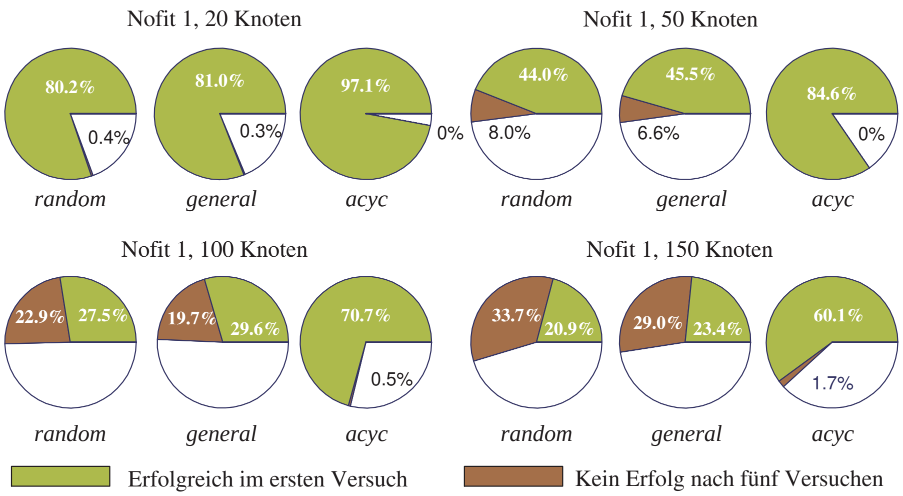


Abbildung 6.8: Erfolgshäufigkeiten der Crossover-Operatoren beim Testproblem *Nofit 1*, gemittelt über jeweils 60 Läufe.

Beim Betrachten der Ergebnisse fällt zunächst auf, dass *randomCrossover* bei diesem Benchmark etwas besser abschneidet als beim azyklischen *Klassifikations*-Problem. Die Zahlen in Tabelle 6.8 zeigen auch, dass die Unterschiede zwischen den einzelnen GP-Läufen deutlich kleiner geworden sind. Dies ist ein weiterer Indikator dafür, dass innerhalb eines GP-Laufs beim azyklischen *Klassifikations*-Problem jeweils Graphen erzeugt wurden, die sich mitunter sehr ähnlich waren und der *randomCrossover*-Operator mit einigen dieser strukturellen Ähnlichkeiten besser zurecht kam als mit anderen.

Die geringere Standardabweichung von 1.2 bis 3.3 für den *randomCrossover*-Operator bei *Nofit 1* deutet darauf hin, dass die Ergebnisse dieses Benchmarks die verlässlicheren Zahlen für die Erfolgshäufigkeiten sind, da die getesteten Graphen innerhalb eines Laufs einerseits deutlich unterschiedlicher als beim azyklischen *Klassifikations*-Problem sind, sich aber andererseits trotzdem in allen Läufen ähnliche Erfolgshäufigkeiten einstellen.

Soll *randomCrossover* allerdings später für ein Optimierungsproblem eingesetzt werden,

randomCrossover				
Size	20	50	100	150
Max.	88.1%	51.2%	29.7%	25.7%
Min.	74.3%	40.8%	24.7%	18.2%
Avg.	80.2%	44.0%	27.5%	20.9%
Sd.	3.3	2.2	1.4	1.2

generalCrossover				
Size	20	50	100	150
Max.	88.2%	50.2%	33.5%	25.9%
Min.	72.8%	43.8%	27.9%	21.8%
Avg.	81.0%	45.5%	29.6%	23.4%
Sd.	3.5	1.4	1.2	0.8

acycCrossover				
Size	20	50	100	150
Max.	98.7%	87.9%	77.0%	69.4%
Min.	95.0%	79.5%	64.8%	52.3%
Avg.	97.1%	84.6%	70.7%	60.1%
Sd.	0.8	1.8	2.6	3.2

Legende

Size:	maximale Graphgröße
Min:	kleinste Erfolgshäufigkeit
Max:	größte Erfolgshäufigkeit
Avg:	durchschnittliche Erfolgshäufigkeit
Sd:	Standardabweichung

Tabelle 6.8: Die Erfolgshäufigkeit der Operatoren im ersten Versuch – bezogen auf einzelne GP-Läufe bei *Nofit 1*

kann es passieren, dass durch den Weg der Evolution Graphen erzeugt werden, mit denen *randomCrossover* nur schlecht umgehen kann.

Abbildung 6.8 zeigt weiterhin, dass sich die Erfolgshäufigkeiten von *randomCrossover* und *generalCrossover* sehr ähnlich sind. Die Erfolgsraten im ersten Versuch sind bei *generalCrossover* nur um maximal 2.5 Prozent besser, der Unterschied beim Scheitern nach fünf Versuchen beträgt maximal 5 Prozent.

Diese Annäherung ist verständlich, da in Abschnitt 6.3 bereits dargestellt wurde, dass für alle azyklischen Graphen die Bedingungen zu den Verbindungen mit Eingangs- und Ausgangsknoten (Punkte sieben und acht der GP-Graph-Definition 3.1) automatisch erfüllt sind. Für azyklische Probleme liegt der Unterschied zwischen den beiden Operatoren also hauptsächlich darin, dass *generalCrossover* die Möglichkeit hat, vor dem Erzeugen eines Zyklus durch eine neue Kantenzuordnung auf eine andere Eingangs- oder Ausgangskante auszuweichen – sofern noch welche zur Verfügung stehen. Dieser Unterschied hat somit einen sichtbaren, allerdings marginalen Einfluss.

Das eindeutig beste Verfahren für dieses Benchmark-Problem ist *acycCrossover*. Die Erfolgshäufigkeit bei großen GP-Graphen fällt nur auf 60 Prozent und ist somit doppelt so hoch wie die Häufigkeit von *generalCrossover*. Insgesamt führen nie mehr als 2 Prozent aller Anwendungen auch nach fünf Versuchen zu keinem Erfolg. Auch diese Zahl demonstriert gegenüber maximal 29 Prozent bei *generalCrossover* eine deutliche Verbesserung.

Die Ergebnisse belegen, dass der Hauptgesichtspunkt bei GP-Graphen mit azyklischer Struktur, deren Komplexität über der von Bäumen liegt, die Zyklenvermeidung die wichtigste Aufgabe eines Crossover-Operators ist. Da *acycCrossover* dieses ausnutzt, sind die Ergebnisse deutlich besser als die der übrigen Operatoren.

Diese Ergebnisse werden von den Zahlen zu dem Benchmark *Nofit 2* belegt, die in Abbildung 6.9 und Tabelle 6.9 aufgeführt sind.

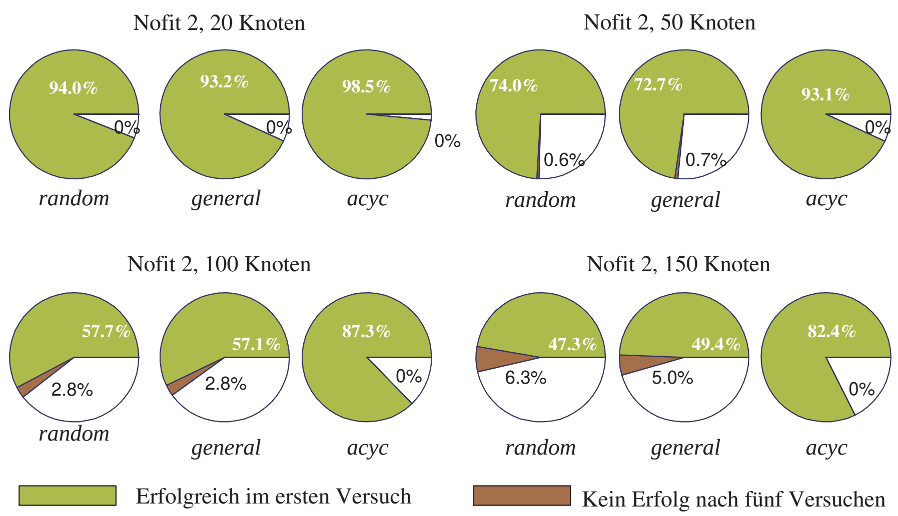


Abbildung 6.9: Erfolgshäufigkeiten der Crossover-Operatoren beim Testproblem *Nofit 2*, gemittelt über jeweils 60 Läufe.

randomCrossover				
Size	20	50	100	150
Max.	96.7%	79.7%	64.5%	53.0%
Min.	90.4%	68.0%	52.4%	40.7%
Avg.	94.0%	74.0%	57.7%	47.3%
Sd.	1.5	2.2	2.7	3.1

generalCrossover				
Size	20	50	100	150
Max.	97.1%	76.2%	63.0%	55.3%
Min.	88.3%	67.6%	52.3%	41.4%
Avg.	93.2%	72.7%	57.1%	49.4%
Sd.	1.8	2.3	2.4	2.8

acycCrossover				
Size	20	50	100	150
Max.	99.3%	94.9%	89.7%	85.5%
Min.	97.5%	91.6%	84.7%	77.8%
Avg.	98.5%	93.1%	87.3%	82.4%
Sd.	0.5	0.6	1.2	1.5

Legende

Size:	maximale Graphgröße
Min:	kleinste Erfolgshäufigkeit
Max:	größte Erfolgshäufigkeit
Avg:	durchschnittliche Erfolgshäufigkeit
Sd:	Standardabweichung

Tabelle 6.9: Die Erfolgshäufigkeit der Operatoren im ersten Versuch – bezogen auf einzelne GP-Läufe bei *Nofit 2*

Eine bisher noch nicht erklärte Beobachtung, ist die, dass die Erfolgshäufigkeiten bei zunehmender Graphgröße abnehmen. Dies gilt bei *Nofit 1* und *Nofit 2* für alle Operatoren und beim azyklischen *Klassifikations*-Problem für *acycCrossover*[1].

[1]Das unterschiedliche Verhalten der übrigen Operatoren bei diesem Benchmark ist wahrscheinlich auf die

Eine Erklärung für dieses Verhalten, die zudem noch belegt, bei welcher Art von Teilgraphen *generalCrossover* schlechter ist als *acycCrossover*, liefern die Abbildungen 6.10 bis 6.13.

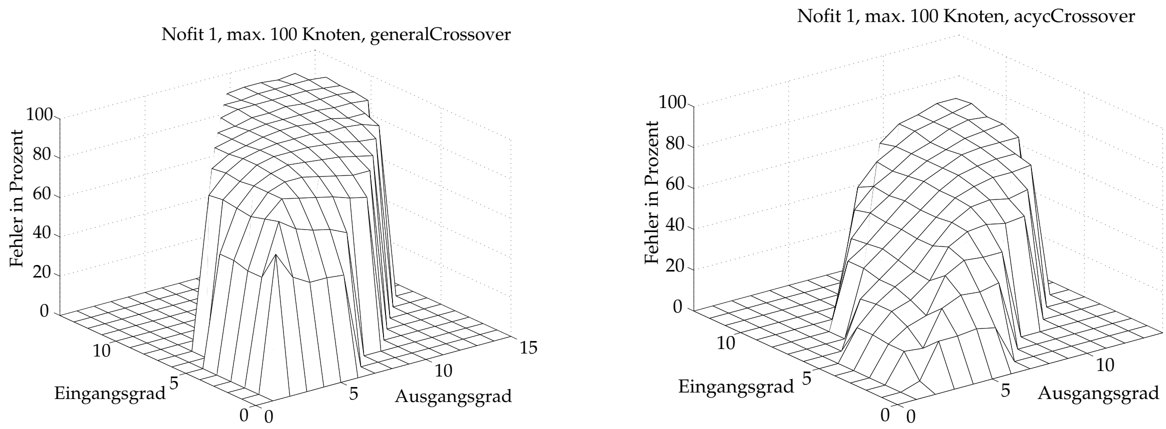


Abbildung 6.10: Prozentuale Fehlerrate bei verschiedenen Eingangs- und Ausgangsgraden der einzufügenden Teilgraphen beim Crossover für das *Nofit 1*-Problem

Während der Ausführung der zur Crossover-Untersuchung verwendeten Benchmarks wurde bei jedem Crossoverversuch mitprotokolliert, wie viele Eingangskanten und Ausgangskanten der einzufügende Teilgraph besitzt. Die Abbildungen 6.10 und 6.12 zeigen exemplarisch für zwei der Benchmarks die Fehlerhäufigkeiten bei *acycCrossover* und *generalCrossover*. Wenn im rechten Graph von Abbildung 6.10 an der Stelle (5,4) der Wert 30 steht, bedeutet dies, dass von allen Crossoverversuchen beim *Nofit-1*-Benchmark mit einer maximalen Graphgröße von 100 Knoten, bei denen der mittels *acycCrossover* einzufügende Teilgraph fünf Eingangskanten und vier Ausgangskanten hatte, 30 Prozent aller Versuche zu keinem verwendbaren GP-Graphen führten.

Zur Verbesserung der Übersichtlichkeit in dieser Abbildung wurden nur Werte an den Stellen eingezeichnet, an denen genügend Tests durchgeführt wurden. Alle Ein-/Ausgangsgrad-Kombinationen, die nicht in mindestens einem Promille aller Crossover-Versuche dieser Benchmarkserie verwendet wurden, sind wegen mangelnder Aussagekraft nicht in dem Diagramm verzeichnet.

Abbildung 6.10 zeigt, dass mit steigendem Eingangs- und Ausgangsgrad auch die Fehlerrate der Crossover-Operatoren ansteigt. Dies ist naheliegend, da infolge einer steigenden Anzahl von Randknoten des Hauptgraphen auch die Gefahr wächst, beim Einfügen eines Teilgraphen einen Zyklus zu erzeugen. Qualitativ entsprechen die Diagramme für andere Graphgrößen und andere Testprobleme den hier gezeigten. Daher werden sie hier nicht abgebildet.

In Abbildung 6.11 sind die absoluten Häufigkeiten dargestellt, mit denen verschiedene Eingangs-/Ausgangsgradkombinationen bei den Abbildung 6.10 zugrunde liegenden

großen Unterschiede zwischen den einzelnen Läufen zurückzuführen.

Testläufen vorkamen. Hierbei zeigt sich, dass die Diagramme für die beiden unterschiedlichen Crossover-Varianten qualitativ ähnlich sind. Dies war zu erwarten, da für beide der gleiche Algorithmus zur Bestimmung der Kantenpaare verwendet wurde.

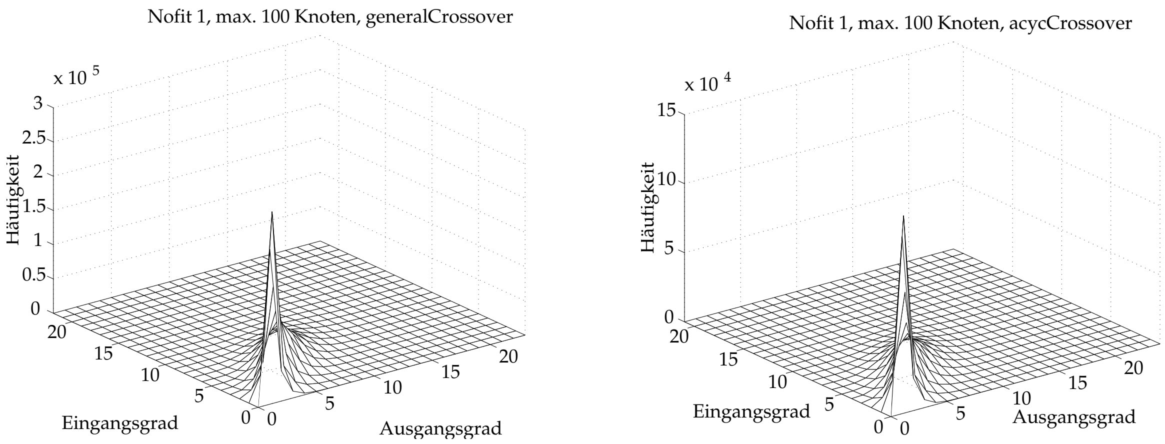


Abbildung 6.11: Absolute Häufigkeit des Auftretens verschiedener Eingangs-/Ausgangsgrad-Kombinationen beim *Nofit 1*-Problem mit 100 Knoten

Der quantitative Unterschied liegt an der unterschiedlichen Erfolgshäufigkeit der beiden Operatoren. Da bei *generalCrossover* wesentlich weniger Crossover-Versuche im ersten Versuch glücken als bei *acycCrossover* und diese bei Misserfolg insgesamt fünf Mal wiederholt wurden, ergibt sich somit eine wesentlich höhere Gesamtzahl einzelner Crossover-Versuche als bei *acycCrossover*. Die Summe aller einzelnen Crossover-Versuche lautet bei diesem Benchmark zum Beispiel 1206187 für *acycCrossover* und 2392815 für *generalCrossover*.

Bei allen Benchmarks hat sich gezeigt, dass die Anzahl der Eingangs- und Ausgangskanten der ausgewählten Teilgraphen meistens relativ dicht beieinander liegen. Dies liegt an der Vorgehensweise des Teilalgorithmus aus Abschnitt 4.2.9.1. Wenn die Menge der möglichen Teilgraphen aufgebaut wird, die zum Crossover herangezogen werden können, wird von Teilgraph zu Teilgraph eine Kante, die aus dem Teilgraph herausführt, verfolgt und der daran anschließende Knoten zum Teilgraphen hinzugenommen. Eingangsgrad und Ausgangsgrad aller Knoten, die in den hier beschriebenen Benchmarks vorkommen, sind kleiner oder gleich zwei. Durch den neuen Knoten kann sich sowohl der Eingangsgrad als auch der Ausgangsgrad des Teilgraphen um maximal eins ändern (je nach Knotentyp und ob weitere Kanten von dem bisherigen Teilgraphen zu diesem Knoten führen). Die kleine Differenz zwischen Eingangsgrad und Ausgangsgrad deutet darauf hin, dass sich die Anzahl der Eingangskanten und die der Ausgangskanten somit ähnlich entwickeln. Dies könnte passieren, wenn im Durchschnitt immer abwechselnd eine Eingangskante und eine Ausgangskante zur Erweiterung des Graphen benutzt werden.

Der Effekt der ähnlichen Eingangs- und Ausgangsgrade wird weiterhin dadurch verstärkt, dass von zwei Teilgraphmengen von beiden Eltern jeweils ein Teilgraph ausgewählt wird.

Diese beiden ausgewählten Graphen müssen in der Anzahl der Eingangskanten und Ausgangskanten identisch sein. Wenn nun in beiden Teilgraphmengen die Wahrscheinlichkeit für ähnliche Eingangs- und Ausgangsgrade relativ hoch ist, multiplizieren sich entsprechend die Wahrscheinlichkeiten bei der Suche nach Übereinstimmungen.

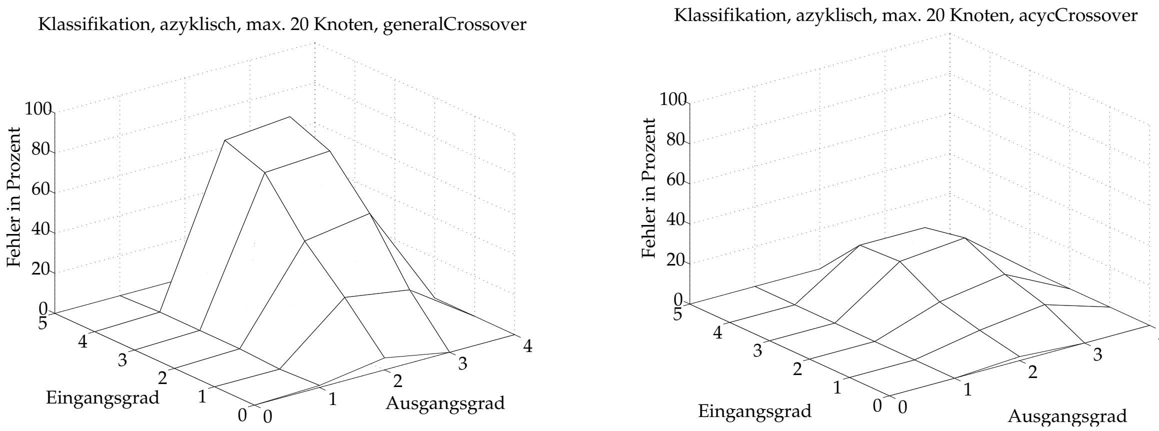


Abbildung 6.12: Prozentuale Fehlerrate bei verschiedenen Eingangs- und Ausgangsgraden der einzufügenden Teilgraphen beim Crossover für das azyklische *Klassifikations*-Problem

Die Abbildungen 6.12 und 6.13 zeigen für das azyklische *Klassifikations*-Problem mit maximal 20 Knoten qualitativ ähnliche Resultate wie die Abbildungen 6.10 und 6.11 für *Nofit 1* bei maximal 100 Knoten.

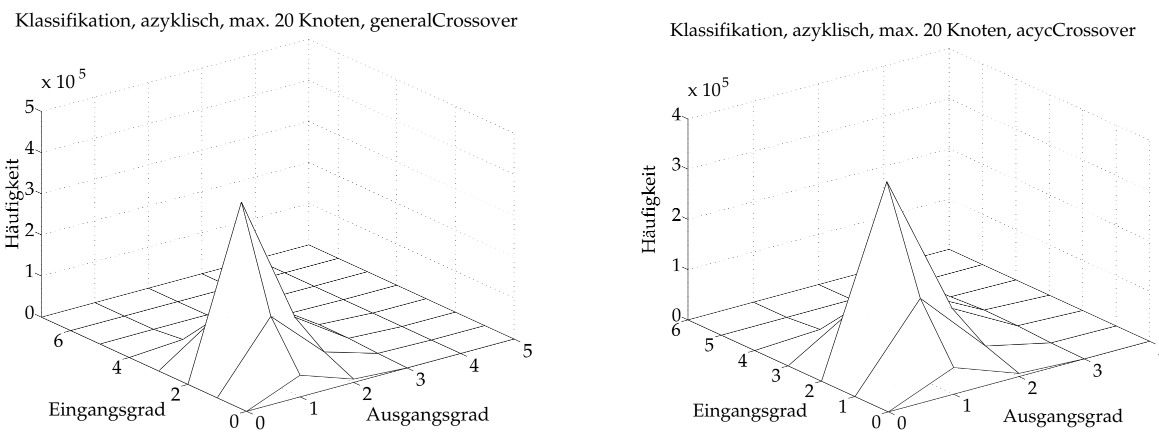


Abbildung 6.13: Absolute Häufigkeit des Auftretens verschiedener Eingangs-/Ausgangsgrad-Kombinationen beim azyklischen *Klassifikations*-Problem mit maximal 20 Knoten

In beiden Beispielen wird deutlich, dass *acycCrossover* besser als der Operator *generalCrossover* ist. Die Fehlerraten steigen bei *acycCrossover* durch eine Erhöhung des Eingangs- oder Ausgangsgrades bei weitem nicht so steil an wie bei *generalCrossover*. In Abbildung

6.12 nimmt die Fehlerrate für *generalCrossover* mit 81 Prozent ihren höchsten Wert beim Eingangs-/Ausgangsgrad (4,3) an, für *acycCrossover* mit 25 Prozent bei (3,3).

Bei dem für beide Operatoren komplizierterem Testproblem *Nofit 1* zeigt sich, dass *generalCrossover* bereits bei sehr kleinen Eingangs- und Ausgangsgraden der Teilgraphen beträchtliche Fehlerraten besitzt, während die Rate bei *acycCrossover* erst mit steigendem Grad langsam zunimmt und dann auch nicht den Grad von *generalCrossover* erreicht.

An diesem Beispiel werden zwei Vorteile von *acycCrossover* gegenüber *generalCrossover* deutlich:

1. Auf der einen Seite hat *acycCrossover* für kleine Eingangs- und Ausgangsgrade eine relativ niedrige Fehlerrate. Da gerade solche Kombinationen vom Algorithmus zur Auswahl der Teilgraphen präferiert werden (siehe Abbildung 6.11), hat der Operator insgesamt eine höhere Erfolgsrate als *generalCrossover*.

2. Auch für größere Eingangs- und Ausgangsgrade liegt die Fehlerrate von *acycCrossover* in allen untersuchten Benchmarks unter der von *generalCrossover*. Somit wird der Abstand zwischen den Erfolgsraten der beiden Operatoren – zusätzlich zu Punkt 1 – noch weiter vergrößert.

6.4.3 Das Artificial-Ant-Problem

Zur Untersuchung des Verhaltens der verschiedenen Crossover-Operatoren bei Testproblemen mit zyklischen GP-Graphen wurde das Problem *Artificial Ant on the Santa Fé Trail* verwendet, die verwendeten GP-Parameter wurden bereits in den Tabellen 6.3 bis 6.5 aufgeführt. Da der Operator *acycCrossover* ausschließlich für azyklische Graphen ausgelegt ist, wurde auf eine Untersuchung bezüglich dieses Testproblems verzichtet.

Abbildung 6.14 zeigt die Ergebnisse der beiden Operatoren, in Tabelle 6.10 sind die Raten für den Erfolg im ersten Crossover-Versuch dargestellt.

Beide Operatoren sind in der Lage, in fast allen Fällen bei allen Graphgrößen nach spätestens fünf Versuchen einen den Definitionen gerechten GP-Graphen zu erzeugen.

Die Ergebnisse für das erfolgreiche Erzeugen eines GP-Graphen im ersten Versuch liegen deutlich über den Ergebnissen, die die beiden Operatoren für die bisher vorgestellten azyklischen Benchmarks erzielen konnten. Bei kleinen Graphen mit maximal 20 Knoten ist *randomCrossover* geringfügig besser, bei größeren Graphen ist *generalCrossover* um bis zu 5 Prozent besser.

Diese Unterschiede sind allerdings zu gering, um einem der beiden Operatoren eine bessere Funktionsweise zu bescheinigen.

Die Fehlerraten beider Operatoren für verschiedene Eingangs-/Ausgangsgrad-Kombinationen bei maximal 50 Knoten (Abbildung 6.15) liegen deutlich unter denen der azyklischen Tests. Bei *randomCrossover* ist eine leichte Erhöhung der Fehlerrate bei einer Erhöhung der Eingangs- oder Ausgangskante zu erkennen, diese fällt aber

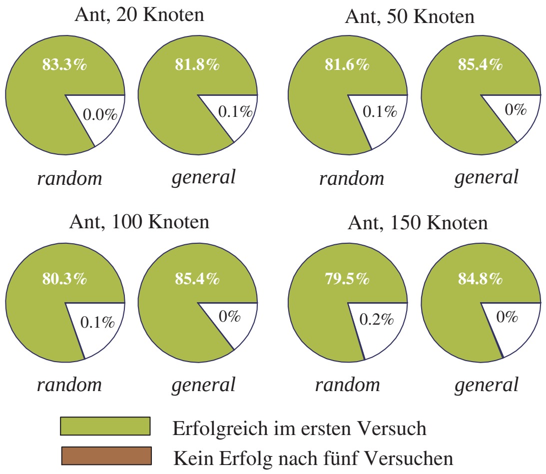


Abbildung 6.14: Erfolgshäufigkeiten der Crossover-Operatoren beim Testproblem *Artificial Ant*, gemittelt über jeweils 60 Läufe.

randomCrossover				
Size	20	50	100	150
Max.	93.4%	91.8%	89.9%	89.5%
Min.	71.5%	69.8%	71.2%	66.7%
Avg.	83.3%	81.6%	80.3%	79.5%
Sd.	4.3	4.0	4.1	4.3

generalCrossover				
Size	20	50	100	150
Max.	91.5%	91.6%	91.1%	92.1%
Min.	69.1%	68.8%	60.3%	75.6%
Avg.	81.8%	85.4%	85.4%	84.8%
Sd.	5.2	3.9	5.7	3.6

Legende

Size: maximale Graphgröße
Min: kleinste Erfolgshäufigkeit
Max: größte Erfolgshäufigkeit
Avg: durchschnittliche Erfolgshäufigkeit
Sd: Standardabweichung

Tabelle 6.10: Die Erfolgshäufigkeit der Operatoren im ersten Versuch – bezogen auf einzelne GP-Läufe bei *Artificial Ant*

wesentlich flacher aus als zum Beispiel bei *Nofit 1*. Selbst bei den größten Eingangs-/Ausgangskombinationen, die während der Testläufe in mindestens einem tausendstel alle Crossover-Versuche verwendet wurden, stieg die Fehlerrate nie über 35 Prozent.

Bei *generalCrossover* sind die Werte noch günstiger. Die Fehlerraten stiegen hier nie über 25 Prozent – es ist kein Anwachsen der Fehlerrate bei höheren Eingangs- oder Ausgangsgraden zu erkennen.

Auch bei größeren GP-Graphen erhöht sich die Fehlerrate nur unwesentlich – im Gegensatz zu dem Verhalten bei den vorgestellten azyklischen Problemen. Abbildung 6.16 enthält die Fehlerraten für Graphen mit bis zu 150 Knoten.

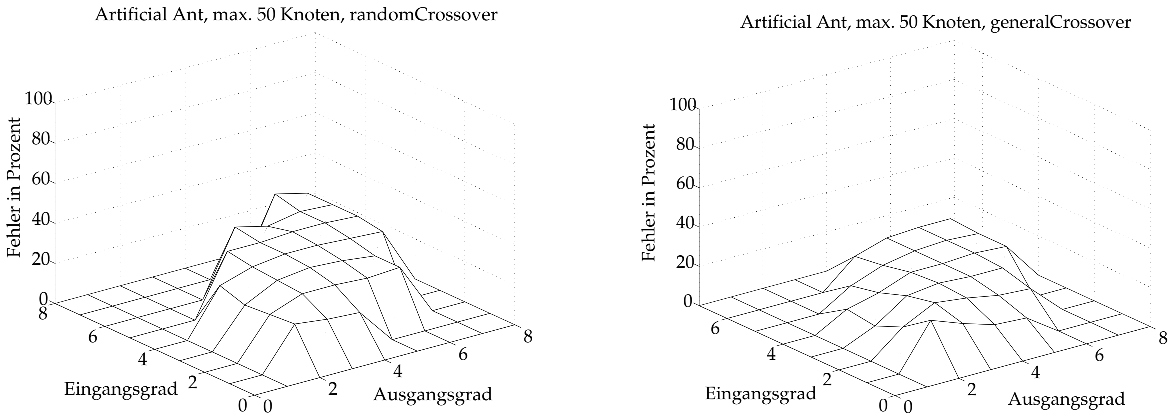


Abbildung 6.15: Prozentualer Fehler bei verschiedenen Eingangs- und Ausgangsgraden der einzufügenden Teilgraphen beim Crossover für das *Artificial Ant*-Problem mit maximal 50 Knoten

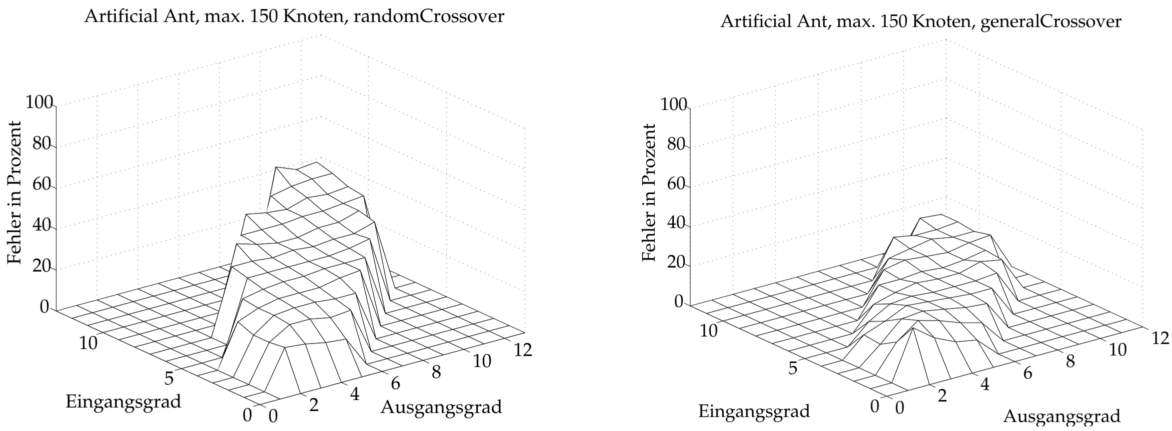


Abbildung 6.16: Prozentualer Fehler bei verschiedenen Eingangs- und Ausgangsgraden der einzufügenden Teilgraphen beim Crossover für das *Artificial Ant*-Problem mit maximal 150 Knoten

Beide Operatoren haben auf den getesteten zyklischen Elter-Graphen gute Ergebnisse erzielen können, wobei *generalCrossover* bei etwas größeren GP-Graphen geringfügig besser abschneiden konnte.

Bei *randomCrossover* sind alle möglichen Anbindungen des Teilgraphen an den Hauptgraphen gleichwahrscheinlich. Bei einer entsprechenden Zahl von Eingängen und Ausgängen des Teilgraphen führt eine mehrmalige Ausführung des Operators somit in der Regel zu unterschiedlichen Graphen. Bei *generalCrossover* hingegen sind die Wahrscheinlichkeiten für alle möglichen GP-Graphen, die nach Definition 3.1 und der Typisierung der Knoten in Frage kämen, ungleichmäßig verteilt. Einige können in den meisten Fällen sogar gar nicht erzeugt werden.

6.4.4 Baumprobleme

Wie zu erwarten war, hatte keiner der drei Crossover-Operatoren Probleme, aus zwei Bäumen einen neuen zu erstellen.

Alle Bäume, die bei einem Crossover-Versuch mit einem der drei Operatoren erstellt werden, genügen einerseits automatisch der GP-Graph-Definition, andererseits ist es durch die Baumstruktur unmöglich, durch das Verbinden von Haupt- und neuem Teilgraphen Zyklen zu erzeugen.

Die einzige Möglichkeit für das Scheitern der Crossover-Operatoren besteht darin, dass kein Teilgraphpaar gefunden werden kann, das nach dem Einsetzen des Teilgraphen des einen Elter in den anderen, die maximale Größe eine Individuums eingehalten hätte.

Tests mit dem 2-*Klassifikations*-Problem auf Baumstrukturen haben gezeigt, dass dieser Fall so gut wie nie eintritt und alle drei Algorithmen somit problemlos einsetzbar sind.

Sollen GP-Probleme mit Baum-Strukturen gelöst werden, stehen neben den hier beschriebenen Crossover-Methoden ebenfalls die traditionellen Varianten zur Verfügung, die aus KOZAs baumbasierten GP-Ansatz hervorgingen. Die Bäume, die mit KOZAs ursprüngliche Baum-Crossover erzeugt werden, sind allerdings nur eine Untermenge der Bäume, die mit den drei hier vorgestellten Verfahren erzeugt werden.

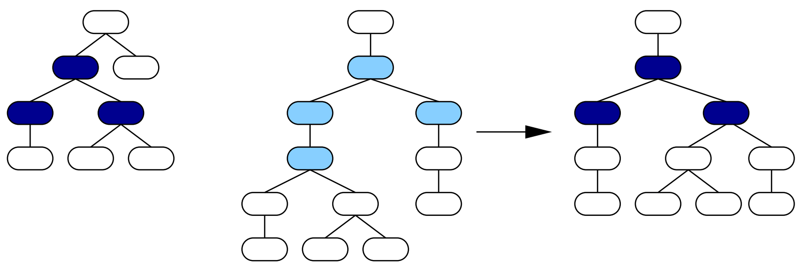


Abbildung 6.17: Ein Crossover zweier Bäume, der erst durch den graphbasierten Ansatz möglich wird.

Während bei KOZA nur komplette Teilbäume ausgetauscht werden können, sind alle drei hier vorgestellten Operatoren durch den graphbasierten Ansatz in der Lage, auch Teile eines Teilbaums zu entnehmen und in den anderen Baum einzufügen. Dies wird in Abbildung 6.17 verdeutlicht.

6.4.5 Vergleich der Laufzeiten

Abbildung 6.18 gibt einen Vergleich über die Laufzeiten der verschiedenen Operatoren. Alle Benchmarks wurden auf PCs mit AMD Athlon 900Mhz-Prozessoren durchgeführt, alle Zahlenangaben stehen für die durchschnittliche Laufzeit eines GP-Laufs in CPU-Sekunden.[1]

[1] Die Laufzeiten für die Benchmarks mit maximal 20 Knoten wurden nicht aufgeführt, da sie tendenziell mit denen für 50 Knoten übereinstimmen.

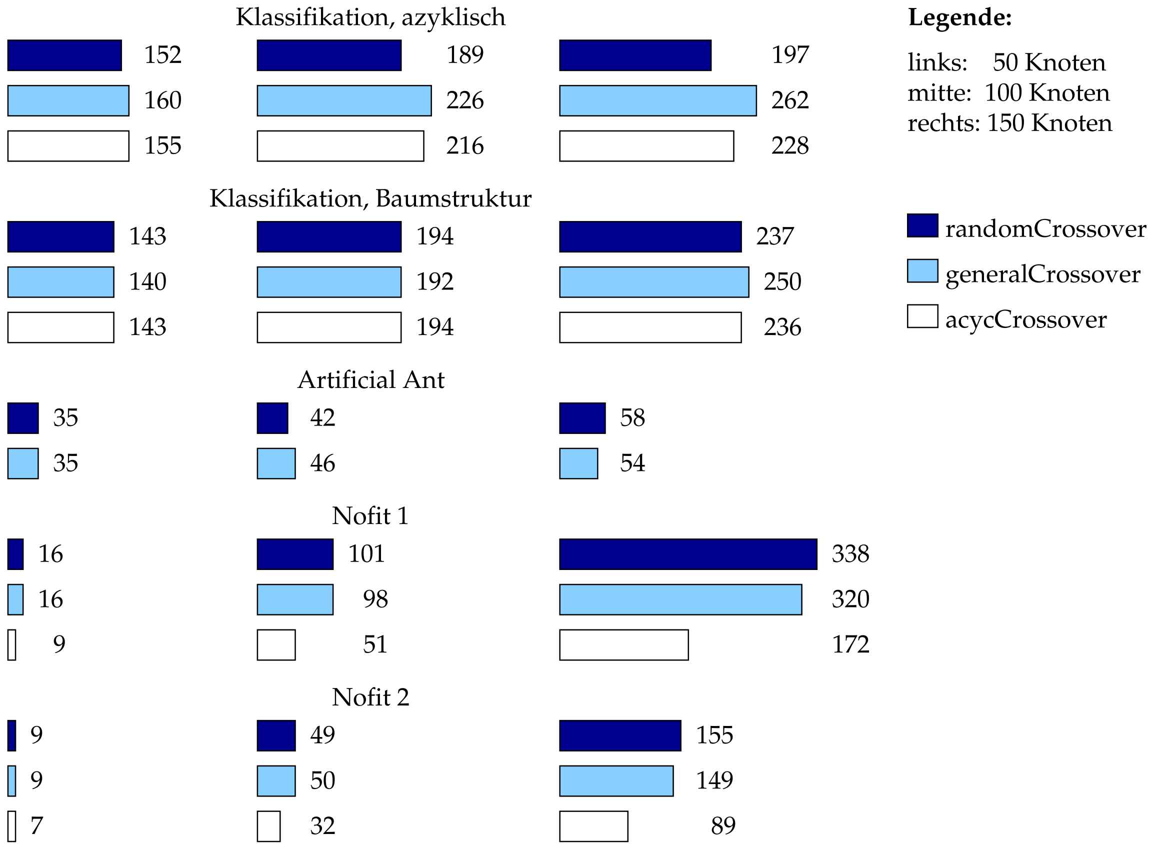


Abbildung 6.18: Durchschnittliche Laufzeiten der Testläufe in Sekunden

Bei den drei Benchmarks *Klassifikation – azyklisch*, *Klassifikation – Baumstruktur* und *Artificial Ant* sind alle drei Operatoren ähnlich gut, wobei *randomCrossover* häufig etwas schneller ist, als die beiden anderen.

Bei den beiden künstlichen Benchmarks *Nofit 1* und *Nofit 2* ist hingegen der Operator *Nofit 2* der mit Abstand schnellste. Dieses, auf den ersten Blick ungewöhnliche Verhalten soll im Folgenden näher untersucht werden.

Alle drei Crossover-Operatoren bestehen aus drei Teilen,

1. dem Ermitteln und Auswählen von Teilgraphen,
2. der Kantenzuordnung und
3. dem Zusammenfügen der zwei Graphen zu einem neuen.

Der erste und der dritte Teil sind bei allen Operatoren identisch. Bei *randomCrossover* muss nach dem dritten zusätzlich überprüft werden, ob der erstellte Graph alle Anforderungen erfüllt. Die unterschiedlichen Laufzeiten kommen somit durch die Implementierung der

Kantenzuordnung und der Anzahl der Crossover-Versuche während eines GP-Laufs zustande.

Die rechenintensivste Komponente aller Operatoren ist das Erzeugen der möglichen Teilgraphen. Bedingt durch die gewählten Datenstrukturen wird für jeden neuen Teilgraphen in quadratischer Laufzeit jede noch freie Kante des zuletzt ermittelten Teilgraphen mit jedem Knoten des Graphen verglichen, wobei die in dieser Schleife liegende Funktion nur sehr wenig Rechenzeit benötigt. Somit braucht diese Routine bei kleinen Graphen insgesamt nur sehr wenig Rechenzeit, kann aber bei großen Graphen durch die kubische Laufzeit sehr langsam werden.

Die Kantenzuordnung bei *randomCrossover* ist durch die einmalige, zufällige Auswahl der Kanten die schnellste Zuordnung. Die beiden übrigen Verfahren beinhalten Komponenten mit quadratischer Laufzeit zur Anzahl der Eingangs- und Ausgangskanten des Teilgraphen. Hierdurch werden sie bei entsprechender Anzahl von Eingangs- und Ausgangskanten deutlich langsamer als *randomCrossover*. Allerdings sind sie in diesen Fällen wiederum deutlich schneller als eine Ermittlung neuer Teilgraphen, wie sie im Falle eines fehlgeschlagenen Crossover-Versuchs vorgenommen wird.

Es ergeben sich somit folgende Punkte, mit denen die Ergebnisse aus Abbildung 6.18 erklärt werden können.

- Für kleine Graphen kann es schneller sein, viele Crossover mit *randomCrossover* durchzuführen, als mit den beiden anderen Verfahren. Diese haben zwar quadratisches Laufzeitverhalten in der Kantenzuordnung – im Gegensatz zum kubischen in der Teilgraphbestimmung, sie müssen aber in den einzelnen Schleifendurchläufen wesentlich mehr rechnen.
- Für große Graphen kann ein Operator mit deutlich höheren Erfolgsraten als *randomCrossover* besser sein als dieser, da die quadratische Laufzeit in der Kantenzuordnung in diesem Fall kürzer ist, als die sehr häufige Ausführung der Teilgraphbestimmung beim *randomCrossover*

Wie in den Abbildungen 6.8 und 6.9 zu sehen ist, liegt die Erfolgsrate von *acycCrossover* bei *Nofit 1* und *Nofit 2* deutlich höher als die der anderen beiden Verfahren. Dies bedeutet, dass die Routine zur Teilgraphbestimmung seltener aufgerufen werden muss[1]. Da die Graphen relativ groß sind, liegt die Laufzeit von *acycCrossover* entsprechend unter der der anderen Operatoren.

Das gute Abschneiden von *randomCrossover* beim azyklischen *Klassifikations*-Problem lässt sich dadurch erklären, dass in diesen Fällen die Graphen deutlich kleiner waren, als durch das Maximum der erlaubten Knoten pro GP-Graph vorgegeben war. In diesem Fall wirkt sich eine häufigere Ausführung der Teilgraphbestimmung nicht so schlecht auf die Laufzeit aus, wie die rechenintensivere Kantenzuordnung der beiden anderen Operatoren.

[1]Bei einem genauer ausgewerteten Testlauf für *Nofit 2 - 150 Knoten* ergaben sich 17576 Aufrufe für *acycCrossover*, 29488 für *randomCrossover* und 28652 für *generalCrossover*

Alle übrigen Laufzeitunterschiede sind nicht signifikant und können zudem noch ungenau sein. Es wurde zwar die Prozessorzeit zur Messung benutzt, trotzdem aber könnten noch zusätzliche Ungenauigkeiten durch das Betriebssystem in die Ergebnisse eingeflossen sein.

Eine Verbesserung der Laufzeit der Operatoren kann erreicht werden, indem der Algorithmus zur Bestimmung der Teilgraphen so umgeschrieben wird, dass intern andere Datenstrukturen verwendet werden (Hashtabellen). Auf diese Weise könnte der kubische Anteil des Programms durch einen mit quadratischer Laufzeit ersetzt werden.

Des Weiteren gelten alle in Abschnitt 4.2.9.5 angesprochenen Möglichkeiten zur Verbesserung des Laufzeitverhaltens.

6.4.6 Die Anzahl der möglichen Nachkommen

Ein großer Unterschied zwischen den drei Crossover-Operatoren ist die Anzahl der möglichen Nachkommen, die aus zwei Eltern erzeugt werden können.

Alle drei Algorithmen können nur einen kleinen Teil der für zwei Eltern mit Crossover denkbaren Nachkommen erzeugen, da nur eine kleine Teilmenge von Teilgraphen pro Elter überhaupt für einen Crossover-Versuch herangezogen werden kann (Abschnitt 4.2.9.1).

Die unterschiedlich Anzahl, welche die drei Algorithmen erzeugen können, gehen auf die verschiedenen Kantenzuordnungsverfahren der Operatoren zurück.

Die größte Anzahl verschiedener Nachkommen ist bei *randomCrossover* möglich. Wenn der neue Teilgraph über n Eingangskanten und m Ausgangskanten verfügt, kann der Operator $n!m!$ verschiedene Zuordnungen vornehmen, die auch der größten möglichen Anzahl verschiedener Nachkommen für zwei vorgegebene Teilgraphen entsprechen. Die Zahl verringert sich zwangsläufig dadurch, dass viele der Zuordnungen in den meisten Fällen nicht zu GP-Graphen führen, die Definition 3.1 erfüllen und – falls gefordert – zyklenfrei sind.

Bei *acycCrossover* handelt sich um ein vollständig deterministisches Verfahren. Für zwei Eltern wird somit immer derselbe Nachkomme erzeugt. Die einzige Ausnahme hierfür ist, wenn ein Graph in der internen Datenstruktur mehrere Repräsentationen hat und diese bei der Sortierung der Kanten zu Beginn des Algorithmus dafür sorgen, dass Kanten, die die gleiche Anzahl von Verbindungen zu anderen Kanten aufweisen, in eine unterschiedliche Reihenfolge gebracht werden (siehe Abschnitt 6.3.1).

Der Operator *generalCrossover* liegt mit der Anzahl erzeugbarer Individuen zwischen den beiden übrigen Verfahren. Die Eingangs- und Ausgangskanten werden vor Beginn der eigentlichen Zuordnung gemischt, diese erfolgt danach ebenfalls deterministisch.

Bei *acycCrossover* macht eine solche Randomisierung vor dem eigentlichen Start der Zuordnung wenig Sinn, da die Kanten zu Beginn des Algorithmus wieder sortiert würden. Es könnte sich lediglich die Reihenfolge der Kanten ändern, die bezüglich der der Sortierung zugrunde liegenden Ordnung den gleichen Wert hätten.

Bei *generalCrossover* werden die Kanten zu Beginn des Algorithmus nicht vollständig sortiert, sondern nur gemäß der Abschnitte 6.2.3 und 6.2.4 in verschiedene Klassen eingeteilt.

Innerhalb dieser Klassen kann durch die vorangegangene Randomisierung die Reihenfolge der Kanten bei verschiedenen Versuchen unterschiedlich sein und somit zu verschiedenen Nachkommen führen.

Wenn für ein Testproblem die Erfolgsraten zweier der Crossover-Operatoren gleich sind und die Laufzeiten sich ebenfalls nur um akzeptable Faktoren unterscheiden, sollte auf jeden Fall der Operator benutzt werden, der für ein Teilgraphpaar mehr verschiedene Nachkommen generieren kann, also *randomCrossover* an Stelle von *generalCrossover* und dieser wiederum eher als *acycCrossover*. Auf diese Weise ist eine bessere Durchdringung des Suchraums möglich.

6.4.7 Zusammenfassung

In diesem Kapitel wurden zwei neue Crossover-Operatoren eingeführt. Anhand mehrerer Benchmarks wurde das Verhalten aller drei Crossover-Operatoren untersucht und bewertet.

Es wurde der Operator *generalCrossover* vorgestellt, ein Crossover-Operator, der auf allen Graphstrukturen eingesetzt werden kann und zu etwas besseren Ergebnissen als *randomCrossover* führt. Er erreicht dies, indem vor jeder einzelnen Kantenzuordnung überprüft wird, ob durch sie die Erfüllung der geforderten GP-Graph-Eigenschaften unmöglich würde. Allerdings kommt der Operator mit unterschiedlichen azyklischen Graphen nicht immer gleichgut zurecht. Des Weiteren skaliert er nur ungenügend, wenn die Anzahl der Eingangskanten und Ausgangskanten des einzufügenden Teilgraphen wächst.

Als dritter Operator wurde *acycCrossover* vorgestellt. Er funktioniert nur auf azyklischen Graphstrukturen und ist den anderen Operatoren hier deutlich überlegen. Dies wird dadurch erreicht, dass ausschließlich auf Zyklenvermeidung geachtet wird und hierbei zusätzlich zuerst Kantenzuordnungen ausprobiert werden, für welche die Erzeugung von Zyklen am unwahrscheinlichsten ist.

Die Laufzeit, die für einen Crossover benötigt wird, hängt bei großen Graphen hauptsächlich von der Bestimmung der Teilgraphen ab, die bei allen Verfahren gleich realisiert wurde. In diesem Fall werden Operatoren umso schneller, je höher ihre Erfolgsrate ist und je seltener sie somit neue Teilgraphen suchen müssen.

Bei kleinen Graphen liegt die Hauptrechenzeit in der Kantenzuordnung. Hier ist *randomCrossover* schneller als die anderen Operatoren und kann durch die in linearer Zeit erfolgende Kantenzuordnung, die durch die niedrigere Erfolgsrate bedingte häufigere Wiederholung des Operators teilweise kompensieren.

Kapitel 7

Anwendungshäufigkeiten der genetischen Operatoren

Zu Beginn eines GP-Laufs werden allen genetischen Operatoren Wahrscheinlichkeiten zugeordnet, mit denen sie im Verlauf der Evolution eingesetzt werden. Bei den in Kapitel 5 durchgeführten Benchmarks wurden sie weitestgehend gleichgesetzt[1].

In diesem Kapitel werden verschiedene adaptive Anpassungsverfahren betrachtet, mit denen die Wahrscheinlichkeiten während eines GP-Laufs angepasst werden können. Zusätzlich wird ein erweitertes Auswahlverfahren für die genetischen Operatoren vorgestellt, das vor der Betrachtung der Wahrscheinlichkeiten untersucht, ob Operatoren mit dem aktuell betrachteten Individuum überhaupt neue Graphen erzeugen können.

Ein Teil der hier vorgestellten Verfahren wurde bereits in [80] beschrieben. Zur Zeit der Veröffentlichung verfügte das GP-System über keinen Crossover-Operator und es wurden nur zwei Testprobleme betrachtet. Aus diesem Grund sind alle Versuche mit dem jetzt aktuellen GP-System *GGP* wiederholt. Zusätzlich wurden weitere Fragestellungen in die Untersuchung miteinbezogen.

7.1 Auswahlverfahren der Operatoren während eines GP-Laufs

Die Operatorwahrscheinlichkeiten sind Teil der Initialisierungsparameter des GP-Systems. Jedem der n Operatoren Op_i aus $Op_1, \ldots, Op_n$ wird bei der Initialisierung des Laufs ein Wert p_i zwischen 0 und 100 zugewiesen. Diese Werte entsprechen den prozentualen Wahrscheinlichkeiten, mit denen die einzelnen Operatoren verwendet werden, da bei der Erzeugung eines neuen Individuums genau ein Operator benötigt wird. Die Auswahl der Operatoren bei der Individuenmutation erfolgt nun proportional zu den Werten p_i. Es gilt $\sum_{i=1}^{n} p_i \leq 100$. Ist die Summe kleiner als 100, so wird der verbliebene Prozentsatz durch Replikation abgedeckt. Bei den im Folgenden vorgestellten Verfahren wird vor einer Mutation

[1] Ausnahmen waren die Fälle, in denen einzelne Operatoren gar nicht benutzt oder nur für das einmalige Beseitigen überflüssiger Knoten benötigt wurden

ein neuer Parametersatz errechnet, der anstelle der ursprünglichen Werte für die Operatorwahl herangezogen wird.

7.2 Beschreibung der Adaptationsverfahren

In der Literatur sind verschiedene Arten der Adaptation bekannt. Zur besseren Einordnung der hier vorgestellten Verfahren wird zunächst in Abschnitt 7.2.1 eine Einordnung der verschiedenen Ansätze geschildert. Ab Abschnitt 7.2.2 werden die Verfahren zur Adaptation der Operatorwahrscheinlichkeiten vorgestellt und anhand von Abschnitt 7.2.1 eingeordnet.

7.2.1 Übersicht

Die hier vorgestellte Einordnung verschiedener Adaptationstechniken wurde von ANGELINE 1995 veröffentlicht. Eine entsprechend ausführlichere Übersicht befindet sich in [4].

Adaptive Verfahren im Bereich der Evolutionären Algorithmen lassen sich in drei verschiedene Gruppen einteilen. Zunächst können Verfahren Adaptationen auf der Populationsebene vornehmen. Dies könnte eine Änderung der Individuenrepräsentation sein oder populationsweite Wahrscheinlichkeitsänderungen für die Benutzung bestimmter genetischer Operatoren. Beispiele für Operatorwahrscheinlichkeitsadaptation sind die $1/5$-Erfolgsregel bei den Evolutionsstrategien [103] oder eine populationsweite Anpassung der Wahrscheinlichkeiten bei der Genetischen Programmierung [124] sowie, ebenfalls bei GP, der Prozentsatz mit dem innere Knoten bzw. Blätter beim Crossover ausgewählt werden [5]. Anpassungen der Individuenrepräsentation finden sich für Genetische Algorithmen unter anderem in [105, 101, 120]. Bei der Genetischen Programmierung können aus Teilbäumen, die sich im Lauf der Evolution als sinnvoll erwiesen haben, neue Funktionen erzeugt werden, die dann durch einen Knoten repräsentiert werden können [94, 3].

Die zweite Gruppe adaptiver Verfahren bezieht sich auf einzelne Individuen. Die anzupassenden Parameter sind in diesem Fall genau einem Individuum zugeordnet. Der Vorteil dieser Methode besteht darin, dass sich einzelne Individuen in unterschiedlichen Bereichen des Suchraums befinden können und in diesen für einen Fortschritt unterschiedliche Parameterwerte nötig sind. Bei den Genetischen Algorithmen kann zum Beispiel für jedes Individuum ein Parameter angegeben werden, der die Stelle des nächsten Crossover festlegt [102]. Bei baumorientierten Verfahren kann der genetische Operator in Abhängigkeit der Tiefe des Baumes gewählt werden [54]. Die Adaptation erfolgt somit nicht über einen expliziten Parameter, sondern wird implizit durch das Genom des Individuums vorgegeben.

Der dritte Ansatzpunkt für Adaptation ist die Komponentenebene, also ein Teil eines Individuums. Am bekanntesten sind hier die Strategieparameter der Evolutionsstrategien, die für jede einzelne Komponente eines Individuums vorhanden sein können [103]. In der Genetischen Programmierung existieren Crossover-Operatoren, bei denen für jeden Knoten eines Baumes eine Wahrscheinlichkeit mitgeführt wird, mit der ein Crossover an genau dieser Stelle stattfindet [6].

Ein zweites Kriterium zur Unterscheidung adaptiver Verfahren ist die Art der Parameterberechnung. Es werden absolute und selbstadaptive Verfahren unterschieden. Die erste Gruppe umfasst alle Ansätze, bei denen die Parameter aufgrund fest vorgegebener Regeln variiert werden. In der zweiten Gruppe unterliegen die Parameter selbst wieder einem evolutionären Prozess. Ein Beispiel für den Unterschied ist die Anpassung der Schrittweiten bei Evolutionsstrategien. Dies kann entweder selbstadaptiv [103] oder absolut über eine Kovarianzmatrix-Adaptation [39] erfolgen.

7.2.2 Population-Level Dynamic Probabilities (Pdp)

Das erste Adaptationsverfahren, das hier vorgestellt werden soll, berücksichtigt die bisherigen Erfolge aller genetischen Operatoren innerhalb des GP-Laufs. Als Erfolg gilt es, wenn durch die Variation mit einem genetischen Operator ein Individuum erzeugt wurde, das eine bessere Fitness hat, als das Individuum, aus dem es hervorgeht. Beim *Crossover* muss die Fitness besser sein als die beider Eltern.

Für jeden Operator i wird ein Quotient aus der Anzahl der Erfolge $success_i$ zum Quadrat und der Anzahl der Anwendungen $used_i$ gebildet[1]. Anstelle des initialen Werts p_i wird die Operatorwahrscheinlichkeit proportional zu diesen Quotienten bestimmt.

Würde ausschließlich dieses Verfahren benutzt, wären zu Beginn eines GP-Laufs alle Quotienten gleich Null. Dies würde bedeuten, dass kein Operator ausgewählt werden könnte. Würde außerdem der Quotient eines Operators während eines GP-Laufs durch ständigen Misserfolg so klein, dass er nicht mehr ausgewählt würde, könnte der Quotient nicht mehr vergrößert werden, selbst wenn der Operator zu einem späteren Zeitpunkt deutlich besseren Einfluss auf die Evolution hätte. Aus diesen beiden Gründen wird das oben beschriebene Verfahren nur in 80 Prozent aller Fälle angewandt. In den verbliebenen 20 Prozent wird einer der genetischen Operatoren gleichverteilt ausgewählt.

Bei n Operatoren und Quotienten der Form $r_i = \frac{success_i^2}{used_i}$ ergibt sich ein neuer Parametersatz aus:

$$\hat{p}_i = \left\lfloor \frac{20}{n} \right\rfloor + \left\lfloor r_i \frac{(100 - n \left\lfloor \frac{20}{n} \right\rfloor)}{\sum_{j=1}^{n} r_j} \right\rfloor$$

Die Anzahl der Erfolge geht quadratisch in den Quotienten ein, da der Unterschied zwischen Erfolgszahl $success_i$ und Anwendungszahl $used_i$ eines Operators wesentlich höher ist als die der Erfolgszahlen verschiedener Operatoren untereinander. Werden alle Operatoren ungefähr gleich oft angewendet, würde eine nicht-quadratischer *success*-Wert im Zähler somit zu annähernd gleichen Quotienten bei allen Operatoren führen.

7.2.3 Fitness Based Dynamic Probabilities (Fbdp)

Dem *Fbdp*-Adaptationsverfahren liegt die Überlegung zu Grunde, dass Operatoren, die sich bei einem bestimmten Graphen als erfolgreich erwiesen haben, bei demselben oder ähnli-

[1] Um nicht durch null zu teilen, wird $used_i$ immer mit eins initialisiert

chen Graphen Erfolg versprechender sind als genetische Operatoren, die auf diesem Graphen nur Misserfolge vorzuweisen haben. Zur Realisierung dieser Überlegung wird eigentlich ein Ähnlichkeitsmaß auf Graphen, also dem Genotyp eines Individuums benötigt. Da dieses aber nicht vorliegt, wird statt dessen der Phänotyp in Form des Fitnesswerts eines Individuums verwendet.

Wenn im GP-Lauf ein Operator Erfolg hat, wird die Fitness des Elter-Individuums (beim Crossover die Fitness des besseren Elter) zusammen mit einem Verweis auf den verwendeten Operator in ein nach Fitnesswerten sortiertes Feld geschrieben. Wird nun später zur Bestimmung eines zu verwendenden genetischen Operators ein neuer Parametersatz $\hat{p}_i$ gesucht, wird die Fitness des Elterindividuums in dem sortierten Feld gesucht und die 100 Elemente des Feldes betrachtet, die dieser Position bezüglich der Fitnessdifferenz zwischen neuem Individuum und Feldelement am nächsten liegen. Aus den Häufigkeiten der bei diesen 100 Elementen verwendeten Operatoren ergeben sich die neuen $\hat{p}$-Werte.

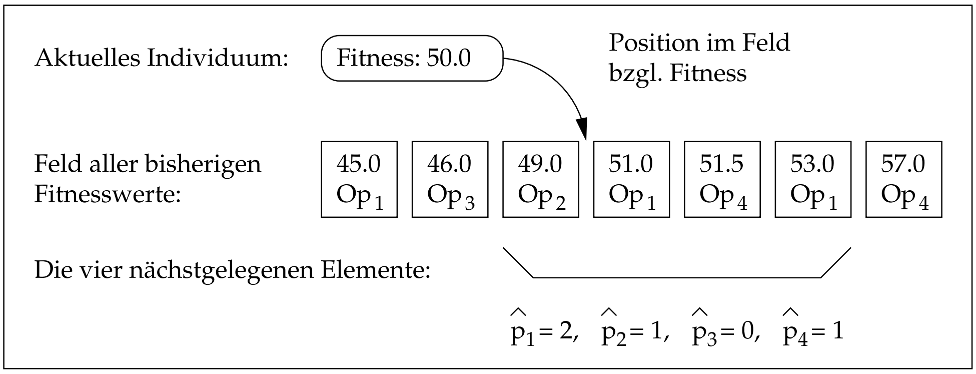


Abbildung 7.1: Beispiel zur Berechnung neuer Operatorwahrscheinlichkeiten nach dem *Fbdp*-Verfahren

Abbildung 7.1 zeigt ein Beispiel für dieses Verfahren. Aus Gründen der Übersicht werden statt 100 nur die nächsten vier Elemente betrachtet. Die Summe der $\hat{p}$-Werte liegt somit nicht bei 100, sondern ebenfalls bei vier.

Wird das beste Individuum variiert, sind die 100 Fitnesswerte des Feldes alle nicht besser als der eigene. Es werden zur Auswahl des genetischen Operators demnach nur vorangegangene Mutationen betrachtet, die Individuen mit schlechterer Fitness als Elter-Individuen verwendeten. Für das neue Individuum könnte daher ein anderer genetischer Operator nötig sein, als für die vorigen. Damit dieser benutzt werden kann, obwohl er bei den 100 nächstgelegenen Feldern nicht auftaucht, wird die neue Wahrscheinlichkeitsverteilung für die genetischen Operatoren – wie im *Pdp*-Fall auch (Abschnitt 7.2.2) – nur zu 80 Prozent aus den eben berechneten $\hat{p}$-Werten gewählt und zu 20 Prozent gleichverteilt aus allen Operatoren.

7.2.4 Individual-Level Dynamic Probabilities (Idp)

Das dritte adaptive Verfahren berechnet die Wahrscheinlichkeiten auf dem Individuen-Level. was bedeutet, dass die Operatorwahrscheinlichkeiten für ein Individuum sich aus diesem zugeordneten Parametern ergeben.

Wurden beim *Pdp*-Verfahren noch alle während des Laufs durchgeführten Mutationen für die Berechnung der neuen Operatorwahrscheinlichkeiten herangezogen, so berechnen sich diese jetzt aus den Mutationen, die bereits mit diesem Individuum durchgeführt wurden. Zusätzlich gehen die Parametersätze der Vorfahren in abgeschwächter Form ebenfalls in die Berechnung ein.

Zu jedem Individuum j gibt es für jeden genetischen Operator i einen Zähler cnt_j^i, der angibt, wie oft hintereinander auf dieses Individuum der Operator j angewendet wurde und nicht zu einer Fitnessverbesserung führte. Sobald eine Anwendung des Operators zu einem Erfolg führt, wird der Zähler auf Eins zurückgesetzt. Die Operatorwahrscheinlichkeiten berechnen sich nun aus den Verhältnissen der Zählerstände der genetischen Operatoren untereinander, wobei größere Zähler für geringere Wahrscheinlichkeiten stehen. Ist ein Operator also schon häufig gescheitert, ist die Wahrscheinlichkeit für seine Benutzung geringer als die eines Operators, der gerade bei demselben Individuum eine Fitnessverbesserung erzeugt hat.

Um das Aussterben eines Operators für ein bestimmtes Individuum auszuschließen, werden die Wahrscheinlichkeiten auch in diesem Fall zu 20 Prozent gleichverteilt und nur zu den verbliebenen 80 Prozent mit dem hier beschriebenen Verfahren bestimmt.

Die entsprechende Formel zur Berechnung der $\hat{p} - Werte$ lautet

$$\hat{p}_i = \left\lfloor \frac{20}{n} \right\rfloor + \left\lfloor \frac{(c_{max} - cnt_j^i)(100 - n\left\lfloor \frac{20}{n} \right\rfloor)}{\sum_{k=1}^{n} \left(c_{max} - cnt_j^k \right)} \right\rfloor, \quad \text{mit} \quad c_{max} = 1 + \max_{1 \le k \le n} cnt_j^k.$$

Der vordere Summand steht hierbei für die 20 Prozent gleichverteilten Wahrscheinlichkeitswerte und der Term $c_{max} - cnt_j^i$ sorgt dafür, dass den kleinsten Zählerständen die höchsten Wahrscheinlichkeiten zugeordnet werden. Der Rest des zweiten Summanden sorgt dafür, dass die Summe der $\hat{p}$-Werte möglichst nahe an 100 liegt[1].

Das Erzeugen neuer Individuen erfolgt beim *Idp*-Verfahren nach folgendem Prinzip:

1. Wenn eine Selektion stattgefunden hat, wird das Sieger-Individuum j an die Stelle des Verlierer-Individuums k kopiert und es werden die $\hat{p}$-Werte des Siegers ermittelt.

2. Über die $\hat{p}$-Werte wird ein Operator i zur Variation des kopierten Individuums k ausgewählt und angewendet.

3. Es wird die Fitness des neu erzeugten Nachkommens ermittelt.

[1] Wie bereits früher erläutert, benutzt *GGP* nur ganze Zahlen als $\hat{p}$-Werte.

4. Der zum verwendeten Operator i des Elter-Individuums j gehörende Zähler cnt_j^i wird angepasst. Hat das neue Individuum eine bessere Fitness als der Vorfahre, wird der Zähler auf eins gesetzt, ansonsten wird er um eins erhöht.

5. Der Nachkomme k erbt die Zählerstände des Elter-Individuums j, wobei alle Zähler nach folgender Formel einander angenähert werden:

$$cnt_k^i = \frac{1}{2}\left(cnt_j^i + \frac{1}{n}\sum_{l=1}^{n} cnt_j^l\right)$$

6. Wird als Operator *Crossover* ausgewählt, bei dem das neue Individuum sowohl aus dem Genom des Turniergewinners als auch des Verlierers besteht, werden trotzdem nur die Zähler des Siegers berücksichtigt.

Die Zählerstände werden nicht unverändert von Eltern weiter vererbt, da die Nachkommen sich von Generation zu Generation mehr vom Urahn unterscheiden und somit Operatoren, die bei diesem gut/schlecht waren, sich bei Nachkommen deutlich anders verhalten können.

7.2.5 Vergleichsparametrisierungen

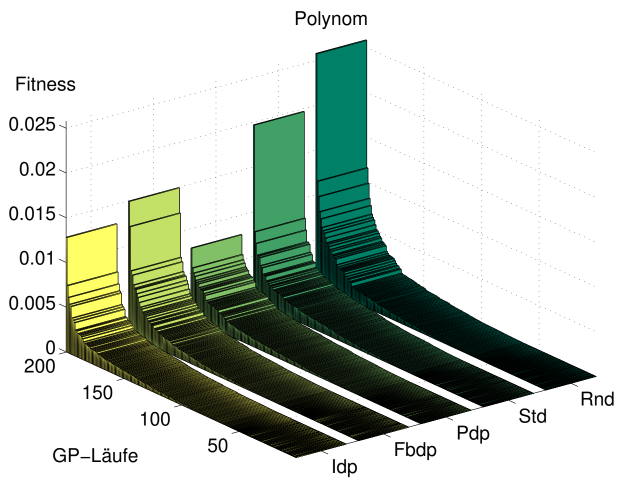
Zur Beurteilung der drei vorgestellten Verfahren werden zwei weitere Parametrisierungen zum Vergleich herangezogen und anhand der Testprobleme aus Kapitel 5 untersucht. In beiden Fällen werden die Wahrscheinlichkeiten zu Beginn eines Laufs einmalig festgelegt und danach nicht mehr verändert. Zum einen werden die in Kapitel 5 verwendeten Standardparametrisierungen für die Operatorwahrscheinlichkeiten verwendet, die in den folgenden Diagrammen als *Std* bezeichnet werden. Zum anderen wird eine Zufallsparametrisierung *Rnd* vorgenommen, die hier näher erläutert werden soll:

Wenn in einem GP-Lauf n verschiedene genetische Operatoren eingesetzt werden, jeder zugehörige $\hat{p}_i$-Wert zwischen 0 und 100 liegen muss und die Summe aller $\hat{p}$-Werte 100 sein soll, so gibt es $\frac{(n+99)!}{100!(n-1)!}$ unterschiedliche Parametrisierungen mit ganzen Zahlen. Zu Beginn eines GP-Laufs wird eine davon ausgewählt, alle sind dabei gleich wahrscheinlich.

7.3 Ergebnisse der Adaptationsverfahren

Zu jedem der drei adaptiven Verfahren (*Pdp*, *Fbdp* und *Idp*) und den zwei Vergleichsparametrisierungen (*Std* und *Rnd*) wurden für alle Testprobleme (*Polynom*, *Two-Boxes*, *Sinus*, *Klassifikation*, *Artificial Ant* und *Lawnmower*) aus Kapitel 5 jeweils 200 Läufe durchgeführt. Die Parametrisierungen entsprachen denen aus Abschnitt 5, die Operatorwahrscheinlichkeiten wurden gemäß den in diesem Kapitel beschriebenen Verfahren gewählt. In den Abbildungen 7.2 bis 7.7 sind die jeweils besten Fitnesswerte aller Läufe dargestellt. Zur besseren Übersicht wurden die jeweils 200 Werte einer Adaptationsvariante zu einem Testproblem der Größe nach sortiert, so dass Unterschiede zwischen den Verfahren besser zu erkennen sind.

Die in Abbildung 7.2 dargestellten Vergleichsläufe zum *Polynom*-Problem zeigen, dass die drei adaptiven Verfahren in diesen Experimenten im Durchschnitt geringfügig besser sind, als die zufällige oder gleichverteilte Operatorwahl. Ein Wilcoxon-Rangsummentest konnte für ein Konfidenzniveau von 99 Prozent jedoch keine signifikanten Unterschiede zwischen den Ergebnissen der einzelnen Verfahren feststellen, so dass ein Schluss von den 200 Experimenten auf den allgemeinen Fall nicht möglich ist. Die in der linken Grafik abgebildeten Fitnesswerte sind alle relativ nahe am Optimum. Es zeigt sich somit, dass das Testproblem für GP einfach zu lösen ist und adaptive Verfahren nicht nötig sind.

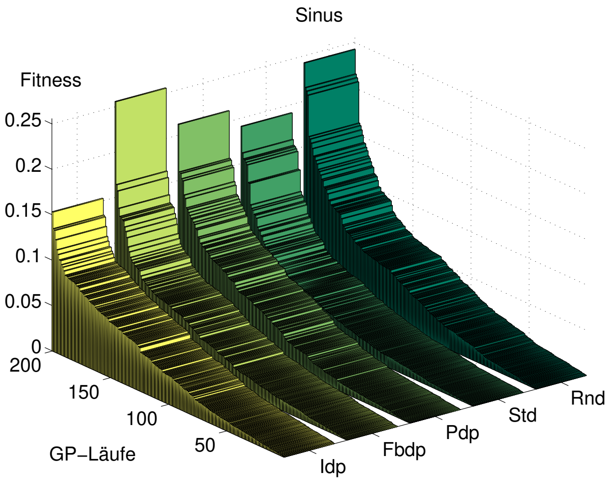


	min	max	mean
Std	0	0.01971	0.00089
Pdp	0	0.00784	0.00079
Fbdp	0	0.01497	0.00085
Idp	0	0.01282	0.00070
Rnd	0	0.0258	0.00110

Abbildung 7.2: Die Ergebnisse der Testläufe zum *Polynom*-Problem

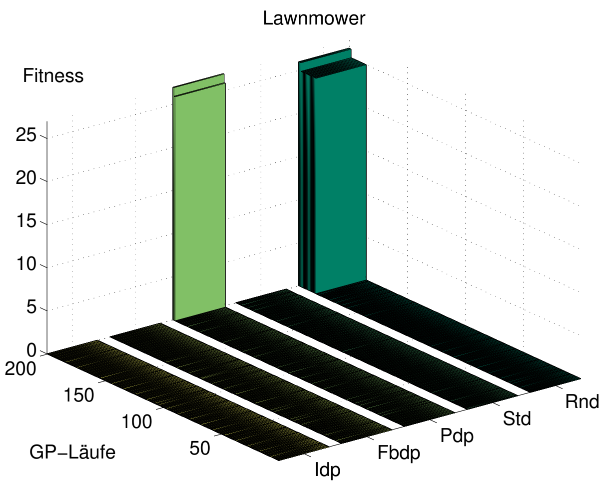
Beim *Sinus*-Problem, dessen Ergebnisse in Abbildung 7.3 zusammengefasst sind, ergibt sich ebenfalls kein klares Bild. Während die gleichverteilte Operatorwahl (*Std*) im Durchschnitt die besten Ergebnisse erzielen konnte, wurde die insgesamt beste Lösung mit dem *Pdp*-Verfahren evolviert. Betrachtet man wiederum das jeweils schlechteste Ergebnis der 200 Läufe, so war das Ergebnis des *Idp*-Verfahrens von diesen noch am besten. Ein Wilcoxon-Rangsummentest konnte auch hier in den meisten Fällen keine signifikanten Unterschiede zwischen den einzelnen Verfahren feststellen – lediglich *Std* war signifikant besser als *Idp* und *Rnd* und *Fbdp* ist signifikant besser als *Rnd*.

Für das *Lawnmower*-Problem ergibt sich ein signifikanter Unterschied zwischen einer zufälligen Verteilung der Operatorwahrscheinlichkeiten *Rnd* und allen anderen Verfahren. In der Tabelle von Abbildung 7.4 steht in der Spalte *succ.* die Anzahl der Läufe, in denen eine Lösung des Problems gefunden wurde. Der im Vergleich zu den restlichen Verfahren kleine Wert von 185 beim *Rnd*-Verfahren ist signifikant schlechter als die anderen Werte. Dies wird von einem entsprechenden Wilcoxon-Rangsummentest bestätigt. Die Spalte *impr.* gibt an, wie viele Fitnessauswertungen in den erfolgreichen Läufen durchschnittlich nötig



	min	max	mean
Std	0.00013	0.19124	0.03207
Pdp	0.00008	0.21231	0.04045
Fbdp	0.00045	0.25603	0.03743
Idp	0.00025	0.15318	0.03924
Rnd	0.00042	0.24252	0.04866

Abbildung 7.3: Die Ergebnisse der Testläufe zum *Sinus*-Problem

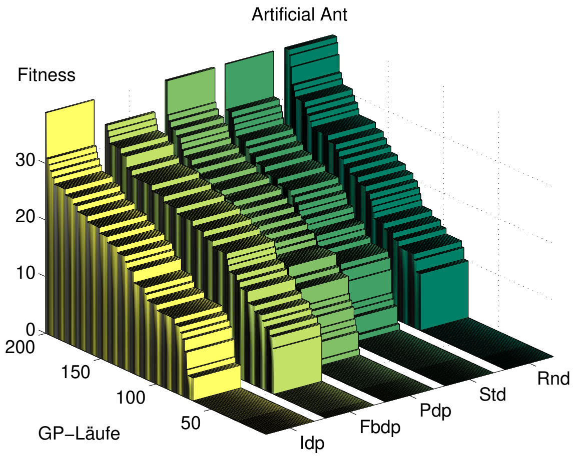


	succ.	impr.	effort (evals)
Std	200	1835	4572 (4572)
Pdp	198	1450	4729 (2417)
Fbdp	200	1790	5382 (4572)
Idp	200	1700	4431 (4041)
Rnd	185	2576	9662 (2358)

Abbildung 7.4: Die Ergebnisse der Testläufe zum *Lawnmower*-Problem

waren, bis eine Lösung gefunden wurde. Auch hier zeigt sich ein deutlicher Abstand zwischen *Std* und den restlichen Verfahren, die alle wiederum recht nahe beieinander liegen. Die *effort*-Spalte wird der Vollständigkeit halber mit angegeben, sollte aber entsprechend Abschnitt 5.4 nicht zu stark gewichtet werden.

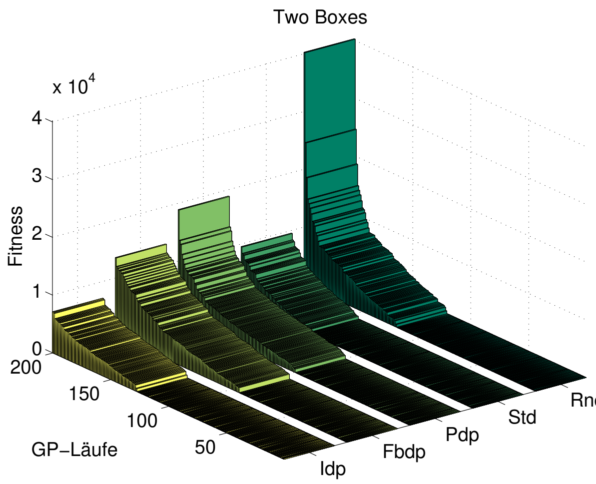
Beim *Artificial Ant*-Problem erweisen sich alle adaptiven Verfahren als kontraproduktiv. Die Tabelle aus Abbildung 7.5 zeigt, dass sowohl mit einer gleichverteilten Operatorwahl als auch mit einer zufälligen, bessere Ergebnisse erzielt werden können. Ein Wilcoxon-Rangsummentest belegt, dass *Std* hierbei auch signifikant besser ist als alle anderen Verfah-



	succ.	impr.	effort (evals)
Std	85	55389	570720 (30000)
Pdp	67	50657	792887 (45000)
Fbdp	47	47745	1286922 (31000)
Idp	66	58985	854942 (28000)
Rnd	77	57818	743067 (35000)

Abbildung 7.5: Die Ergebnisse der Testläufe zum *Artificial Ant*-Problem

ren. Eine mögliche Erklärung für dieses Verhalten liegt in der zerklüfteten Fitnesslandschaft des Problems. In [66] zeigen LANGDON und POLI, dass GP bei diesem Problem nur unwesentlich besser abschneidet als eine Zufallssuche. Beim Übergang von einem Graphen zu einem weiteren mit besserer Fitness ist es offensichtlich eher störend, Operatoren zu bevorzugen, die in früheren Fitnessauswertungen erfolgreich waren.

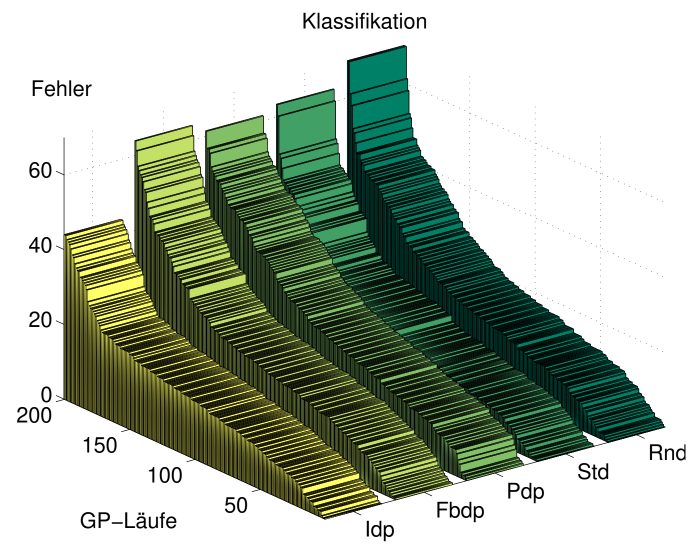


	succ.	impr.	effort (evals)
Std	142	25717	61037 (4354)
Pdp	98	32092	76597 (2221)
Fbdp	93	27879	99421 (3893)
Idp	126	27784	60462 (3012)
Rnd	119	29575	78482 (2082)

Abbildung 7.6: Die Ergebnisse der Testläufe zum *Two-Boxes*-Problem

Das *Two-Boxes*-Problem liefert ein ähnliches Bild wie *Artificial Ant*. In diesem Fall kann lediglich das *Idp*-Verfahren mit der Zufallsbelegung mithalten. Die gleichverteilte Operatorauswahl mit einer empirischen Erfolgshäufigkeit von 71 Prozent ist allen anderen Verfahren überlegen. Die Unterschiede sind nach einem Wilcoxon-Rangsummentest zwischen *Std* und den Verfahren *Fbdp*, *Pdp* und *Rnd* sowie zwischen *Idp* und den Verfahren *Fbdp* und *Pdp* signifikant.

Die Gemeinsamkeit von *Artificial Ant* und *Two-Boxes* liegt darin, dass kleine Veränderungen des Graphen oft zu sehr großen Änderungen in der Fitness eines Individuums führen können. Bei *Two-Boxes* wird die vom Graphen dargestellte mathematische Formel durch die Änderung eines Knotens immer stark verändert, da der Graph einerseits auf maximal 20 Knoten begrenzt ist und andererseits keine parametrisierten Knoten zur Verfügung stehen, die den Einfluss einzelner Knoten abschwächen könnten. Beim *Artificial-Ant*-Problem wiederum führt das Einfügen eines einzelnen für eine Bewegung zuständigen Knoten zu einem komplett veränderten Pfad. Diese Zusammenhänge deuten darauf hin, dass für einen erfolgreichen Einsatz adaptiver Verfahren bei der Operatorwahl ein stark kausaler Zusammenhang zwischen Graphvariationen und Fitnessveränderungen existieren sollte.



	min	max	mean
Std	0	127	35.5
Pdp	1	128	42.6
Fbdp	1	123	47.5
Idp	0	88	32.6
Rnd	0	140	42.1

Abbildung 7.7: Die Ergebnisse der Testläufe zum Klassifikationsproblem. Die Tabelle gibt bereits die Anzahl der Fehlklassifikationen anstelle von Fitnesswerten an.

Beim Klassifikationsproblem stellt sich bei den durchgeführten Testreihen das *Idp*-Verfahren als das Beste heraus. Die Zahlen in der Tabelle von Abbildung 7.7 stehen für die Fehlklassifikationen innerhalb der Läufe, im Diagramm sind die Fitnesswerte abgetragen. *Idp* ist im Schnitt besser als alle anderen Verfahren, hat bei den 200 Läufen Individuen gefunden, die alle Daten des Problems richtig klassifizieren und die besten Individuen der schlechtesten Läufe waren deutlich besser als bei den restlichen Verfahren. Ein Wilcoxon-Rangsummentest hat ergeben, dass diese Unterschiede gegenüber *Fbdp*, *Pdp* und *Rnd* signifikant sind. Dieselben signifikanten Unterschiede gelten für *Std* gegenüber den genannten Verfahren.

7.4 Kontextsensitive Auswahl der Operatoren

Im vorigen Abschnitt hat sich gezeigt, dass eine Adaptation der Operatorwahrscheinlichkeiten nur in den wenigsten Fällen Vorteile gegenüber einer gleichverteilten Auswahl der Operatoren liefert. In diesem Abschnitt wird die Auswahl der genetischen Operatoren zusätzlich von dem Aufbau der Graphen abhängig gemacht, auf die sie angewandt werden sollen. Es wird sich zeigen, dass in diesem Fall die zusätzliche adaptive Operatorauswahl zu besseren Ergebnissen führen kann.

7.4.1 Änderungen der genetischen Operatoren

Der in Abbildung 4.1 vorgestellte Grundalgorithmus des GP-Systems geht nicht weiter auf die Auswahl eines genetischen Operators ein. In der Grundversion des GP-Systems wurde einer der genetischen Operatoren ausgewählt und auf den (oder die) Graphen angewandt. Wenn der vorgegebene Operator nicht verwendet werden konnte, wurde statt dessen eine Replikation durchgeführt.

Dieser Vorgang wird nun so verändert, dass zuerst überprüft wird, ob ein Operator auf einen Graphen angewendet werden kann. Kann er nicht verwendet werden, wird statt dessen ein anderer Operator mit dem üblichen Auswahlverfahren ermittelt. Erst wenn sich gezeigt hat, dass sämtliche Operatoren nicht anwendbar sind, wird statt dessen eine Replikation durchgeführt.

Es folgt eine Aufstellung aller Situationen, in denen ein Operator scheitern kann:[1]

Knoten mutieren: Es ist kein mutierbarer Knoten vorhanden. Dies ist der Fall, wenn der Graph nur aus Verbindungen der Eingangsknoten mit den Ausgangsknoten besteht oder alle Knoten unparametrisiert sind und zusätzlich zu keinem Knoten des Graphen eine weitere Grundfunktion mit demselben Ein-/Ausgangsgrad existiert.

Knoten verschieben: Im Graph befindet sich kein (1,1)-Knoten

GP-Pfad löschen: Es kann kein GP-Pfad gelöscht werden. Entweder existiert im Graph kein solcher Pfad oder nach seiner Löschung würde der Graph gegen Definition 3.1 verstoßen.

GP-Pfad einfügen: Der Graph hat bereits die maximale Knotenzahl erreicht oder bei GP-Läufen, in denen Zyklen verboten sind, würde das Einfügen zu einem solchen führen. Außerdem kann es passieren, dass keine zwei Kanten vorhanden sind, die zum Einfügen des GP-Pfades nötig wären.

Zyklus: Hat der Graph bereits seine maximale Knotenzahl erreicht, kann ebenfalls kein Zyklus eingefügt werden. Der Operator scheitert ebenfalls, wenn nur eine Kante im Graphen vorhanden ist.

[1] Durch falsche Grundeinstellungen scheiternde Operatoren (z.B. *Knoten einfügen*, wenn keine (1,1)-Grundfunktion existiert) werden hier nicht aufgeführt.

Knoten einfügen: Der Graph hat bereits die maximale Größe erreicht.

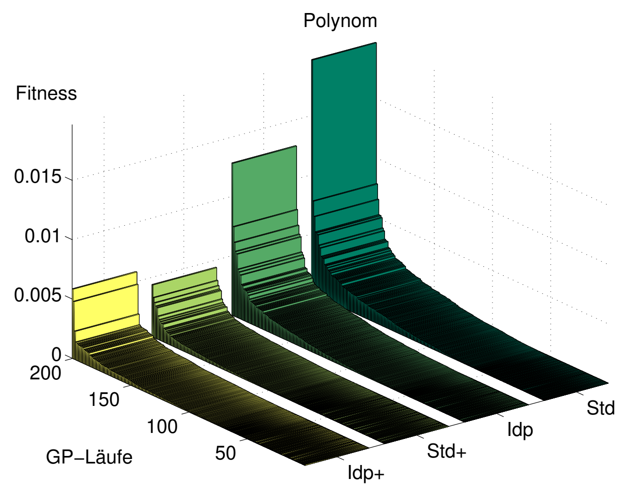
Knoten löschen: Im Graph befindet sich kein (1,1)-Knoten, der gelöscht werden könnte.

Crossover: Ursachen für ein Scheitern eines *Crossover*-Operators sind mannigfaltig und wurden bereits in Abschnitt 4.2.9 und Kapitel 6 vorgestellt.

7.4.2 Ergebnisse

Der Einfluss der zuvor beschriebenen Änderungen des Selektionsverfahrens wurde untersucht, indem für alle sechs Testprobleme jeweils 200 neue GP-Läufe mit der kontextsensitiven Operatorauswahl durchgeführt wurden. Es wurden hierbei einmal eine gleichverteilte Operatorwahrscheinlichkeit (*Std+*) und das verbesserte *Idp*-Verfahren (hier *Idp+* genannt) untersucht. Die Ergebnisse wurden jeweils mit denen der entsprechenden kontextfreien Verfahren verglichen und sind in den Abbildungen 7.8 bis 7.13 dargestellt.

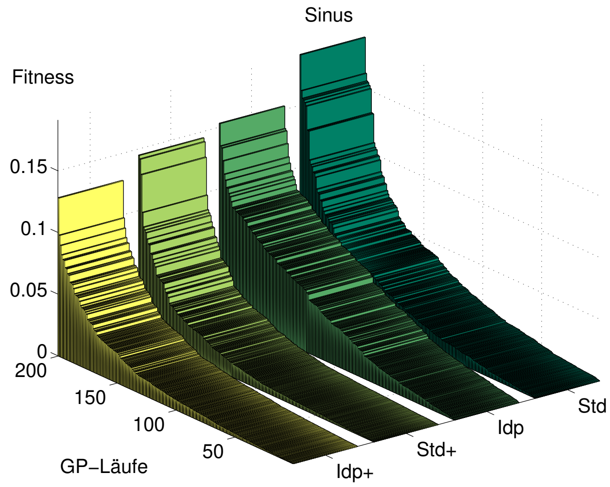
Die Ergebnisse, die beim *Polynom*-Problem erzielt wurden, sind wesentlich besser als die mit den entsprechenden kontextfreien Verfahren *Std* und *Idp* erzielten. In der Grafik von Abbildung 7.8 ist deutlich zu sehen, dass beide neuen Verfahren auch in den schlechteren Läufen besser waren als die alten. Wilcoxon-Rangsummentests ergeben, dass sowohl *Std+* als auch *Idp+* signifikant besser sind als sämtliche alte kontextfreie Verfahren. Der Unterschied zwischen den beiden neuen Verfahren wiederum ist nicht signifikant.



	min	max	mean
Std	0	0.01971	0.00089
Idp	0	0.01282	0.00070
Std+	0	0.00439	0.00027
Idp+	0	0.00586	0.00029

Abbildung 7.8: Die Ergebnisse der Testläufe zum *Polynom*-Problem

Betrachtet man die Entwicklung der Knotenzahl des jeweils besten Individuums während eines GP-Laufs, so zeigt sich, dass beim *Std*-Verfahren die durchschnittliche Größe bei 77.7 Knoten liegt, die Individuen also immer sehr nahe an die maximal erlaubte Knotenzahl heranreichen. Die Erklärung der Verbesserung durch *Std+* liegt somit darin, dass bei *Std* häufig genetische Operatoren nicht ausgeführt werden können, da die resultierenden Graphen gegen die erlaubte Maximalgröße verstoßen würden (z.B. *Pfad einfügen*). Bei sehr vielen der durchgeführten Graphvariationen handelt es sich demnach um Reproduktion eines Elter-Individuums. Diese – in den meisten Fällen überflüssigen Operationen – werden durch *Std+* verhindert, so dass bei insgesamt 200000 Fitnessauswertungen wesentlich mehr unterschiedliche Individuen erzeugt werden.



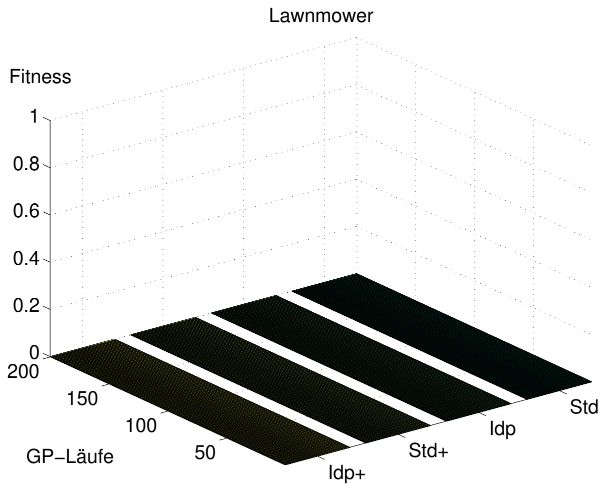
	min	max	mean
Std	0.00013	0.19124	0.03207
Idp	0.00025	0.15318	0.03924
Std+	0.00001	0.14506	0.01773
Idp+	0.00008	0.12816	0.01733

Abbildung 7.9: Die Ergebnisse der Testläufe zum *Sinus*-Problem

Beim *Sinus*-Problem sind die Ergebnisse ähnlich. Die kontextsensitiven Erweiterungen haben im Durchschnitt jeweils für eine ungefähre Halbierung der Fitnesswerte gesorgt. War bei den kontextfreien Varianten *Std* im Durchschnitt noch etwas besser als *Idp*, so sind die kontextsensitiven Varianten gleich gut. Die Wilcoxon-Rangsummentests zeigen wiederum einen signifikanten Unterschied zwischen den kontextsensitiven Varianten und sämtlichen alten Verfahren.

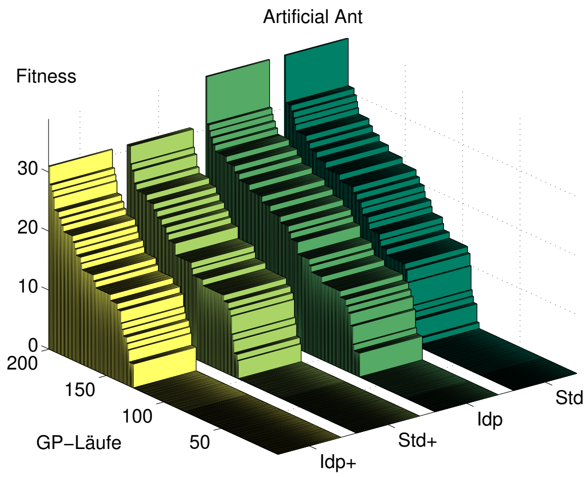
Das *Lawnmower*-Problem wurde bereits von den kontextfreien Verfahren gut gelöst. Die kontextsensitiven Varianten benötigten jedoch im Durchschnitt weniger Fitnessauswertungen als ihre kontextfreien Pendants, das *Idp+*-Verfahren war im Durchschnitt etwas besser als *Std+*.

Von allen Testproblemen sind die Auswirkungen der neuen Verfahren beim *Artificial Ant*-Problem am größten. Das *Idp+*-Verfahren konnte die Anzahl der erfolgreichen Läufe fast verdoppeln und dabei die Anzahl der benötigten Fitnessauswertungen um ein Viertel reduzieren. Waren die 200 Testläufe von *Std* noch mit einer Erfolgshäufigkeit von 42.5 Prozent besser als die von *Idp* mit 33 Prozent, so ist das Verhältnis von *Std+* zu *Idp* mit 51.5 Prozent zu 63 Prozent genau umgekehrt. Dieser Unterschied wird sogar ebenfalls von einem Wilcoxon-Rangsummentest als signifikant belegt. Das *Idp+*-Verfahren ist für das *Artificial Ant*-Problem bei der gewählten Parametrisierung signifikant besser geeignet als sämtliche andere getestete Verfahren.



	succ.	impr.	effort (evals)
Std	200	1835	4572 (4572)
Idp	200	1700	4431 (4041)
Std+	200	1645	4686 (3568)
Idp+	200	1574	3760 (4326)

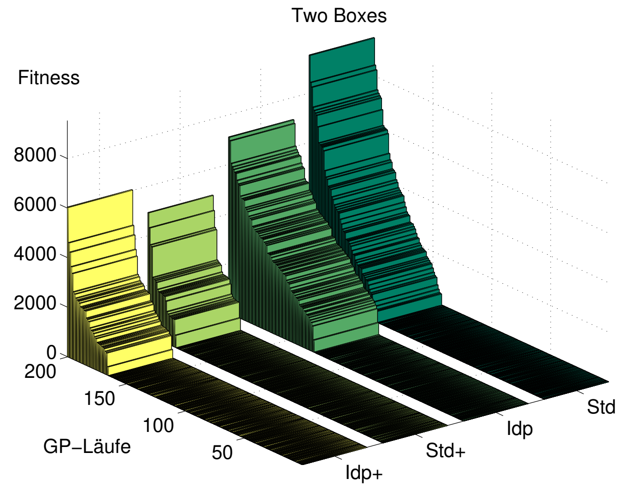
Abbildung 7.10: Die Ergebnisse der Testläufe zum *Lawnmower*-Problem



	succ.	impr.	effort (evals)
Std	85	55389	570720 (30000)
Idp	66	58985	854942 (28000)
Std+	103	49282	378071 (36000)
Idp+	126	45142	341525 (43000)

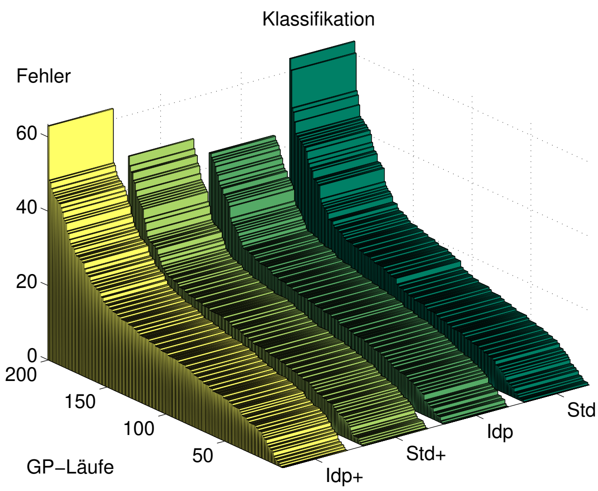
Abbildung 7.11: Die Ergebnisse der Testläufe zum *Artificial Ant*-Problem

Auch beim *Two-Boxes*-Problem sind die Ergebnisse durch die kontextsensitive Operatorauswahl deutlich besser geworden. Wiederum sind die beiden neuen Varianten *Std+* und *Idp+* signifikant besser als alle alten Varianten, die Erfolgshäufigkeit konnte in beiden Fällen deutlich gesteigert werden. Während der Abstand dieser Häufigkeiten kleiner geworden ist, benötigten die erfolgreichen GP-Läufe bei *Idp+* deutlich weniger Fitnessauswertungen, um eine Lösung zu finden.



	succ.	impr.	effort (evals)
Std	142	25717	61037 (4354)
Idp	126	27784	60462 (3012)
Std+	176	22575	36886 (2687)
Idp+	165	16075	35225 (2566)

Abbildung 7.12: Die Ergebnisse der Testläufe zum *Two-Boxes*-Problem



	min	max	mean
Std	0	127	35.5
Idp	0	88	32.6
Std+	0	98	32.5
Idp+	0	126	37

Abbildung 7.13: Die Ergebnisse der Testläufe zum Klassifikationsproblem. Die Tabelle gibt die Anzahl der Fehlklassifikationen anstelle von Fitnesswerten an.

Die einzige Ausnahme von dem bisher gewonnenen Bild liefert das *Klassifikations*-Problem. Während sich *Std+* verbessern konnte, fällt *Idp+* gegenüber der kontextfreien Variante deutlich ab. Dies scheint darauf hinzudeuten, dass bei diesem Problem ein höherer Replikationsanteil von Nutzen sein kann.

7.5 Zusammenfassung

In diesem Kapitel wurde beschrieben, auf welcher Basis während eines GP-Laufs entschieden wird, welcher genetische Operator angewendet werden soll. Es wurden drei verschiedene Verfahren vorgestellt, mit denen auf unterschiedliche Weise diese Wahl adaptiv vorgenommen werden kann. Alle drei Verfahren waren bei den sechs betrachteten Testproblemen nicht signifikant besser als eine gleichverteilte Operatorauswahl, in einigen Fällen konnten die Verfahren ebenfalls nicht die Ergebnisse einer zufälligen Initialisierung der Operatorwahrscheinlichkeiten erreichen.

Dies kann mehrere Ursachen haben:

- Eine sinnvolle Adaptation der Operatorwahrscheinlichkeiten kann nur dann stattfinden, wenn eine Korrelation zwischen der bisherigen Performanz eines Operators und seinen zu erwartenden Fitnessveränderungen existiert. Eine Untersuchung dieses Zusammenhangs ist extrem schwierig und ließe sich zunächst am Einfachsten für künstlich erzeugte Probleme durchführen. Bei Problemen ohne Korrelation sind die Ergebnisse der adaptiven Verfahren schlechter als bei anderen Benchmarks. In diesen Fällen führen kleine Änderungen am Graphen im Normalfall zu extremen Veränderungen der Fitness (*Artificial Ant*/*Two-Boxes*).

- Eine weitere Voraussetzung für eine Adaptation ist, dass bereits Vergleichswerte vorliegen, mit denen sich eine sinnvolle Neubestimmung der Operatorwahrscheinlichkeiten begründen lässt. So lange diese am Anfang eines GP-Laufs noch nicht vorliegen, muss ein adaptives Verfahren mit Operatorwahrscheinlichkeiten arbeiten, die im Zweifelsfall schlechter geeignet sind als die eines nicht-adaptiven Verfahrens. Aus diesem Grund kann die Fitnessentwicklung bei nicht-adaptiven Verfahren am Anfang eines GP-Laufs schneller sein als bei einem adaptiven Verfahren. Kann ein Problem ohne Adaptation gut gelöst werden, ist dieser Vorsprung eventuell nicht mehr einholbar.

In Abschnitt 7.4 wurde beschrieben, warum verschiedene genetische Operatoren scheitern können. Wenn Operatoren häufiger nicht angewendet werden können, führt dies automatisch zu einer Erhöhung des Replikationsanteils bei den Individuen der Population. Um diesen Effekt zu vermeiden, wird bei einer kontextsensitiven Auswahl der genetischen Operatoren vor der Veränderung eines Individuums darauf geachtet, dass nur Operatoren eingesetzt werden, die tatsächlich ein der Definition eines GP-Graphen (3.1) entsprechendes Individuum erzeugen können. Für die zur Auswahl benutzten Operatorwahrscheinlichkeiten bedeutet das, dass die Wahrscheinlichkeiten der ungeeigneten Operatoren auf Null gesetzt und die frei werdenden Anteile auf die verbliebenen Operatoren verteilt werden.

Durch dieses Verfahren konnten die Ergebnisse sowohl von adaptiven als auch von nicht-adaptiven Verfahren signifikant verbessert werden. Das adaptive Verfahren *Idp+* war mit einer Ausnahme dem *Std+*-Verfahren überlegen. Die Unterschiede waren in den meisten

Fällen jedoch nicht signifikant. Die Ergebnisse zeigen, dass eine Wahrscheinlichkeitsadaptation zusammen mit einer kontextsensitiven Wahl der Operatoren zu deutlichen Verbesserungen des GP-Grundalgorithmus führen kann.

Kapitel 8

Schrittweitensteuerung genetischer Operatoren

Bei der Anwendung der genetischen Operatoren werden zuerst zwei Individuen der Population ausgewählt, von denen das eine durch eine Variation des anderen[1] ersetzt wird. Hierzu wird mit den in Kapitel 7 vorgestellten Methoden genau ein Operator ausgewählt. In anderen GP-Systemen, in denen der *Crossover*-Operator eine exponierte Stellung einnimmt (z.B. [57, 9]), wird oftmals dieser und ein Mutationsoperator gemeinsam angewendet. In diesem Kapitel soll untersucht werden, ob die sequenzielle Anwendung mehrerer Operatoren vor der Fitnessberechnung Einfluss auf den Evolutionsverlauf hat und eventuell ein stark kausaler Zusammenhang zwischen der Anzahl der angewandten Operatoren und der Fitnessveränderung besteht.

Zunächst wird beschrieben, wie mehrere Operatoren nacheinander auf ein Individuum angewendet werden können. Daran anschließend, folgt die Beschreibung der Experimente, die die Motivation zur Untersuchung der sequenziellen Anwendung mehrerer Operatoren liefern. Abschließend werden adaptive und nicht-adaptive Verfahren vorgestellt und mit Hilfe der bekannten Testprobleme untersucht.

8.1 Sequenzielle Anwendung mehrerer genetischer Operatoren

Nach der Auswahl eines Individuums als Turniergewinner und eines als Turnierverlierer sollen n Operatoren auf das Siegerindividuum angewendet werden. Der auf diese Weise erzeugte neue Graph überschreibt dann das Verliererindividuum. Eine Sonderstellung unter den Operatoren nimmt hierbei wiederum der *Crossover*-Operator ein, da ein neues Individuum aus dem genetischen Code beider Eltern zusammengesetzt wird. Abbildung 8.1 zeigt den Algorithmus, der zur Variation der Individuen benutzt wird. Die Anzahl der hintereinander ausgeführten Operatoren wird als *Schrittweite der Variation* bezeichnet.

[1] bzw. durch eine Kombination beider beim Crossover

Gegeben:
winner {Siegerindividuum}
loser {zu überschreibendes Individuum}

Wähle Anzahl n der nacheinander auszuführenden genetischen Operatoren

For i := 1 To n

Wähle einen genetischen Operator aus

Crossover?

Ja: Lösche Teilgraphen von loser und füge an die Stelle einen von winner ein.

Nein: Wende den Operator auf loser an

∅

Abbildung 8.1: Algorithmus zur sequenziellen Anwendung mehrerer genetischer Operatoren

Zunächst wird untersucht, ob ein stark kausaler Zusammenhang zwischen der Schrittweite und den Fitnessveränderungen zwischen dem neu erzeugten Individuum und dem besseren Elter existiert. Hierzu werden von jedem der sechs in Kapitel 5 beschriebenen Testprobleme insgesamt 100 Läufe durchgeführt. Vor jeder Operatorwahl wird per Zufall eine Schrittweite festgelegt. Es wird hierzu gleichverteilt eine Zahl zwischen 1 und 15 ermittelt, die als Anzahl der sequenziell anzuwendenden Operatoren verwendet wird. Die Auswahl der einzelnen genetischen Operatoren erfolgt jeweils gleichverteilt und kontextsensitiv, also gemäß dem in Abschnitt 7.4 *Std+* genannten Verfahren. Nach jeder erfolgten Fitnessberechnung wird die Differenz zwischen der neuen Fitness und der besseren der beiden Fitnesswerte der Elterindividuen errechnet und gespeichert. Bei 100 Läufen mit je 200000 Fitnessauswertungen und Startpopulationen von je 100 Individuen ergeben sich somit für jede Schrittweite bei jedem Testproblem ungefähr 1.3 Millionen Differenzen.

Abbildung 8.2 zeigt die Anzahl der Verbesserungen und Verschlechterungen der Fitnesswerte der Untersuchungen. Zunächst ist offensichtlich, dass es bei allen Testproblemen immer deutlich mehr Fitnessverschlechterungen als -verbesserungen gibt. Bei einer Erhöhung der Schrittweite nimmt dieser Unterschied immer weiter zu. Es zeigt sich, dass die Abnahme der Verbesserungen bzw. die Zunahme der Verschlechterungen für größere Schrittweiten nicht linear ist. Vielmehr findet die stärkste Veränderung zwischen den Schrittweiten eins und fünf statt. Bei größeren Schrittweiten verflachen die Kurven zusehends.

Die Abbildungen 8.3 bis 8.5 zeigen die Verteilungsfunktionen der Fitnessverbesserungen bei den verschiedenen Testproblemen für jeweils mehrere Schrittweiten. Wenn für ein Testproblem bei einer Schrittweite von k die Fitness insgesamt n-mal verbessert wurde und die absoluten Beträge der Fitnessverbesserungen $\{d_1, d_2, \ldots, d_n\}$ sind, so steht in dem Diagramm die Kurve $SW = k$ für die Funktion $SW_k(x) = \frac{\sum_{i,d_i \leq x} 1}{n}$. Die in den Diagrammen

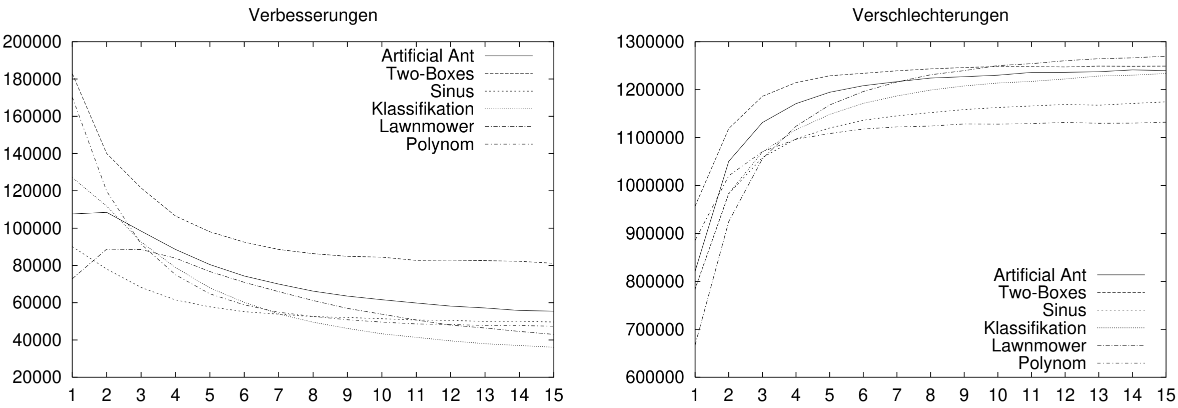


Abbildung 8.2: Anzahl der Fitnessverbesserungen und -verschlechterungen bei unterschiedlichen Schrittweiten

dargestellten Wertebereiche wurden so gewählt, dass die Unterschiede zwischen den einzelnen Kurven möglichst deutlich werden. Auf die Abschnitte der Kurven, bei denen der Funktionswert den Wert Eins annimmt, wurde in den meisten Fällen verzichtet: Die Kurven der einzelnen Schrittweiten liegen einerseits immer sehr nahe beieinander, andererseits führten die zugehörigen sehr hohen Fitnessverbesserungen während der Optimierung nur in den seltensten Fällen zu einer für die Population sehr guten Fitness. Vielmehr näherten sich extrem schlechte Individuen an den Durchschnitt der Population an.

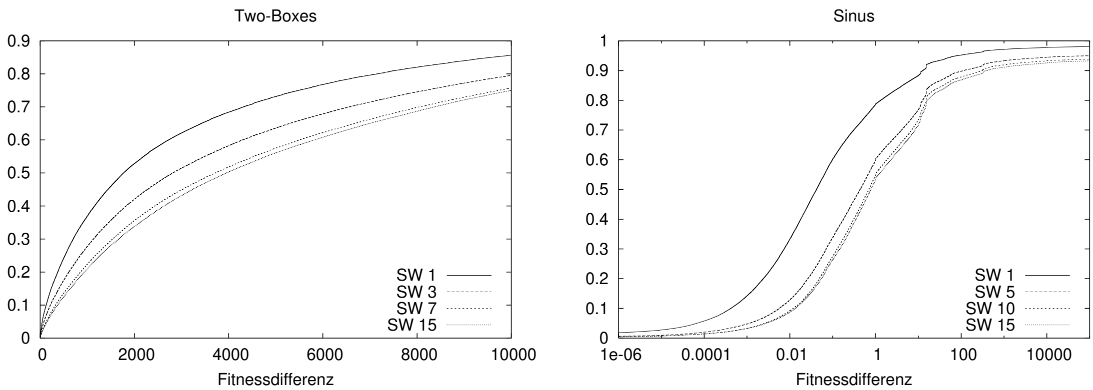


Abbildung 8.3: Verteilungsfunktionen der Fitnessverbesserungen bei unterschiedlichen Schrittweiten für *Two-Boxes* und *Sinus*

In den Abbildungen wurden nur die Kurven von wenigen Schrittweiten eingetragen. Die Verbleibenden liegen jedoch alle so ohne Überschneidungen zwischen den eingezeichneten, dass eine Darstellung lediglich der Übersichtlichkeit schaden würde. Einzige Ausnahme ist die Grafik zum *Artificial Ant*-Problem. Hier schneiden sich die Kurven bei der Schrittweite 3 und verlaufen dann genau in einer zu den übrigen Testproblemen vertauschten Reihenfolge: Die Kurve der Schrittweite 1 ist die Niedrigste, die Kurve zur Schrittweite 15 die

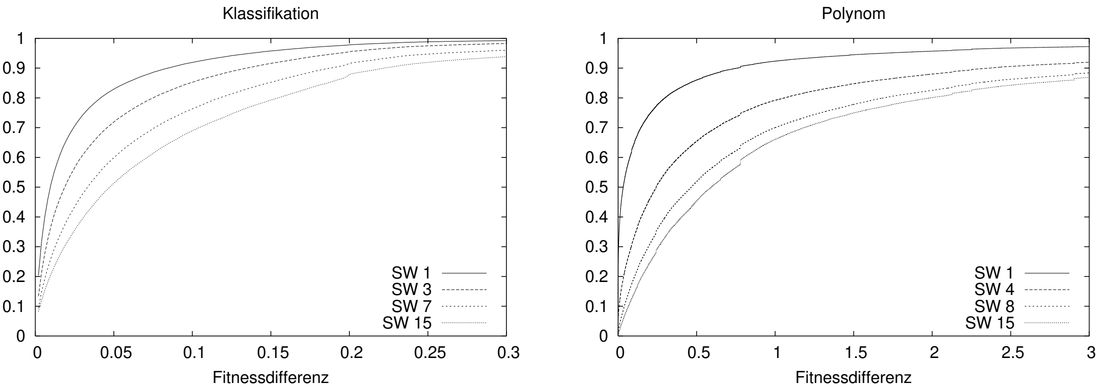


Abbildung 8.4: Verteilungsfunktionen der Fitnessverbesserungen bei unterschiedlichen Schrittweiten für *Klassifikations-* und *Polynom-*Problem

höchste. Die Treppenform der Kurven entsteht durch die zweigeteilte Fitnessfunktion, die ganze Zahlen für die Anzahl der gefundenen Nahrungsstücke und Werte kleiner eins für die Anzahl der Schritte, die eine Ameise gemacht hat, benutzt.

Die Kurven machen deutlich, dass kleinere Schrittweiten im Durchschnitt kleinere Fitnessverbesserungen zur Folge haben: Je schneller eine Kurve ansteigt, desto mehr Verbesserungen bestehen aus kleinen Fitnessdifferenzen. Werden die Fitnessverbesserungen isoliert von den restlichen Veränderungen betrachtet, liegt somit ein stark kausaler Zusammenhang zwischen Schrittweiten und Fitnessverbesserungen vor.

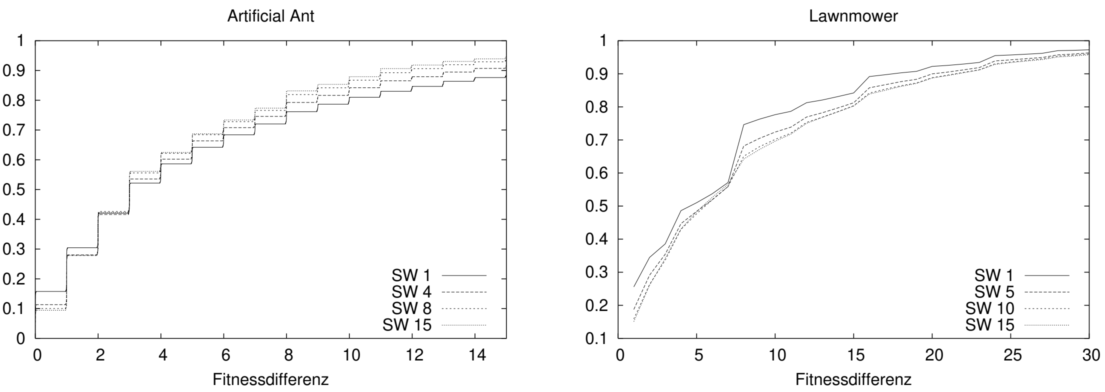


Abbildung 8.5: Verteilungsfunktionen der Fitnessverbesserungen bei unterschiedlichen Schrittweiten für *Artificial Ant* und *Lawnmower*

Zusammen mit Abbildung 8.2 ergibt sich jedoch, dass eine Erhöhung der Schrittweite nicht unbedingt zu einem günstigeren Verlauf eines GP-Laufs führt. Zwar würden die durchschnittlichen Fitnessverbesserungen bei größeren Schrittweiten ebenfalls größer, die absolute Zahl der Verbesserungen würde jedoch so stark abnehmen, dass die somit mehr hinzugekommenen Verschlechterungen der Fitness den Verlauf der Optimierung so ungünstig beeinflussen, dass das Ergebnis des GP-Laufs insgesamt schlechter als mit kleinen Schrittweiten wäre.

Bei einigen Testproblemen ist die Abnahme der erzielten Fitnessverbesserungen bereits bei einer Erhöhung der Schrittweite von eins auf zwei extrem (*Two-Boxes, Polynom*), bei anderen setzt die Reduzierung erst bei einer Schrittweite von zwei oder drei ein (*Lawnmower, Artificial Ant*). Für verschiedene Testprobleme können demnach unterschiedliche Schrittweiten sinnvoll sein.

Im folgenden Abschnitt wird ein Verfahren für die Einbindung unterschiedlicher Schrittweiten vorgestellt und anhand der Testprobleme aus Kapitel 5 näher untersucht.

8.2 Schrittweitensteuerung

Im vorigen Abschnitt wurde betrachtet, wie groß einzelne Fitnessveränderungen bei unterschiedlichen Schrittweiten der genetischen Operatoren sind. Hier wird nun untersucht, wie sich die Fitnessentwicklung während eines vollständigen GP-Laufs durch Schrittweiten verändert. Es wird also untersucht, ob sich durch den Einsatz unterschiedlicher Schrittweiten bessere Ergebnisse erzielen lassen.

Hierzu wurde der Algorithmus zur Auswahl der Schrittweite aus Abbildung 8.1 in das GP-System eingebunden. Die Anzahl der maximal erlaubten genetischen Operationen, die sequenziell ausgeführt werden, wurde auf drei beschränkt. Die vor jeder Fitnessberechnung aus diesen drei Möglichkeiten zu wählende Schrittweite wird gleichverteilt bestimmt. Eine größere maximal erlaubte Schrittweite ist – zumindest bei den hier verwendeten Testproblemen – nicht erforderlich, da der Anteil der Fitnessverbesserungen an allen Fitnessveränderungen für größere Werte bereits so klein ist (Abb. 8.2), dass ein positiver Effekt nicht zu erwarten ist. Diese Annahme wird durch Untersuchungen zur adaptiven Schrittweitensteuerung in Abschnitt 8.3 bestätigt.

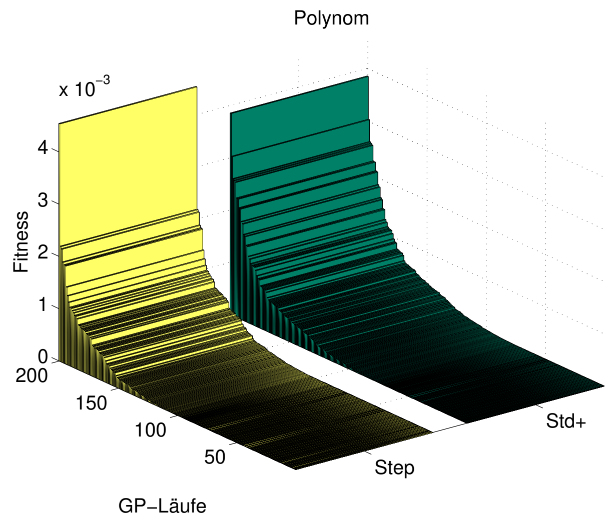
Zu den sechs in Kapitel 5 eingeführten Testproblemen werden wiederum je 200 GP-Läufe mit den dort ebenfalls beschriebenen Parametrisierungen durchgeführt. Als Vergleichswerte dienen die Ergebnisse, die mit dem kontextsensitiven Operatorwahlverfahren *Std+* aus Abschnitt 7.4 erzielt wurden.[1] Das Verfahren mit Schrittweitensteuerung verwendet ebenfalls die kontextsensitive Operatorwahl. Der Grund hierfür ist der Folgende: Die Verbesserungen, die durch die Verwendung einer kontextsensitiven Operatorwahl eintreten, beruhen darauf, dass die Zahl der Replikationen von Individuen reduziert wird. Wird eine Schrittweitensteuerung ohne kontextsensitive Operatorwahl durchgeführt, würde als ein Effekt ebenfalls die Wahrscheinlichkeit für replizierte Individuen gesenkt, da die ausschließliche Verwendung von Replikationen[2] bei Schrittweiten größer eins gering ist. An

[1] Es wurden neue Läufe mit veränderten Initialisierungen der Zufallszahlengeneratoren durchgeführt, so dass es zu Abweichungen zu den Ergebnissen aus Abschnitt 7.4.2 kommt.

[2] Inklusive der genetischen Operatoren, die keine korrekten GP-Graphen erzeugen können und daher durch Replikationen ersetzt werden.

dieser Stelle sollen jedoch nur die Auswirkungen untersucht werden, die unabhängig von der Anzahl der Replikationen sind.

In den Abbildungen 8.6 bis 8.11 sind die jeweils besten Fitnesswerte aller Läufe dargestellt. Zur besseren Übersicht wurden die 200 Werte einer Adaptationsvariante zu einem Testproblem der Größe nach sortiert, so dass Unterschiede zwischen den Verfahren besser zu erkennen sind.

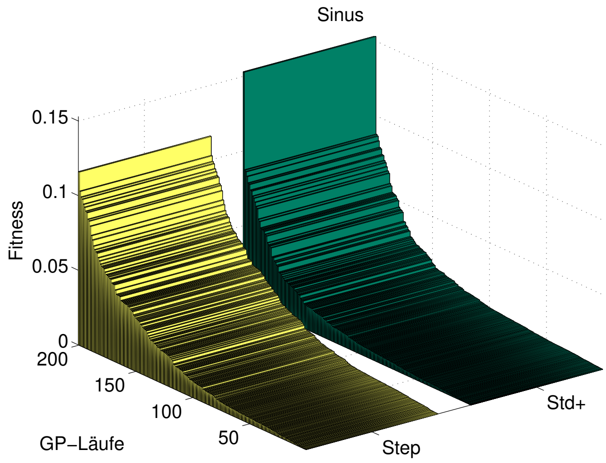


	min	max	mean
Std+	0	0.00386	0.00035
Step	0	0.00454	0.00023

Abbildung 8.6: Die Ergebnisse der Testläufe zum *Polynom*-Problem

Abbildung 8.6 zeigt die besten Fitnesswerte, die sich beim *Polynom*-Problem mit und ohne Schrittweitensteuerung in den je 200 Läufen ergeben. Durch die Verwendung der Schrittweiten konnten die bereits durch die kontextsensitive Operatorwahl gegenüber den ursprünglichen Werten aus Abschnitt 5.1.1 verbesserten Ergebnisse noch einmal (laut Wilcoxon-Rangsummentest) signifikant verbessert werden.

Die Ergebnisse, die beim *Sinus*-Problem erzielt wurden, sind zwar immer noch besser als beim Grundalgorithmus (siehe z.B. Abb. 7.9, Zeile *Std*), sie sind jedoch signifikant schlechter als ohne Schrittweitensteuerung. Eine mögliche Ursache hierfür ist der geringe Anteil der Fitnessverbesserungen an allen Fitnessveränderungen. Aus Abbildung 8.2 wird ersichtlich, dass von allen sechs untersuchten Problemen das *Sinus*-Problem für Schrittweiten größer eins die geringste Anzahl von Fitnessverbesserungen vorweist. Überträgt man die Zahlen aus der Abbildung auf einen vollständigen GP-Lauf mit konstanter Schrittweite eins, so ergibt sich ein Anteil von nur 6.8 Prozent Fitnessverbesserungen. Wenn zu gleichen Teilen ebenfalls die Schrittweiten zwei und drei benutzt würden, würde dieser Wert weiter auf 5.9 Prozent absinken. Möglicherweise wird die globale Fitnessentwicklung durch eine so geringe Anzahl der Fitnessverbesserungen so verlangsamt, dass die absoluten Werte der Fitnessverbesserungen nur noch eine sekundäre Rolle spielen.



	min	max	mean
Std+	0.00001	0.153124	0.01631
Step	0.00016	0.116115	0.02305

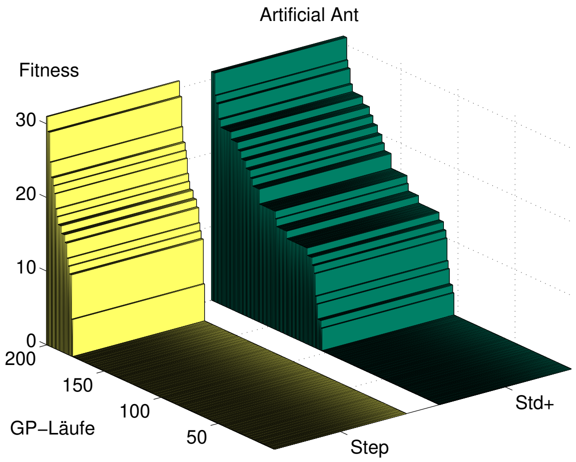
Abbildung 8.7: Die Ergebnisse der Testläufe zum *Sinus*-Problem

Auf eine Abbildung der Fitnesswerte der insgesamt 400 GP-Läufe beim *Lawnmower*-Problem wurde an dieser Stelle verzichtet, da sämtliche Läufe eine Lösung finden konnten und somit den Fitnesswert 0 erreichten. Betrachtet man allerdings die Anzahl der Fitnessauswertungen, die zum Erreichen dieses Wertes nötig waren, so kommt ein GP-Verfahren mit Schrittweitensteuerung mit weniger Berechnungen aus. Nach einem Wilcoxon-Rangsummentest ist dieser Unterschied signifikant. Wie auch beim folgenden *Artificial Ant*-Problem nimmt die Anzahl der positiven Fitnessveränderungen bei einer Erhöhung der Schrittweite zunächst zu (Abb. 8.2). Zusammen mit einer durchschnittlich größeren Fitnessverbesserung bei jeder erfolgreichen Individuenvariation (Abb. 8.5) wird somit die gesamte Optimierung beschleunigt.

	succ.	impr.	effort (evals)
Std+	200	1636	4686 (3568)
Step	200	1373	3311 (2314)

Abbildung 8.8: Die Ergebnisse der Testläufe zum *Lawnmower*-Problem

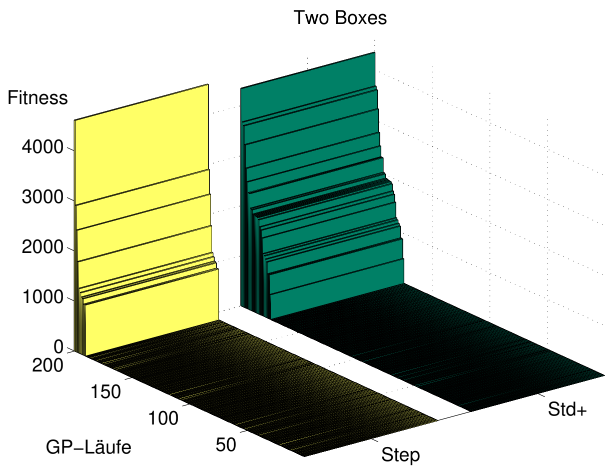
Von allen Testproblemen profitiert das *Artificial Ant*-Problem am deutlichsten von den variierten Schrittweiten. Die Anzahl der erfolgreichen Läufe konnte hier von 103 auf 177 gesteigert werden. Eine mögliche Ursache hierfür könnte die von LANGDON in [66] beschriebene zerklüftete Fitnesslandschaft darstellen: Erst durch eine größere Schrittweite wird das Verlassen eines lokalen Optimums möglich. Soll in einem Individuum zur Verbesserung der Fitness zum Beispiel ein Programmfragment der Form <**If** *Food* **then** *Move*> integriert werden, so wären ohne die Schrittweite zwei insgesamt zwei Einfügeoperationen nötig (*GP-Pfad einfügen*, *Knoten einfügen*, siehe Abschnitt 5.3.1), von denen die Erste wahrscheinlich zu einer Verschlechterung der Fitness führt und die zweite somit in den meisten Fällen über-



	succ.	impr.	effort (evals)
Std+	103	49282	378071 (36000)
Step	177	43955	167804 (10000)

Abbildung 8.9: Die Ergebnisse der Testläufe zum *Artificial Ant*-Problem

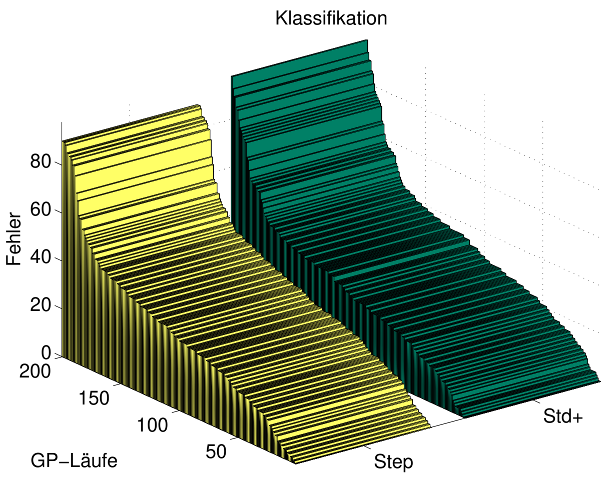
haupt nicht mehr in Frage kommt, da das Individuum die folgenden Selektionsturniere nicht mehr gewinnen würde.



	succ.	impr.	effort (evals)
Std+	174	23992	36886 (2687)
Step	190	18489	41636 (5740)

Abbildung 8.10: Die Ergebnisse der Testläufe zum *Two-Boxes*-Problem

Das *Two-Boxes*-Problem konnte ebenfalls von der Einführung der Schrittweiten profitieren. Einerseits wurde die Anzahl der gefundenen Lösungen signifikant von 174 auf 190 erhöht, andererseits benötigten die erfolgreichen GP-Läufe im Durchschnitt 23 Prozent weniger Fitnessauswertungen. An dieser Stelle wird eine weitere Eigenart des *Effort*-Wertes deutlich (vgl. Abschnitt 5.4): Obwohl das *Step*-Verfahren sowohl mehr Lösungen gefunden hat als auch im Durchschnitt weniger Fitnessauswertungen dazu benötigt hat, erzielte *Std+* den deutlich besseren *Effort*-Wert. Dies liegt daran, dass bei *Std+* die Lösungen relativ wenige oder extrem viele Fitnessauswertungen benötigen. Bei *Std+* liegt die Standardabweichung der Fitnessauswertungszahlen bei 41487.3, bei *Step* beträgt sie 30951.7. Bei *Std+* hatten bereits 57 GP-Läufe nach 2687 Fitnessauswertungen eine Lösung gefunden. Selbst wenn in allen verbleibenden 117 erfolgreichen Läufen die Lösung erst in der 200000. Fitnessauswertung gefunden worden wäre, bliebe der *Effort*-Wert derselbe.



	min	max	mean
Std+	0	98	32.5
Step	0	90	32.4

Abbildung 8.11: Die Ergebnisse der Testläufe zum Klassifikationsproblem. Die Tabelle gibt bereits die Anzahl der Fehlklassifikationen anstelle von Fitnesswerten an.

Das einzige Testproblem, bei dem sich durch die Einführung von Schrittweiten nichts ändert, ist das *Klassifikations*-Problem. Die Vorteile, die die im Durchschnitt größeren Fitnessverbesserungen bei größeren Schrittweiten bringen, werden durch den Nachteil der Verringerung der Anzahl positiver Fitnessänderungen neutralisiert.

8.3 Adaptive Schrittweitensteuerung

Die schnelle Abnahme positiver Fitnessverbesserungen bei größeren Schrittweiten deutet bereits darauf hin, dass – zumindest bei den hier untersuchten Testproblemen – eine Schrittweite größer drei keine weiteren Vorteile für einen GP-Lauf bietet. Um diese These zu untermauern, wurde ein Verfahren implementiert, mit dem Schrittweiten während eines GP-Laufs an den jeweils optimalen Wert angepasst werden können.

Hierzu werden zur Schrittweitenwahl drei benachbarte Werte verwendet - zu Beginn eines GP-Laufs eins, zwei und drei. Aus diesen Werten wird immer einer gleichverteilt ausgewählt und angewendet. Nach 1000 Fitnessauswertungen wird überprüft, welche der drei Schrittweiten zu den meisten Fitnessverbesserungen führte. Ist dieses die größte Schrittweite, so werden alle drei Werte um eins erhöht. Für die Initialisierungswerte eins, zwei und drei würde dies bedeuten, dass die meisten Fitnessverbesserungen mit einer Schrittweite von drei realisiert wurden und somit nach 1000 Fitnessauswertungen die möglichen Schrittweiten auf zwei, drei und vier gesetzt werden. Analog wird die Anzahl um eins verringert, wenn die kleinste Schrittweite für die meisten Fitnessverbesserungen verantwortlich ist. Dies geschieht allerdings nur, wenn durch die Dekrementierung alle drei möglichen Schrittweiten größer null bleiben. Dieses Verfahren entspricht HINTERDINGs Methode zur Adaptation von Populationsgrößen [41].

Zu jedem Testproblem wurden wiederum 200 Läufe durchgeführt. Die Ergebnisse sind in Tabelle 8.1 zusammengefasst.

Die Ergebnisse entsprechen zu großen Teilen denen des nicht-adaptiven Schrittweitenverfahrens. Hervorzuheben ist, dass beim *Two-Boxes*-Problem sämtliche GP-Läufe eine Lösung finden konnten. Beim Betrachten der einzelnen Läufe stellt sich heraus, dass fast immer das Tripel (1, 2, 3) für die Schrittweiten benutzt wurde. Nur in den seltensten Fällen wurde die Schrittweite 4 verwendet.

Problem	min	max	mean
Polynom	0	0.00454	0.00023
Sinus	0	0.150664	0.0230
Klassif.	0	87	30.1

Problem	succ.	impr.	effort (evals)
Mower	200	1351	2509 (2887)
Ant	174	42816	165970 (30000)
Two-Boxes	200	23225	42261 (5740)

Tabelle 8.1: Die Ergebnisse der Testprobleme mit adaptiver Schrittweitensteuerung

8.4 Zusammenfassung

Wie die Ergebnisse des Abschnitts 8.2 zeigen, sind unterschiedliche Schrittweiten bei der Wahl der genetischen Operatoren sinnvoll. Die Versuchsreihen aus Abschnitt 8.1 legen nahe, dass ein stark kausaler Zusammenhang zwischen der Anzahl der vor einer Fitnessberechnung angewandten genetischen Operatoren und dem Absolutwert einer dadurch erzielten Fitnessverbesserung besteht. Dieser Zusammenhang kann dadurch ausgenutzt werden, dass anstelle von genau einem genetischen Operator vor einer Fitnessauswertung mehrere Operatoren hintereinander auf ein Individuum angewendet werden.

Das GP-System *GGP* wurde so erweitert, dass statt eines genetischen Operators gleichverteilt ein bis drei Operatoren angewandt werden. Versuchsreihen mit den Testproblemen aus Kapitel 5 haben signifikante, zum Teil sehr große Verbesserungen bei den Ergebnissen von vier der sechs Probleme ergeben.

Eine dynamische Veränderung der erlaubten Schrittweiten während eines GP-Laufs führte zu vergleichbaren Ergebnissen wie bei statisch festgelegten Schrittweiten. Auf die Anwendung der Adaptation kann somit verzichtet werden, da der zusätzliche Aufwand zu keinen besseren Ergebnissen führt.

Kapitel 9

Populationsdiversität

Der Begriff der Diversität einer Populationen bei der Genetischen Programmierung wird in der Literatur auf mannigfaltige Weise eingeführt. Allen diesen Erklärungen ist gemein, dass sie versuchen, die Unterschiede der Individuen innerhalb der Population zu erfassen. Die Unterschiede können auf verschiedenen Ebenen betrachtet werden. KOZA beurteilte die Diversität einer Population zunächst nach der Anzahl der unterschiedlichen Individuen [57], während u. a. ROSCA [95] und LANGDON [63] die Anzahl unterschiedlicher Fitnesswerte innerhalb der Population als Hinweis auf die Diversität verwenden. Die Diversität wird hierbei über den Phänotyp, also die Fitness, bewertet.

Bei anderen Ansätzen werden Genotypen, also die Bäume, miteinander verglichen. Eine der einfachsten Formen ist der Vergleich auf unterschiedliche Baumgrößen [95]. KEIJZER überprüft in [52], ob sowohl Bäume als auch einzelne Teilbäume in der Population nur einmal vorkommen.

In weiteren Arbeiten werden Möglichkeiten vorgestellt, wie Bäume miteinander verglichen werden können. Über die Unterschiede zwischen sämtlichen Bäumen lassen sich so Abstandsmaße definieren und die Diversität einer Population somit an den Abständen der Individuen untereinander festmachen. So legt DE JONG zwei Bäume übereinander und berechnet den Abstand aus der Summe der an verschiedenen Stellen des Baumes unterschiedlichen Knoten. Jeder Unterschied steht für die Vergrößerung des Abstandes um den Wert eins. Identische Bäume haben somit den Abstand null [22]. Dieser Ansatz ist am ehesten vergleichbar mit der bei Genetischen Algorithmen verwendeten *Hamming*-Distanz [31].

Probleme dieses Verfahrens – wie z.B. variable Baumgrößen, werden von O'REILLY durch Verwenden der *edit tree*-Distanz [86] bzw. von IGEL mit der LEVENSHTEIN-Distanz [45, 46] umgangen (beide werden in [99] vorgestellt). In beiden Verfahren wird der Abstand zwischen zwei Bäumen durch die Anzahl von Elementaroperationen bestimmt, die nötig sind, um einen Baum in den anderen zu überführen. Elementaroperationen sind hierbei u.a. das Einfügen, Löschen und Mutieren von Knoten. Der Unterschied besteht darin, dass bei der *Levenshtein*-Distanz die Bäume zunächst in einen String umgeformt werden und die Operation dann darauf ausgeführt werden. Hierdurch ist es möglich, dass während der Berechnung Strings entstehen, die keinen gültigen Baum repräsentieren.

McPhee und Hopper beschreiben die Diversität über den Ursprung der einzelnen Individuen bzw. über die Herkunft einzelner Knoten in den Individuen [77]. Von der ersten Generation an wird mitverfolgt, welche Individuen aus welcher der Ursprungspopulationen hervorgehen. Die Diversität ergibt sich daraus, von wie vielen Individuen der Ursprungspopulation die aktuelle Population abstammt bzw. wie viele unterschiedliche Knoten aus der Ursprungspopulation noch vorhanden sind.

In der Genetischen Programmierung ist allgemein akzeptiert, dass eine gewisse Diversität in der Population nötig ist, um die Evolution voran zu bringen und das Verlassen lokaler Optima zu ermöglichen [9, 77]. Burke, Gustafson und Kendall untersuchen hierzu in [18] den Zusammenhang zwischen Fitnessentwicklung und Diversitätsmaßen. Zur Diversitätserhaltung gibt es verschiedene Ansätze, wie zum Beispiel die Veränderung des Selektionsdrucks durch unterschiedliche Turniergrößen oder das Einfügen neuer Individuen während eines GP-Laufs. Bersano-Begey verfolgt, welche Fitness-Cases besonders selten gelöst werden und unterstützt durch eine höhere Gewichtung dieser Fälle das Finden von Individuen, die eben diese besser lösen können. Die Methode, die im Folgenden näher betrachtet werden soll, ist die Aufteilung der Population in Teilpopulationen, so genannten *Demes* [28]. Dieses Verfahren wurde von den Genetischen Algorithmen übernommen [113] und bedeutet, dass eine Population in gleich große Teilpopulationen aufgeteilt wird, die sich zunächst autark entwickeln. Nach einer vorher festgelegten Anzahl von Fitnessauswertungen (bzw. Generationen) migriert ein ebenfalls als Parameter festgelegter Anteil von Individuen in andere Demes. Es kann hierbei eine Topologie für die Demes festgelegt werden, so dass Individuen nur in benachbarte Teilpopulationen migrieren können. Das Verfahren eignet sich gut für eine Parallelisierung eines GP-Laufs mit mehreren Prozessoren, wobei auf einem Prozessor jeweils die Evolution in einem Deme vorangetrieben wird [114]. Ein Nachteil von Demes liegt darin, dass die Diversität durch die Migration sehr schnell wieder verloren gehen kann: Wenn Migranten eine wesentlich höhere Fitness als alle Individuen einer Zielpopulation haben, können die alten Individuen sehr schnell durch Varianten des neuen Individuums verdrängt werden. Dieses Problem wird in Abschnitt 9.1.1 näher betrachtet. Das in diesem Kapitel vorgestellte Verfahren verwendet ebenfalls Demes, kann diese jedoch bei Bedarf dynamisch erzeugen und auch wieder zusammenlegen.

9.1 Dynamische Demes

In diesem Abschnitt wird ein Verfahren vorgestellt, mit dem eine dynamische Aufteilung der Population in Teilpopulationen (Demes) möglich ist. Ein Deme kann sich in zwei unabhängige Demes aufteilen, und mehrere Demes können unter bestimmten Bedingungen wieder zu einem zusammengefasst werden. Ziel des Verfahrens ist es, dass innerhalb eines GP-Laufs in der Gesamtpopulation eine Diversität vorhanden ist, die einerseits das Verlassen lokaler Optima während der Evolution ermöglicht, andererseits eine zielgerichtete Suche noch möglich macht.[1]

[1] Die höchste Diversität würde erzielt, wenn jedes neue Individuum zufällig und ohne Zusammenhang zu einem Elterindividuum erzeugt würde. In diesem Fall wäre jedoch keine zielgerichtete Suche mehr möglich.

Hierzu müssen folgende Fragen beantwortet werden:

1. Unter welchen Bedingungen wird eine Population aufgetrennt?
2. Unter welchen Bedingungen werden Teilpopulationen wieder zusammengefasst?

Die erste Frage wird im folgenden Abschnitt 9.1.1 näher untersucht. Die Betrachtung von Vereinigungen einzelner Teilpopulationen wird in Abschnitt 9.1.2 beschrieben. Der aus diesen beiden Abschnitten hervorgehende Algorithmus wird abschließend in 9.1.3 anhand der Testprobleme aus Kapitel 5 bewertet.

9.1.1 Aufteilen einer Population

Wenn in einer Population die Fitnessentwicklung stagniert, verringert sich mit fortschreitender Anzahl der mutierten Individuen die Diversität. GOLDBERG und DEB beschreiben in [34] die so genannte *Takeover Time* für generationsbasierte Selektionsmethoden. Sie sagt aus, nach wie vielen Selektionen ohne Fitnessverbesserung die gesamte Population nur noch aus Kopien des besten Individuums besteht. Derselbe Effekt wurde von RUDOLPH für turnierbasierte Selektionsmethoden untersucht [98]. An dieser Stelle soll ein ähnliches, bereits von MCPHEE und HOPPER in [77] beschriebenes Verhalten einer Population untersucht werden: Nach nur wenigen Generationen enthalten alle Individuen nur noch den genetischen Code desselben Individuums der Ausgangspopulation. Es ist davon auszugehen, dass dieses ursprüngliche Individuum (oder seine frühen Nachfolger) über eine relativ gute Fitness verfügte und mit einem, GOLDBERGs *Takeover*-Berechnungen ähnlichem Effekt die Population übernahm.

Dieser Effekt kann ebenfalls beim graphbasierten GP-System *GGP* beobachtet werden und wird im Folgenden ebenfalls *Takeover*-Effekt genannt. Zu jedem Individuum der Population wird die Nummer des Individuums der Ursprungspopulation vermerkt, von dem es abstammt. Ein neu entstandenes Individuum erhält immer die Nummer, mit der das Individuum markiert war, aus dem es erzeugt wurde. Entsteht ein Individuum durch *Crossover* zweier Individuen, wird ebenfalls nur die Nummer des Ursprungindividuums weitergereicht, die dem Besseren der beiden Eltern zugeordnet war.

Die so erfolgte Zuordnung ist aus zwei Gründen ungenau.

1. Das zweite Individuum eines Crossovers gibt ebenfalls genetischen Code weiter.
2. Nach mehreren Mutationen eines Individuums ist es denkbar, dass vom ursprünglichen Individuum kein genetischer Code mehr vorhanden ist und sich das Individuum ausschließlich aus Knoten zusammensetzt, die durch die Mutationen neu entstanden sind.

Diese Zufallssuche ist im allgemeinen schlechter als bekannte GP-Verfahren.

Zur Untersuchung des Effekts wurden mit allen sechs Testproblemen des Kapitels 5 zweimal 200 Läufe durchgeführt. Zum einen wurde das ursprüngliche Verfahren mit kontextsensitiven Operatoren (*Std+*, Abschnitt 7.4) eingesetzt, zum anderen wurden zusätzlich die erweiterten Schrittweiten *Step* (Abschnitt 8.2) verwendet. Die Ergebnisse sind in Abbildung 9.1 zusammengefasst.

	Klassif.	Ant	Mower	Polynom	Sinus	Two-Boxes
Std+	985	1192	984	2208	1395	1703
Step	1499	2176	1460	3932	2559	3086

Abbildung 9.1: Der *Takeover*-Effekt bei GP-Läufen mit und ohne Schrittweitensteuerung

Die Zahlen geben an, nach wie vielen Fitnessauswertungen durchschnittlich alle Individuen der Population vom selben Individuum der Ursprungspopulation abstammen. Selbst wenn diese Zahlen durch die beiden oben beschriebenen Ungenauigkeiten etwas zu niedrig sind, zeigt sich, dass die Diversität einer zufällig erzeugten Ursprungspopulation nach nur wenigen Fitnessauswertungen vollständig verloren gegangen ist. Gleichzeitig ist zu sehen, dass durch die Wahl des Schrittweitenverfahrens *Step* die Beschränkung auf den genetischen Code genau eines Individuums der Ursprungspopulation hinausgezögert wird. Es finden in der Regel 1.5 bis 2 Mal so viele Fitnessauswertungen statt, wie beim *Std+*-Verfahren. Definiert man die Diversität einer Population somit nach McPhee und Hopper über die Abstammung einzelner Individuen, so dient die *Step*-Erweiterung des GP-Grundalgorithmus der Diversitätserhaltung.

Wenn neben der Abstammung aller Individuen von demselben Ursprungsgraphen zusätzlich die Fitness stagniert, deutet dies darauf hin, dass sich die Fitness einem lokalen Optimum angenähert hat. Da sich alle Individuen zu diesem Zeitpunkt relativ ähnlich sind, findet hauptsächlich eine lokale Suche statt, die die Population eher noch näher an das lokale Optimum heranführt, als dessen Umgebung zu verlassen.

Das hier vorzustellende Verfahren der *dynamischen Demes* teilt die Population in dieser Situation in zwei gleich große Demes auf. Die Individuen werden nach ihrer Fitness sortiert und die erste Hälfte wird dem ersten Deme zugeordnet. Die zweite Hälfte der Individuen wird durch neue, zufällig erzeugte ersetzt, so dass der zweite Deme aus einer vollständig neuen Population besteht.

Ab diesem Zeitpunkt werden beide Teilpopulationen als autarke GP-Läufe betrachtet. Die Anzahl der Selektionen pro Deme wird proportional zur Anzahl der Individuen innerhalb des Demes gewählt. In beiden Teilpopulationen wird nun wiederum auf die Situation von erfolgtem *Takeover*-Effekt und stagnierender Fitness gewartet, woraufhin der entsprechende Deme wiederum aufgeteilt wird.

Zur Vermeidung von zu kleinen eigenständigen Teilpopulationen wird dieses Verfahren nur für Demes mit mehr als acht Individuen angewandt. Kleinere Demes werden weiterhin als eigenständige GP-Läufe integriert und entsprechend dem im Abschnitt 9.1.2 vorzustellenden Verfahren wieder mit anderen Demes zusammengefasst, werden aber nicht weiter aufgeteilt.

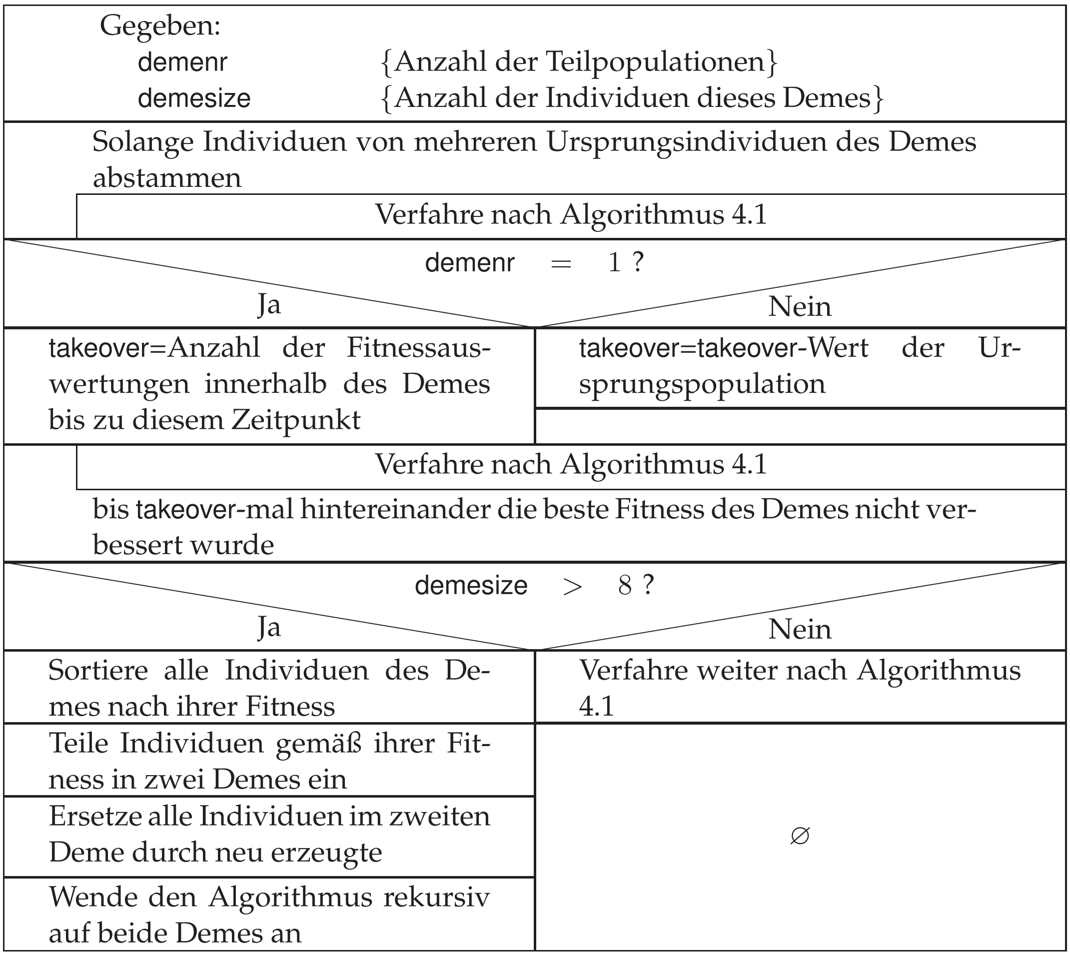


Abbildung 9.2: Algorithmus zur Aufteilung einer Teilpopulation

Der genaue Zeitpunkt der Aufteilung einer Population berechnet sich nach dem Algorithmus aus Abbildung 9.2. Da für kleinere Teilpopulationen die Anzahl der Fitnessauswertungen bis zur Übernahme der Population durch ein Individuum wesentlich geringer ist als bei der vollständigen Anfangspopulation wird die Dauer der Fitnessstagnation vor einer Demespaltung grundsätzlich von der *Takeover*-Zeit der Ursprungspopulation abhängig gemacht. Kleinere Demes treten erst zu einem Zeitpunkt auf, an dem die Geschwindigkeit der Evolution im Allgemeinen bereits stark zurückgegangen ist. In dem Deme, der bei einer Aufspaltung die besseren Individuen der Population erhält, stagniert die Fitness bereits. Durch die Verwendung der ursprünglichen *Takeover*-Zeit wird eine Weiterentwicklung der Fitness möglich, ohne dass sich der Deme sofort wieder halbiert.

9.1.2 Vereinigung mehrerer Teilpopulationen

Der *Takeover*-Effekt stellt auch für statische Deme-Implementierungen ein Problem dar, bei denen die Population von Anfang an in mehrere Teilpopulationen aufgespalten ist und einzelne Individuen zu festgelegten Zeiten zwischen den Demes migrieren. Sobald ein Individuum in einen anderen Deme überführt wird, das eine bessere Fitness hat als alle Individuen des neuen Demes, so wird sofort der *Takeover*-Effekt einsetzen und alle Individuen des neuen Demes durch Mutationen dieses Individuums ersetzen. Somit wird der ganze Zuwachs an Diversität, der durch die Aufteilung in Teilpopulationen erreicht wird, durch die Migrationen zunichte gemacht.

Diese statische Form der Demes hat in Bezug auf die Populationsdiversität gegenüber panmiktischen Populationen immer noch Vorteile, da oft auch Individuen migrieren, die keinen vernichtenden Einfluss auf den neuen Deme haben. Durch eine moderate Wahl der Migrationsgeschwindigkeit kann der Effekt weiterhin für jedes zu lösende Problem individuell angepasst werden.

Der hier vorgestellte Ansatz für die Verwendung von Teilpopulationen kommt vollständig ohne Migration aus. Statt dessen können während eines GP-Laufs Teilpopulationen wieder zu einer zusammengefasst werden. Es muss darauf geachtet werden, dass keine der beiden zusammen zu führenden Teilpopulationen sofort sämtliche Individuen der anderen durch Mutationen eigener Individuen überschreibt. Dies wäre dann der Fall, wenn die besten Individuen der einen Teilpopulation deutlich bessere Fitnesswerte als die der anderen hätten.

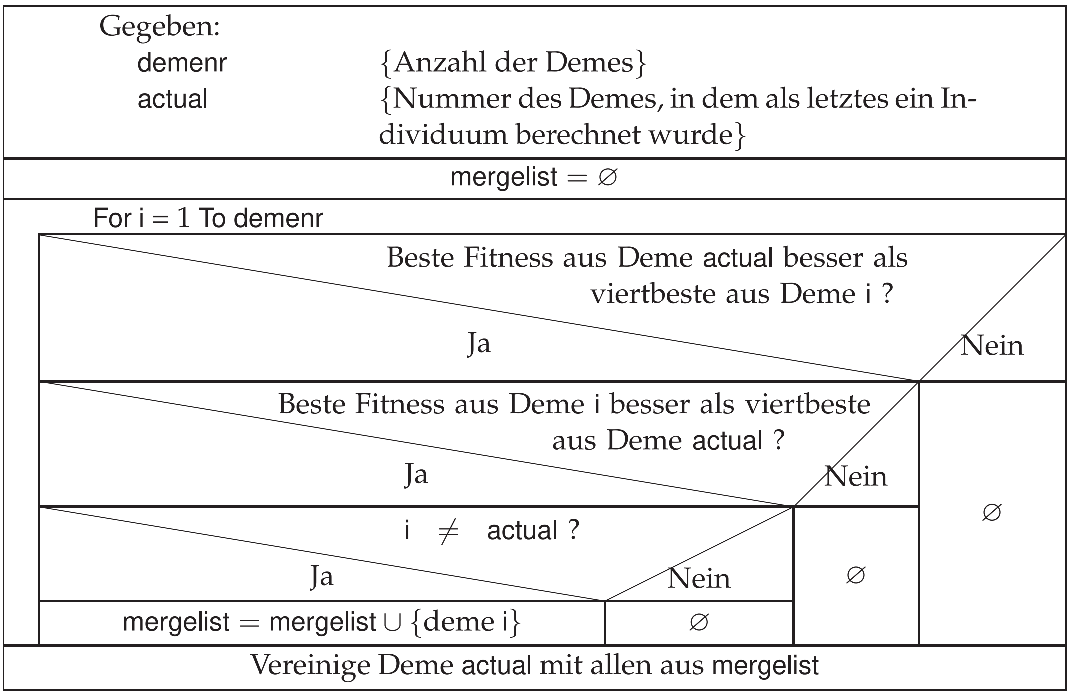


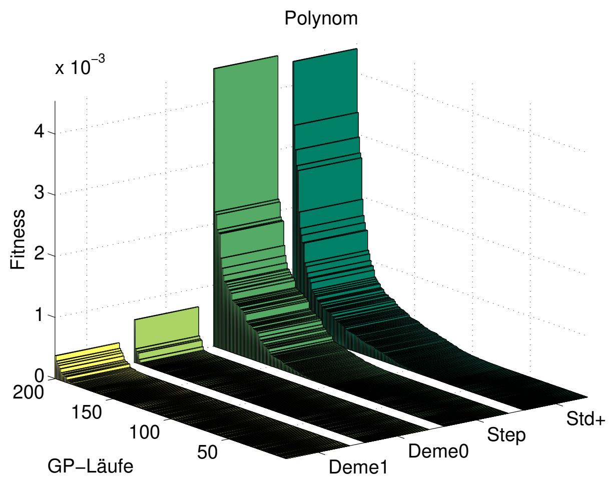
Abbildung 9.3: Algorithmus zur Vereinigung zweier Teilpopulationen

Der in *GGP* verwendete Algorithmus ist in Abbildung 9.3 dargestellt. Er wird immer dann ausgeführt, wenn mehrere Demes vorhanden sind und zu einem mutierten Individuum ein neuer Fitnesswert berechnet wurde. Zwei Demes werden genau dann vereinigt, wenn der jeweils beste Fitnesswert eines Demes zu den besten vier Werten des anderen Demes gehören würde. Auf diese Weise wird sichergestellt, dass die besten Fitnesswerte in beiden Teilpopulationen ungefähr gleich sind und der *Takeover*-Effekt etwas länger dauert, so dass noch möglichst lange die Möglichkeit besteht, mittels *Crossover* neue Individuen zu erzeugen, die Teile des genetischen Codes aus beiden Teilpopulationen in sich vereinen.

9.1.3 Ergebnisse

Das GP-System *GGP* wurde um die Verfahren zur dynamischen Demeverwaltung aus den vorigen Abschnitten erweitert und mit Hilfe der sechs Testprobleme aus Kapitel 5 näher untersucht.

Zu jedem der Probleme wurden zweimal 200 Läufe durchgeführt. Bei beiden Varianten wurde die Parametrierung der Läufe entsprechend Kapitel 5 vorgenommen. Zusätzlich wurden bei allen Testreihen neben der dynamischen Demeverwaltung kontextsensitive genetische Operatoren verwendet (Abschnitt 7.4). Die Testreihen wurden zum einen ohne Schrittweitensteuerung (im Folgenden als *Deme*0 bezeichnet), zum anderen mit Schrittweitensteuerung (*Deme1*) durchgeführt. Die Ergebnisse wurden mit den entsprechenden Testreihen ohne Verwendung von Demes verglichen (*Std+*, Abschnitt 7.4.2 und *Step*, Abschnitt 8.2) und sind in den Abbildungen 9.4 bis 9.9 dargestellt. Außer beim *Two-Boxes*-Problem ergibt sich bei allen Testproblemen eine signifikante Verbesserung – sowohl gegenüber *Std+* als auch *Step*.

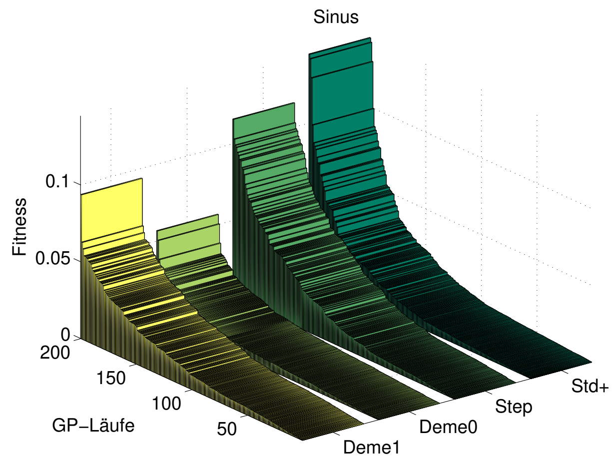


	min	max	mean
Std+	0	0.00439	0.00027
Step	0	0.00454	0.00024
Deme0	0	0.00070	0.00001
Deme1	0	0.00038	0.00001

Abbildung 9.4: Die Ergebnisse der Testläufe zum *Polynom*-Problem

Beim *Polynom*-Problem sind beide *Deme*-Verfahren signifikant besser als *Std+* und *Step*. Die

Grafik aus Abbildung 9.4 zeigt, dass sich die Fitnesswerte für *Std+* und *Step* jeweils in zwei Gruppen einteilen lassen: Die Werte kleiner 10^{-6}, die in der Grafik nicht von der X-Achse zu unterscheiden sind und die übrigen. Die beiden Verfahren *Deme0* ohne und *Deme1* mit Schrittweitensteuerung reduzieren die zweite Gruppe auf jeweils weniger als 30 Werte. Die hohen Werte der ursprünglichen Verfahren beruhen darauf, dass die entsprechenden Läufe lokale Fitnessoptima erreicht hatten und diese nicht mehr verlassen konnten. Durch die dynamische Demeverwaltung konnten entsprechende Populationen aufgeteilt werden und in den zusätzlichen Demes haben sich Individuen gebildet, die entweder direkt besser als die Individuen des ungeteilten Demes waren oder mit deren Hilfe nach einer Zusammenlegung der Demes bessere Fitnesswerte jenseits des lokalen Optimums gefunden werden konnten.



	min	max	mean
Std+	0.00001	0.14506	0.01773
Step	0.00016	0.11612	0.02305
Deme0	0.00014	0.05683	0.00848
Deme1	0.00030	0.09379	0.01643

Abbildung 9.5: Die Ergebnisse der Testläufe zum *Sinus*-Problem

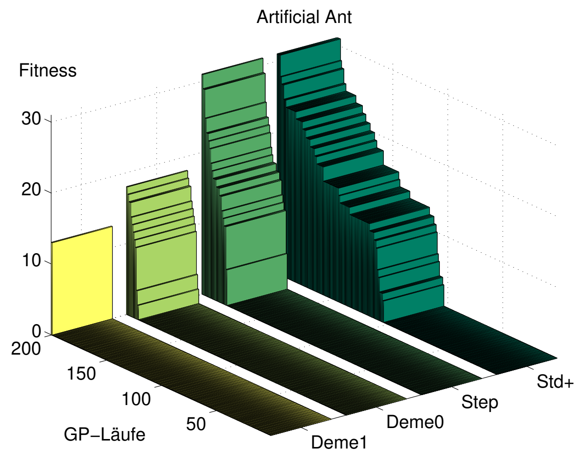
Beim *Sinus*-Problem war das Verfahren ohne Schrittweitenvariation *Std+* besser als *Step*. An dieser Eigenschaft hat sich durch die Verwendung dynamischer Demes nichts geändert. Allerdings konnte *Deme0* signifikant bessere Ergebnisse erzielen als die drei übrigen Verfahren. Die Zahlen der Tabelle zeigen ebenfalls, dass in den Testläufen *Deme1* besser zu sein scheint als das entsprechende Verfahren ohne Demes (*Step*). Ein Wilcoxon-Rangsummentest konnte diese Aussage allerdings nicht verallgemeinern.

Beim *Lawnmower*-Problem hat die Verwendung von Demes zu keinen Veränderungen bei den Ergebnissen der GP-Läufe geführt. Wie die Tabelle aus Abbildung 9.6 zeigt, sind die Ergebnisse für die beiden Verfahren ohne bzw. für die Verfahren mit verschiedenen Schrittweiten nahezu identisch geblieben. Dieses Resultat war zu erwarten, da *Lawnmower* ein für GP-Verhältnisse extrem einfaches Problem ist. Wie die *improv*-Spalte der Tabelle zeigt, werden perfekte Individuen bereits nach durchschnittlich etwa 1370 bzw. 1640 Fitnessauswertungen gefunden. Da für dieses Problem nach der Abbildung 9.1 der *Takeover*-Effekt erst nach durchschnittlich 984/1460 Fitnessauswertungen einsetzt und vor einer Demeteilung die Fitness noch einmal ebenso lange stagnieren müsste, werden Individuen mit optimaler

Fitness bereits vor der ersten Demeteilung gefunden.

	succ.	impr.	effort (evals)
Std+	200	1645	4686 (3568)
Step	200	1373	3311 (2314)
Deme0	200	1637	4686 (3568)
Deme1	200	1374	3311 (2314)

Abbildung 9.6: Die Ergebnisse der Testläufe zum *Lawnmower*-Problem



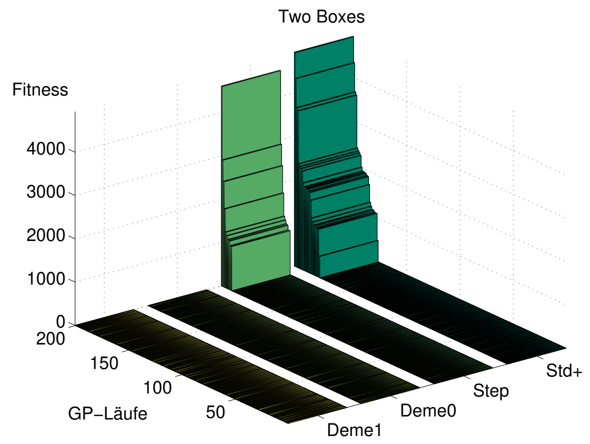
	succ.	impr.	effort (evals)
Std+	103	49282	378071 (36000)
Step	177	43955	167804 (10000)
deme0	189	60624	274082 (4000)
deme1	199	29498	102415 (87000)

Abbildung 9.7: Die Ergebnisse der Testläufe zum *Artificial Ant*-Problem

Eine weitere Verbesserung der Ergebnisse gelang mittels Demes beim *Artificial Ant*-Problem. Das Verfahren mit einer Kombination aus Demes und Schrittweiten konnte in 199 von 200 Fällen eine Lösung finden. Dieses Ergebnis ist deutlich besser als aktuell veröffentlichte Ergebnisse anderer GP-Systeme (z.B. 16 Lösungen bei 500 Läufen in [71] bei LUKE und PANAIT). LANGDONs Aussage, dass GP bei diesem Problem nur mit einer Zufallssuche vergleichbare Ergebnisse erzielt, ist zumindest für das hier vorgestellte graphbasierte GP-System *GGP* ebenfalls nicht mehr gültig. Wie auch schon im Vergleich zwischen *Std+* und *Step* ist das Verfahren mit verschiedenen Schrittweiten auch beim Einsatz von Demes signifikant besser als dasselbe Verfahren ohne Schrittweiten.

Bei dem *Two-Boxes*-Problem ergibt sich ein ähnliches Bild. Beide Deme-Verfahren sind signifikant besser als die Vergleichsverfahren und können beide in allen Testläufen eine Lösung finden. Untereinander sind die beiden Verfahren *Deme0* und *Deme1* mit einem Wilcoxon-Rangsummentest nicht signifikant von einander zu unterscheiden.

An diesem Testproblem zeigen sich ebenfalls wieder interessante Eigenschaften der *Effort*-Wert Berechnung. Obwohl die Verfahren mit und ohne Demes sich den Ergebnissen nach signifikant unterscheiden (dies gilt sowohl für das Paar mit als auch ohne Schrittweiten), haben die jeweils zusammen gehörenden Paare fast dieselben *Effort*-Werte.



	succ.	impr.	effort (evals)
Std+	174	23992	36886 (2687)
Step	190	18489	41636 (5740)
Deme0	200	13661	36839 (3446)
Deme1	200	13091	42702 (6335)

Abbildung 9.8: Die Ergebnisse der Testläufe zum *Two-Boxes*-Problem

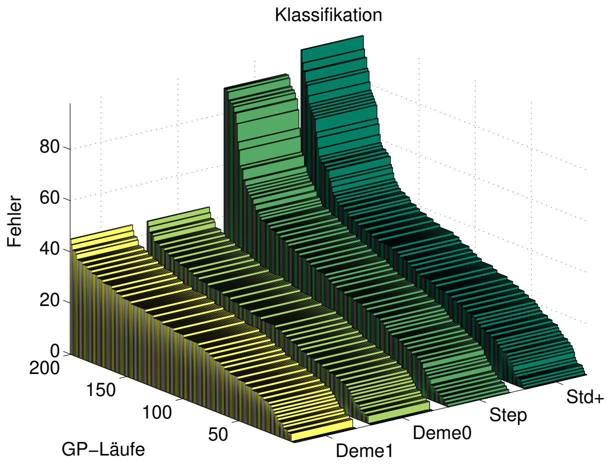
Beim Klassifikationsproblem waren ohne den Einsatz von Demes die Ergebnisse der beiden Vergleichsverfahren relativ ähnlich, die Schrittweiten konnten die Fitnessentwicklung nicht positiv beeinflussen. Dieser Zustand hat sich durch die Hinzunahme von Demes nicht verändert. Die Ergebnisse von *Deme0* und *Deme1* sind sich nach wie vor sehr ähnlich. Allerdings konnte der Einsatz der Demes die Ergebnisse beider Verfahren wiederum signifikant verbessern, die Zahl der maximal falsch klassifizierten Testdaten konnte sogar halbiert werden.

9.2 Zusammenfassung

In diesem Kapitel wurden zunächst verschiedene, aus der Literatur bekannte Ansätze vorgestellt, wie bei der Genetischen Programmierung die Diversität in einer Population gemessen werden kann und welche Verfahren zur Diversitätserhaltung eingesetzt werden. Der Einsatz von so genannten *Demes* wurde an dieser Stelle gesondert betrachtet.

Anschließend wurde ein Verfahren vorgestellt, das Fitnessstagnation und die Abstammung sämtlicher Individuen der Population für Diversitätsbetrachtungen heranzieht. Im Fall von Stagnation wird die Population in mehrere Demes aufgeteilt, wobei neue zufällige Individuen zur Diversitätserhöhung erzeugt werden. Demes, die sich unabhängig voneinander auf ein vergleichbares Fitnessniveau entwickelt haben, werden wieder zusammenführt.

Anhand der Testprobleme aus Kapitel 5 wurde gezeigt, wie das neue Verfahren zur dynamischen Demeverwaltung die Ergebnisse zum Teil deutlich verbessern konnte. Das Verfahren kann hierbei zusätzlich zum im Abschnitt 8.2 vorgestellten Verfahren zur Verwendung unterschiedlicher Schrittweiten *Step* eingesetzt werden. Die Verbesserungen, die durch beide



	min	max	mean
Std+	0	98	32.5
Step	0	90	32.4
Deme0	0	45	23.4
Deme1	2	45	24.8

Abbildung 9.9: Die Ergebnisse der Testläufe zum Klassifikationsproblem, die Tabelle gibt bereits nicht die Fitnesswerte sondern die Anzahl der Fehlklassifikationen an.

Erweiterungen erreicht werden, sind unabhängig voneinander. Beide Verfahren können zusammen (hier *Deme1* genannt) eingesetzt werden. Das Verfahren eignet sich somit dazu, die fest vorgegebene Anzahl von Fitnessberechnungen effizienter zum Finden eines Optimums zu verwenden.

Kapitel 10

Isomorphe Graphen

Bei vielen Anwendungen der Genetischen Programmierung sind einzelne Fitnessauswertungen noch sehr zeitaufwändig. Dies kann zum Beispiel am numerischen Lösen von Differentialgleichungen oder ähnlich rechenintensiven Aufgaben liegen. Beispiele hierfür sind die Evolution von Schachprogrammen [35], von Laufrobotern [19] oder – allgemeiner – die Modellierung und Regelung dynamischer Systeme, wie sie in Anhang A vorgestellt wird.

Für genannte Anwendungsfälle ist es sinnvoll, die Anzahl der benötigten Fitnessauswertungen möglichst gering zu halten. Ein erster Ansatz hierfür ist, den durch eine Reproduktion erzeugten Individuen die Fitness des identischen Elter-Individuums zuzuordnen. Bei diesem und bei allen weiteren Ansätzen des Kapitels ist zu beachten, dass sie nur verwendet werden können, wenn identischen Individuen immer dieselbe Fitness zugeordnet werden kann. Wird zum Beispiel immer nur eine Teilmenge der zur Verfügung stehenden Testfälle zur Fitnessberechnung herangezogen, ist die Fitness in den meisten Fällen für jede Teilmenge unterschiedlich und muss jedes Mal neu berechnet werden. Ein entsprechender Fall tritt zum Beispiel bei der in [33] vorgestellten *Dynamic Subset Selection* oder ähnlichen Verfahren ein. Eine andere Möglichkeit für unterschiedliche Fitnesswerte bei identischen Individuen sind sich dynamisch ändernde Fitnesslandschaften, bei denen sich ein Optimum im Laufe eines GP-Laufs verschiebt und es somit Aufgabe des Algorithmus ist, erst sich dem Optimum anzunähern und diesem dann zu folgen.

Das in dieser Arbeit vorgestellte GP-System *GGP* weist von vornherein Individuen, die durch Replikation entstehen, die Fitness des entsprechenden Elter-Individuums zu, ohne diese neu zu berechnen. Neben dieser Form von identischen Individuen kann es zusätzlich vorkommen, dass ein Individuum erzeugt wird, das sich zwar von seinen direkten Vorgängern unterscheidet, jedoch trotzdem schon einmal während des GP-Laufs aufgetreten ist.

Des Weiteren kann es passieren, dass zwei Graphen zwar gleich sind, dies aber aufgrund der verwendeten Datenstruktur innerhalb des GP-Systems nicht erkannt wird. So sind durch die Implementierung des Datentyps *Graph* alle Knoten eines Individuums durchnummeriert. Wenn bei zwei Individuen die Reihenfolge der Knoten unterschiedlich ist, die Strukturen aber durch ebenfalls verschobene Kantennummern denselben Graphen repräsentieren, können sie nicht durch einen direkten Vergleich der Datenstruktur als identisch erkannt werden. Knoten, die kommutative Funktionen repräsentieren, sind eine weitere Ursache dafür, dass in der Datenstruktur unterschiedliche Individuen denselben GP-Graphen darstellen können. Alle diese Formen von Graph-Isomorphien werden in diesem Kapitel näher betrachtet und anhand der hierfür geeigneten Testprobleme aus Kapitel 5 auf ihre Relevanz untersucht. Ähnliche Untersuchungen wurden bereits von IGEL für *Neuronale Netze* durchgeführt [48].

10.1 Die Implementierung

Sollen während eines GP-Laufs identische GP-Graphen wieder erkannt werden, müssen diese in einer geeigneten Datenstruktur abgelegt und während des GP-Laufs mit weiteren Graphstrukturen verglichen werden können. Für die Speicherung sämtlicher GP-Graphen während eines GP-Laufs kann sich so ein sehr hoher Speicherbedarf einstellen. In den Testproblemen aus Kapitel 5 werden jeweils 200000 Fitnessauswertungen durchgeführt – die in den Beispielen verwendete maximale Graphgröße beträgt 80 Knoten. Zusätzlich müssen zu jedem einzelnen Knoten noch weitere Informationen abgespeichert werden, zum Beispiel eventuell vorhandene Parameter und der Funktionstyp, der durch den Knoten repräsentiert wird. Des Weiteren muss jeder neu entstandene Graph mit den bereits vorhandenen verglichen werden.

Um sowohl den Speicherbedarf als auch die Rechenzeit im Rahmen zu halten, wurde folgende Implementierung gewählt:

- Die Graphen werden in einem binären Suchbaum [38] abgelegt.
- Zu jedem Graphen wird die Adjazenzmatrix und die Zuordnung der Knoten zu der entsprechenden Grundfunktion (Definitionen 3.2 und 3.4) gespeichert.
- Die Adjazenzmatrix wird gemeinsam mit den Knotenzuordnungen mittels *zlib*-Kompression [24, 25, 26] komprimiert.
- Die komprimierten Graphen werden zusammen mit der komprimierten Größe und der Anzahl der zum Graphen gehörenden Knoten gespeichert. Diese drei Elemente dienen als Knotenmarkierung des Binärbaums.
- Wird ein neues Element mit einem des Baumes verglichen, wird zunächst mit der komprimierten Größe, dann mit Anzahl der Knoten und zuletzt mit den komprimierten Daten verglichen.

10.1.1 Einschränkungen

Die vorgestellten Punkte sorgen zwar zum einen dafür, dass der Ressourcenbedarf des GP-Systems für heutige Rechner zu bewältigen ist, zum anderen verursachen sie mehrere Einschränkungen für die zu bearbeitenden Probleme:

1. Bei der gewählten Datenstruktur werden keine parametrierten Knoten berücksichtigt.
2. In der oben beschriebenen Variante können keine nicht-kommutativen Grundfunktionen verwendet werden.

Während die zweite Einschränkung, wie später gezeigt wird, für die verwendeten Testprobleme aufgehoben werden kann, bleibt die Erste bestehen: Wenn in einem Graph Knoten vorkommen, die über eigene Parameter verfügen, können diese nicht in den Knotenmarkierungen des Binärbaums berücksichtigt werden. Ein Knoten, der eine Konstante mit dem Wert 4.6 repräsentiert, kann somit nicht von einem mit dem Wert -2 unterschieden werden. Hierdurch reduzieren sich die geeigneten Testprobleme aus Kapitel 5 auf das *Artificial Ant-*, das *Lawnmower-* und das *Two-Boxes*-Problem.[1]

Findet allerdings eine hybride Form der Optimierung statt, bei der die Struktur einzelner Individuen mit der Genetischen Programmierung und die Parameter einzelner Knoten über andere Verfahren, wie zum Beispiel *Evolutionsstrategien* [93], *Simulated Annealing* oder *Particle Swarm Optimization* [53] ermittelt werden, kann eine Untersuchung auf Isomorphien eingesetzt werden, um bestimmte Strukturen als bereits untersucht zu kennzeichnen und sie im weiteren Verlauf der Optimierung nicht mehr zu berücksichtigen.

Zur Komprimierung des GP-Graphen wird eine Adjazenzmatrix für einen einfachen gerichteten Graphen verwendet. Da es sich bei GP-Graphen um Multigraphen handelt, bei denen nicht nur die Anzahl der Kanten zwischen zwei Knoten größer eins sein kann, sondern die Kanten eines Knotens auch noch eine bestimmte Reihenfolge haben, ist die Abbildung von den GP-Graphen in die Menge der Adjazenzmatrizen[2] nicht injektiv. Die Verwendung von Adjazenzmatrizen lässt sich jedoch nicht umgehen, da das zur Isomorphieuntersuchung verwendete externe Programmpaket *nauty* (Abschnitt 10.2, [75]) nur diese unterstützt.

Um eine injektive Abbildung zwischen den GP-Graphen der Testprobleme und den Adjazenzmatrizen herzustellen, müssen zwei Punkte beachtet werden:

- Was passiert, wenn zwischen zwei Knoten mehrere Kanten parallel verlaufen?
- Was muss bezüglich der Reihenfolge der Kanten beachtet werden?

Bei allen verwendeten Testproblemen sind maximaler Eingangs- und Ausgangsgrad aller Knoten kleiner oder gleich zwei. Wenn in einer Zeile der Adjazenzmatrix, die für einen Knoten mit Ausgangsgrad zwei steht, nur eine Verbindung angegeben ist, bedeutet dies, dass beide Kanten zum selben Knoten führen. Analoges gilt für die Spalten der Matrix. Probleme würden erst auftreten, wenn mehr als zwei Kanten aus einem Knoten herausführen. In solch einem Fall müsste der Graph um Zusatzknoten erweitert werden. Für jeden möglichen Eingang und Ausgang wird ein neuer (1,1)-Knotentyp *Eingang1*, *Eingang2*, *Ausgang1* usw. erzeugt. Die diesen Zusatzknoten zugeordnete Funktion ist jeweils die Identitätsfunktion. Eine Kante vom I-ten Ausgang von Knoten m zum J-ten Eingang des Knotens n wird

[1] Bei den beiden letztgenannten mussten Knotentypen weggelassen werden, siehe hierfür Abschnitt 10.3.
[2] Erweitert um die Zuordnungen der Knoten zu den Grundfunktionen.

durch einen Teilgraphen (V, E) mit $V = \{a, e\}$ und $E = \{(m, a), (a, e), (e, n)\}$ ersetzt, wobei dem Knoten a die Funktion *AusgangI* und dem Knoten e die Funktion *EingangJ* zugeordnet wird.

Eine Implementierung dieses Ansatzes würde das Abspeichern entsprechend größerer Adjazenzmatrizen nach sich ziehen. Jeder (i,j)-Knoten würde um $i + j$ Zusatzknoten erweitert, so dass für den Knoten alleine $1 + i + j$ Zeilen/Spalten in der Adjazenzmatrix anfallen würden.

Aus den oben genannten Gründen konnte für die verwendeten Testprobleme auf eine Implementierung der Zusatzknoten verzichtet werden. Die Reihenfolge der Kanten darf jedoch nur vernachlässigt werden, wenn die einem Knoten zugeordnete Grundfunktion in den entsprechenden Variablen kommutativ ist. Stellt ein GP-Graph mit drei Knoten zum Beispiel den Ausdruck $3 - 1$ dar, so ist seine Adjazenzmatrix identisch mit der eines GP-Graphen $1 - 3$. Diese Beschränkung auf kommutative Funktionen wird jedoch von den verwendeten Testproblemen nicht erfüllt.

Durch diese Einschränkung ist eine rudimentäre Implementierung von Zusatzknoten unumgänglich: Wenn ein Knoten über zwei nicht-kommutative Ausgangs-/Eingangskanten verfügt, wird bei der ersten ein Zusatzknoten vom neuen Typ *Knotenausgang/-eingang* eingefügt. Die Grundfunktionen, die durch diese beiden neuen Typen repräsentiert werden, sind wiederum die Identität. Somit wird die Semantik des GP-Graphen nicht verändert, gleiche GP-Graphen werden aber auf dieselbe Adjazenzmatrix abgebildet. Bei kommutativen Eingängen/Ausgängen kann auf die Zusatzknoten verzichtet werden, da die GP-Graphen zwar unterschiedlich, durch die Kommutativität aber semantisch identisch sind. Die Abbildung auf dieselbe Adjazenzmatrix ist in diesem Fall sogar wünschenswert.

10.2 Isomorphien

Mit der in Abschnitt 10.1 beschriebenen Implementierung ist es möglich, Individuen als identisch zu identifizieren, die dieselbe Darstellung im Speicher haben. Sobald jedoch zwei Knoten in der internen Darstellung (trotz gleicher GP-Graphen) vertauscht sind, ist es nicht mehr möglich, diese Individuen als gleich zu identifizieren. Abbildung 10.1 gibt ein Beispiel hierfür.

In Definition 10.1 wird der aus der Literatur bekannte Isomorphiebegriff (z.B. [29]) für GP-Graphen mit zugehöriger Semantik erweitert.

Definition 10.1
Zwei GP-Graphen $G_1 = (I_1, O_1, V_1, E_1)$ *und* $G_2 = (I_2, O_2, V_2, E_2)$ *mit den Semantiken* $G_{1,func}/G_{1,par}$ *sowie* $G_{2,func}/G_{2,par}$ *heißen isomorph, wenn folgende Bedingungen erfüllt sind:*

1. *Es gibt drei bijektive Abbildungen* $f_I : I_1 \to I_2$, $f_O : O_1 \to O_2$ *und* $f_V : V_1 \to V_2$
2. *Die Ordnungen auf den Mengen* I_1 *und* O_1 *bleiben durch Anwendung der Abbildungen* f_I *und* f_O *auf* I_2 *und* O_2 *erhalten.*

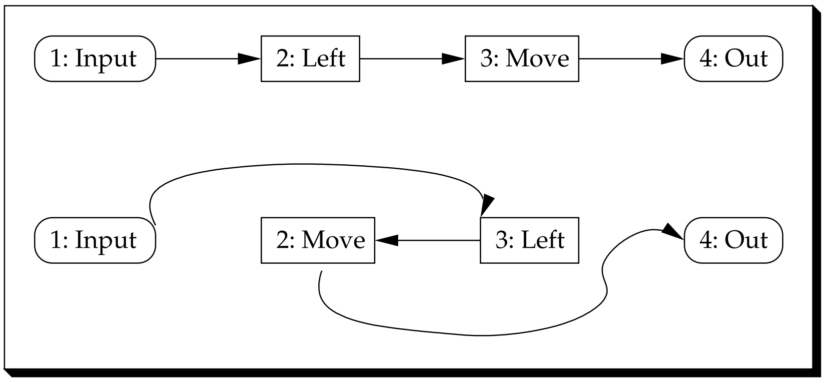


Abbildung 10.1: Zwei isomorphe Graphen zum *Artificial Ant*-Problem

3. *Für alle Elemente v aus V_1 gilt $G_{1,func}(v) = G_{2,func}(f_V(v))$ und $G_{1,par}(v) = G_{2,par}(f_V(v))$*

4. *Zwischen zwei beliebigen Knoten $u, v \in I_1 \cup O_1 \cup V_1$ existiert dieselbe Zahl von Kanten in E_1 wie zwischen den beiden, durch die bijektiven Abbildungen aus Punkt (1) gegebenen Knoten aus $I_2 \cup O_2 \cup V_2$ in E_2.*

5. *Die Reihenfolge der Eingangskanten eines Knotens $v \in V_1$ muss bei nichtkommutativer Funktion $G_{1,func}(v)$ unter der Bijektion f_V erhalten bleiben. Dasselbe gilt für nicht austauschbare Ausgangskanten.*

Mit dem von MCKAY entwickelten Programmpaket *nauty* ist es möglich, Graphen als isomorph zu identifizieren [75]. Das Programm arbeitet mit den Adjazenzmatrizen einfacher, gerichteter Graphen sowie einer Färbung der Knoten. Jedem Knotentyp der GP-Graphen wird hierbei eine Farbe zugeordnet. *Nauty* unterteilt die Menge aller Graphen bezüglich Adjazenzmatrix und unterschiedlicher Färbung in Isomorphieklassen. In jeder dieser Klassen gibt es ein ausgezeichnetes Element, das als kanonischer Graph bezeichnet wird. Durch Anwendung des Programms auf einen Graphen mit Färbung wird der zugehörige kanonische Graph ausgegeben. Die genaue Funktionsweise von *nauty* kann der zugehörigen Literatur entnommen werden [75, 73, 74, 79]. Die entsprechenden Routinen von *nauty* wurden so in das GP-Programm *GGP* integriert, dass anstelle der Adjazenzmatrix plus Knotentypen nun die Adjazenzmatrix des kanonischen Graphen mitsamt der Färbung im binären Suchbaum gespeichert wird. Auf diese Weise ist es möglich, GP-Graphen, zu denen während eines Laufs bereits isomorphe Individuen mit entsprechend identischer Fitness berechnet wurden, ohne Neuberechnung die ursprüngliche Fitness zuzuweisen.

Bei der Implementierung mussten zusätzlich folgende Punkte beachtet werden:

- Bei der Umwandlung der einzelnen Knotentypen in Farben muss jedem Eingangs- und Ausgangsknoten des GP-Graphen eine eigene Farbe zugeordnet werden. Ohne diese Unterscheidung würde eine Kommutativität der Ein-/Ausgänge unterstellt, die nicht vorausgesetzt werden kann. Würden bei einem GP-Graph mit zwei Ausgängen und funktionaler Semantik zum Beispiel der erste Ausgang eine Kraft und der zweite einen Winkel repräsentieren, könnte ein weiterer Graph als isomorph angesehen werden, bei dem beide Ausgänge vertauscht wären.

- Gleiche Farben im kanonischen Graphen müssen immer denselben Knotentypen des GP-Graphen repräsentieren. Aus diesem Grund müssen in den GP-Graphen, die *nauty* übergeben werden, immer alle Farben/Knotentypen in derselben Reihenfolge vorkommen. Wäre dies nicht der Fall, könnten zwei Graphen als isomorph angesehen werden, bei denen einer an sämtlichen Stellen Multiplikationen repräsentierende Knoten hat, wo sich bei dem anderen Divisionsknoten befinden. Sind bei einer Problemstellung *n* verschiedene Knotentypen erlaubt, werden dem eigentlichen Graphen genauso viele (0,0)-Knoten hinzugefügt, die jeweils eine Farbe/einen Knotentyp repräsentieren. Da die Reihenfolge dieser Knoten immer dieselbe ist, verwendet *nauty* immer dieselbe Farbe für einen Knotentyp.

10.3 Ergebnisse

Zur Überprüfung der Häufigkeit identischer und isomorpher Individuen bei GP-Läufen wurden die den Voraussetzungen entsprechenden Testprobleme aus Kapitel 5 verwendet: Das *Artificial Ant*-Problem konnte ohne Änderungen übernommen werden, bei *Lawnmower* wurde auf die Grundfunktion *Frog*, bei *Two-Boxes* auf *Inpv* verzichtet, da diese Funktionen über eigene Parameter verfügen.

Für alle drei Testprobleme wurden je vier Serien mit jeweils 200 Testläufen durchgeführt, die Parametrierung entsprach wiederum der aus Kapitel 5. Zu jedem Testproblem wurden die Verfahren *Std+* aus Abschnitt 7.4 und *Step* aus Abschnitt 8.2 je zweimal angewandt. Einmal wurden in den binären Suchbäumen direkt die GP-Graphen mit Knotentyp und beim zweiten Mal die kanonischen Graphen mit Knotenfärbung gespeichert. In allen Serien wurde gezählt, wie oft neue Individuen (bzw. deren kanonische Repräsentanten) bereits im Suchbaum abgespeichert waren – wie viele Fitnessauswertungen also eingespart werden können.

Es werden jeweils die Durchschnittswerte der Individuenzahlen über alle 200 Läufe der Testserie verglichen. Wie aus den summierten Operatorhäufigkeiten der Tabellen 5.3, 5.7 und 5.8 aus Kapitel 5 hervorgeht, liegt die Replikationsrate für *Std+* beim *Two-Boxes*-Problem bei zwei und bei den beiden anderen bei je vier Prozent.[1] Wenn keine Graphen zwischengespeichert werden, entspricht dies der Anzahl der Individuen, deren Fitness nicht berechnet werden muss. Der Unterschied zum Wert der identischen Individuen ist die Anzahl der Fitnessauswertungen, die durch das Abspeichern sämtlicher Individuen eingespart werden kann. Die Differenz zur Zahl der doppelten, kanonischen Graphen wiederum kann zusätzlich durch die Auswertung der Isomorphien eingespart werden.

[1] Bei *Step* liegt die tatsächliche Replikationsrate bei 0.68 bzw. 1.4 Prozent, da die Schrittweiten zwischen eins und drei variieren und in allen Schritten jeweils eine Replikation stattfinden muss.

Bei jedem GP-Lauf werden zwei Zeitpunkte genauer betrachtet. Zum einen ist dies der Punkt, an dem die globale Fitness des Laufs zum letzten Mal verbessert wurde, zum anderen wird der Wert nach der Berechnung aller Fitnessauswertungen angegeben. Beim *Artificial Ant*-Problem werden zusätzlich die Werte aufgeführt, bei denen zum ersten Mal ein Individuum mit einer Fitness kleiner eins gefunden wurde, bei dem die Ameise also den kompletten *Santa Fé*-Pfad abläuft.

Tabelle 10.1 enthält die Ergebnisse beim *Two-Boxes*-Problem. Der obere Teil der Tabelle zeigt die Ergebnisse ohne Schrittweitensteuerung (Variante *Std+*), der untere fasst die Resultate der Läufe mit unterschiedlichen Schrittweiten (*Step*) zusammen. In der Spalte mit der Überschrift *200000* steht jeweils die Zahl der Fitnessauswertungen, die nach den kompletten GP-Läufen durchschnittlich redundant berechnet wurden. Die Überschrift der letzten Spalte gibt an, nach wie vielen Fitnessauswertungen durchschnittlich das beste Individuum der Läufe gefunden wurde. Die Zeile *identisch* enthält die Anzahl der Individuen, die bereits im binären Suchbaum enthalten waren. Die Zeile *isomorph* enthält die Zahl der Individuen deren kanonischer Graph bereits im binären Suchbaum vorhanden war, zu denen also bereits ein isomorpher GP-Graph gefunden wurde.

Std+	200000		35133.4 (avg.)	
identisch	90805.7	(45.4%)	14095.8	(40.1%)
isomorph	143641.0	(71.8%)	15839.9	(45.1%)
Step	200000		23436.5 (avg.)	
identisch	39514.7	(19.7%)	2780.9	(11.9%)
isomorph	74108.2	(37.1%)	3162.8	(13.5%)

Tabelle 10.1: Redundante Fitnessberechnungen beim *Two-Boxes*-Problem

Beim Betrachten der beiden Tabellen fällt zunächst auf, dass die Werte zum *Std+*-Verfahren mehr als doppelt so hoch sind, wie die zum *Step*-Verfahren. Dies ist ein weiteres Zeichen für die durch *Step* hervorgerufene Diversitätserhaltung. Die Anzahl der möglichen Individuen, die durch zwei oder drei hintereinander ausgeführte Variationen eines GP-Graphen entstehen können, ist höher als die Zahl der Individuen, die durch eine einzige Variation entstehen können. Selbst wenn die Fitness zu schlecht für eine Weiterverbreitung des so entstandenen genetischen Codes ist, ist die Chance mit mehreren Variationen ein bereits bekanntes Individuum zu erzeugen entsprechend kleiner.

Da es sich bei den Zeilen *identisch* und *isomorph* jeweils um dieselben GP-Läufe handelt, ist klar, dass die Redundanzzahlen bei Berücksichtigung der Isomorphien über denen der identischen Individuen liegen. Es ist jedoch überraschend, dass während eines GP-Laufs im Falle von *Std+* bei Erreichen des Optimums bereits über 40 Prozent der Fitnessberechnungen redundant sind. Bei entsprechend langer Laufzeit steigt der Anteil der redundanten Auswertungen weiter an. Abbildung 10.2 zeigt die Anzahl der redundanten Auswertungen während der Testserien zum *Two-Boxes*-Problem sowie deren prozentualer Anteil an der Gesamtzahl der Fitnessauswertungen.

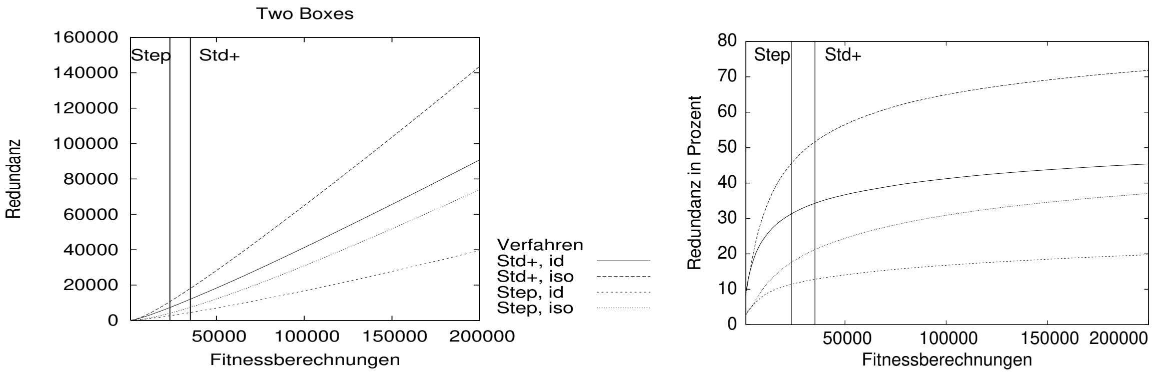


Abbildung 10.2: Redundante Fitnessberechnungen beim *Two-Boxes*-Problem

Die Kurven sind jeweils über die Ergebnisse der 200 Testläufe gemittelt. Die beiden senkrechten Linien symbolisieren die Zeitpunkte, zu denen im Durchschnitt die letzte Fitnessverbesserung eingetreten ist. Die Diagramme zeigen, dass der Anteil redundanter Fitnessauswertungen in allen Fällen zunächst stark ansteigt und sich dann nur noch geringfügig verändert. Dies zeigt, dass auch bei Läufen mit einer geringen Zahl von Fitnessauswertungen bereits sehr schnell viele redundante Berechnungen durchgeführt werden.

Std+		200000		2423.7 (avg.)
identisch	48163.7	(24.0%)	189.4	(7.8%)
isomorph	137750.0	(68.8%)	312.6	(12.9%)
Step		200000		1691.7 (avg.)
identisch	19141.6	(9.5%)	50.7	(3.0%)
isomorph	71587.8	(35.8%)	89.5	(5.3%)

Tabelle 10.2: Redundante Fitnessberechnungen beim *Lawnmower*-Problem

Die Ergebnisse für das *Lawnmower*-Problem in Tabelle 10.2 und Abbildung 10.3 ähneln qualitativ denen des *Two-Boxes*-Problems. Durch das frühe Finden der gesuchten Lösungen nach durchschnittlich 1691 (*Step*) bzw. 2424 Fitnessauswertungen (*Std+*) ist die Zahl redundanter Fitnessberechnungen zu diesen Zeitpunkten noch relativ gering. Während die Anzahl identischer Individuen bis zum Ende der GP-Läufe jedoch relativ klein bleibt, steigt der Anteil isomorpher Individuen sprunghaft an.

Während die mögliche Zeitersparnis beim *Lawnmower*-Problem – bedingt durch das frühe Finden einer Lösung – durch das Mitprotokollieren von identischen/isomorphen Individuen gering ist, können beim *Artificial Ant*-Problem bis zum Finden der ersten Lösung, bei der

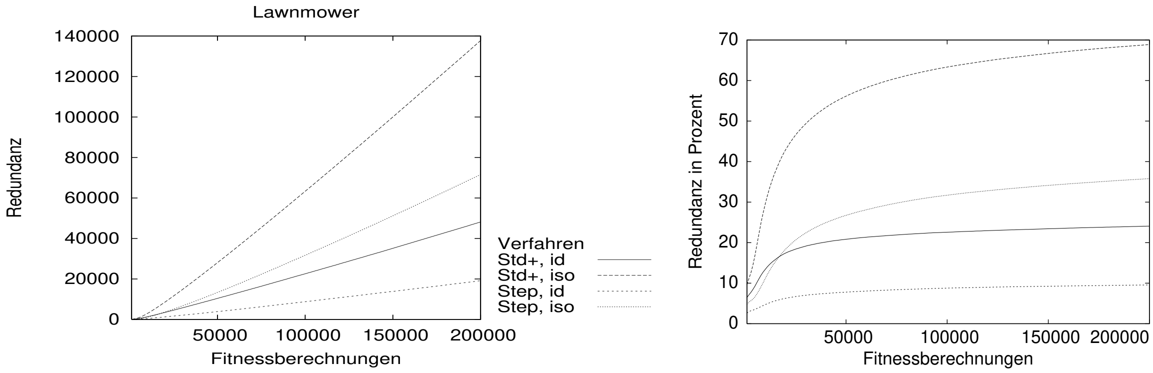


Abbildung 10.3: Redundante Fitnessberechnungen beim *Lawnmower*-Problem

alle Nahrungsstücke eingesammelt werden, – je nach Verfahren – bereits bis zu 47 Prozent der Fitnessauswertungen vermieden werden. Tabelle 10.3 und Abbildung 10.4 fassen die Ergebnisse zusammen. Die beiden senkrechten Linien stehen für die Zeitpunkte, bei denen die Fitness das letzte Mal verbessert wurde.

Std+	200000		91757.3 (avg.)		50306.2 (avg.)	
identisch	84921.4	(42.5%)	34345.8	(37.4%)	18611.3	(37.0%)
isomorph	112325.0	(56.2%)	45459.3	(49.5%)	23605.7	(46.9%)
Step	**200000**		**106003 (avg.)**		**38679.3 (avg.)**	
identisch	29708.8	(14.8%)	12258.6	(11.6%)	3735.4	(9.7%)
isomorph	45329.0	(22.7%)	18697.4	(17.6%)	5337.7	(13.8%)

Tabelle 10.3: Redundante Fitnessberechnungen beim *Artificial Ant*-Problem

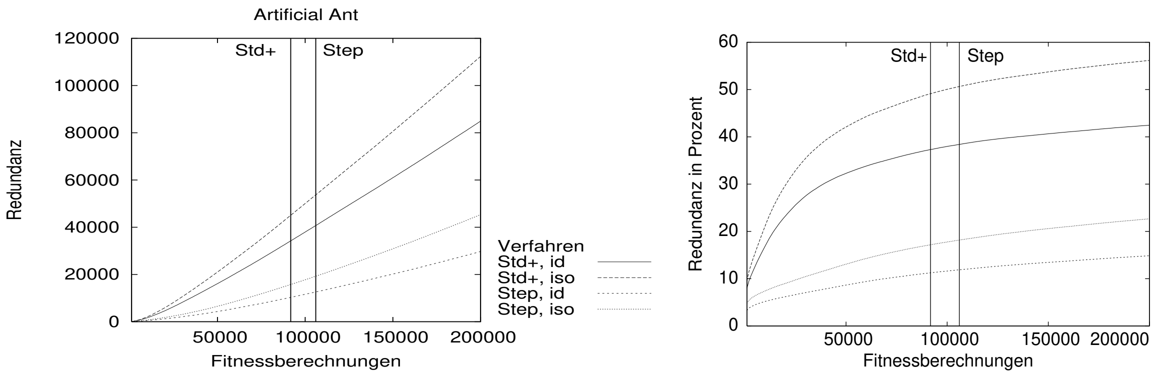


Abbildung 10.4: Redundante Fitnessberechnungen beim *Artificial Ant*-Problem

10.4 Zusammenfassung

Die Ergebnisse der vorangegangenen Abschnitte zeigen, dass bei allen verwendeten Testproblemen deutlich mehr Individuen doppelt erzeugt wurden, als der durch die Operatorwahrscheinlichkeiten implizit vorgegebene Replikationsanteil erwarten ließ. Zusätzlich waren bei allen Benchmarks isomorphe Individuen vorhanden, so dass der Anteil redundant berechneter Fitnesswerte in einzelnen Testreihen auf bis zu 71 Prozent steigen konnte.

Der Anteil redundanter Fitnessberechnungen (sowohl identischer als auch isomorpher) steigt in den Testproblemen während der 200000 durchgeführten Fitnessauswertungen streng monoton an. Zu Beginn eines GP-Laufs ist die Zunahme sehr hoch, nach einer für alle Testserien unterschiedlichen Anzahl von Auswertungen verflacht die entsprechende Kurve jedoch zusehends.

Der Anteil redundanter Fitnessberechnungen ist beim *Std+*-Verfahren wesentlich höher als beim *Step*-Verfahren. Dies zeigt, dass durch die Ausführung mehrerer genetischer Operatoren vor einer Fitnessberechnung der Suchraum wesentlich besser ausgenutzt wird als mit nur einem genetischen Operator. Trotzdem liegt der Anteil redundanter Fitnessberechnungen auch bei diesem Verfahren noch so hoch, dass sich Isomorphiebetrachtungen durchaus lohnen können.

Beim Einsatz der hier vorgestellten Ansätze sind mehrere Punkte zu beachten:

- Die Untersuchung auf identische und isomorphe Graphstrukturen erfordert sowohl Speicherplatz als auch Zeitaufwand. Individuen müssen in einer geeigneten Datenstruktur wie Hashtabellen oder – wie hier angewandt – binären Suchbäumen verwaltet werden. Für eine Zeitabschätzung gilt, dass das Finden von Graphisomorphien zur Klasse NP-unvollständiger Probleme gehört [119] und somit entsprechend Rechenzeit benötigt.[1]

  Dieser Aufwand kann nur in Betracht kommen, wenn die Fitnessberechnung des Problems entsprechend aufwändig ist, so dass zu erwarten ist, dass sich die Rechenzeit für die Redundanzuntersuchungen durch die gesparten Fitnessauswertungen kompensieren lässt.

- Bei vielen GP-Ansätzen spielen aus der Natur bekannte [123] *Introns* und *Bloat* [67, 10] eine wichtige Rolle [2, 1, 76, 64, 84]. Für Individuen, bei denen Teile des Genotyps keinen Einfluss auf die Fitness haben, sind zwar die Fitnesswerte, die verschiedene Individuen erzeugen, identisch, die Genotypen unterscheiden sich jedoch. Solche semantischen Äquivalenzen können mit den hier vorgestellten Methoden nicht erkannt werden. Das graphbasierte System *GGP* hat hier den Vorteil, dass durch die Möglichkeit, kleine Graphstrukturen erfolgreich zu evolvieren, kaum Bloat entsteht. Sollen

[1] Die zur Isomorphiebestimmung eingesetzte Software [75] verwendet effiziente Heuristiken, so dass sich die Rechenzeit bei Problemen mit mehrminütigen Fitnessauswertungen und den Testproblemen ähnlichen Graphgrößen als vermutlich vernachlässigbar ist.

> die hier vorgestellten Isomorphiebetrachtungen auf *Bloat*-anfällige GP-Systeme angewendet werden, so sollten vor der Betrachtung entsprechende Vorkehrungen getroffen werden [107]. Zum Beispiel können Introns aus dem Genotyp entfernt werden [108, 15], die Komplexität oder Größe des Individuums kann, wie bei *GGP*, in die Fitnessfunktion eingehen [44, 125] oder es können spezielle Selektionsmethoden Verwendung finden [106, 30].

Die drei Testprobleme sind sicher nicht repräsentativ für alle Problemstellungen, die mit der Genetischen Programmierung behandelt werden. Die erzielten Ergebnisse geben jedoch Grund zur Annahme, dass gerade bei Problemen mit aufwändiger Fitnessberechnung durch Algorithmen zur Isomorphieerkennung viel Rechenzeit eingespart werden kann. Diese Vermutung wird ebenfalls durch die von IGEL und STAGGE erzielten Ergebnisse für *Neuronale Netze* unterstützt [47].

Kapitel 11

Evolution paralleler Algorithmen

In diesem Kapitel wird gezeigt, wie *GGP* zum Entwurf paralleler Algorithmen eingesetzt werden kann. Dies war bisher mit KOZAs baumbasiertem Grundalgorithmus in dieser Form nicht möglich. In Abschnitt 11.1 wird das *Artificial Ant*-Problem aus Abschnitt 5.3.1 für zwei Ameisen verallgemeinert und in Abschnitt 11.2 wird das parallele Invertieren eines Bitvektores durch zwei Programmthreads behandelt.

Im Rahmen des *Artificial Ant*-Problems wird ein genauerer Blick auf die entstandenen Individuen geworfen, wodurch sich weitere Erkenntnisse über die Fähigkeiten zu Generalisierung ergeben.

Die Ergebnisse dieses Kapitels wurden erarbeitet, bevor die *GGP*-Erweiterungen der Kapitel 7.4 bis 9 existierten. Mit den in diesen Abschnitten vorgestellten GP-Varianten (*Std+*, *Step*,...) wären die Ergebnisse voraussichtlich noch besser ausgefallen, als sie zum Zeitpunkt der Erstellung bereits waren.

In Abschnitt 3.5.2 wird beschrieben, wie algorithmische Semantiken zur Darstellung paralleler Algorithmen eingesetzt werden können. Am Beispiel zweier unterschiedlicher Problemstellungen wird in diesem Abschnitt gezeigt, dass die Evolution solcher Algorithmen mit graphbasiertem GP möglich ist: In Abschnitt 11.1 wird das Problem *Artificial Ant on the Santa Fé Trail* so verallgemeinert, dass zwei Ameisen gleichzeitig auf demselben Feld nach Nahrungsstücken suchen. Das zweite Testproblem besteht darin, mit zwei Threads einen Bitvektor möglichst effizient zu bearbeiten (Abschnitt 11.2).

Die vom GP-System erzeugten Graphen werden im Folgenden als Flussdiagramme dargestellt. In Flussdiagrammen beginnen Algorithmen bei einen ausgezeichneten Startknoten und folgen den Kantenverläufen des Diagramms. Ein Algorithmus terminiert, wenn ein Endknoten erreicht wird. Die Abarbeitung eines Flussdiagramms entspricht somit der Bewertung eines GP-Graphen mit algorithmischer Semantik im sequenziellen Fall. Der Startknoten des Flussdiagramms entspricht dem einzigen Eingangsknoten des Graphen und der Endknoten dem einzigen Ausgangsknoten.

Im Fall paralleler Algorithmen entspricht die Anzahl der Eingangsknoten des GP-Graphen der Anzahl der Threads des Programms, also der Zahl parallel ablaufender Programmflüsse. Es ist weiterhin genau ein Ausgangsknoten vorhanden.

11.1 Der *Santa Fé*-Pfad mit zwei Ameisen

Das *Artificial Ant*-Problem wurde im Rahmen dieser Arbeit bereits mehrfach als Benchmark eingesetzt. Hier soll untersucht werden, ob es möglich ist, mit einem GP-Lauf parallel ablaufende Algorithmen für zwei Ameisen zu evolvieren. Die Ameisen suchen hierbei die Nahrung auf dem gleichen Feld und die Algorithmen teilen sich denselben GP-Graphen.

Folgende Fragen sollen näher untersucht werden:

- Kann ein paralleler Algorithmus gefunden werden, der das *Artificial Ant*-Problem löst? Wie häufig kann eine Lösung gefunden werden?
- Benutzen die beiden Threads exakt die gleichen Teile des Graphen oder verwenden sie jeweils eigene Teilgraphen? Wie groß ist der Anteil des von beiden Threads gemeinsam benutzten Graphen?
- Lassen sich die Lösungen in unterschiedliche Klassen einteilen, die nach dem Anteil des gemeinsam verwendeten Codes unterschieden werden?
- Sind die Lösungen beider Threads generalisierend?
- Lässt sich die Anzahl der Lösungen mit dem sequenziellen Problem vergleichen, wenn der Pfad der Ameise in zwei Testcases mit jeweils der Hälfte des Weges aufgeteilt wird?

11.1.1 Problembeschreibung

Das Problem entspricht weitestgehend dem in Abschnitt 5.3.1 vorgestellten, allerdings werden in diesem Fall zwei Ameisen auf dem Feld positioniert (Abbildung 11.1).

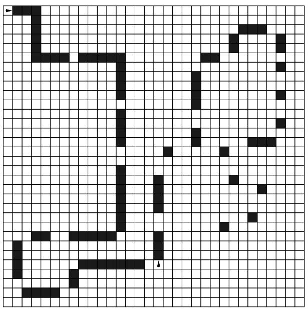
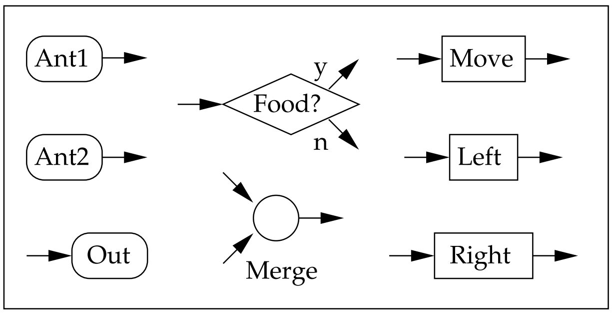



Abbildung 11.1: Der *Santa Fé*-Pfad mit zwei Ameisen. Ein Graph enthält zwei Eingangsknoten, jeder steht für den Startknoten einer Ameise.

Die Menge der Grundfunktionen umfasst die Funktionen *Food, Move, Left, Right* und *Merge*. Im Unterschied zum sequenziellen Fall mit einer Ameise besitzt ein GP-Graph zur Repräsentation eines Ameisenalgorithmus einerseits zwei Eingangsknoten und andererseits eine andere Parametermenge für die Grundfunktionen. In dieser werden zusätzlich die Daten der zweiten Ameise repräsentiert.

Die Parametermenge besteht somit aus einem 4-Tupel $P = (food, ant_1, ant_2, cmd)$. Die Menge $food \subseteq \{(0,0)\dots(0,31),(1,0)\dots(1,31),\dots,(31,0)\dots(31,31)\}$ repräsentiert die Felder, auf denen zu Beginn Nahrungsstücke vorhanden sind. Die Menge ist identisch mit der aus Abschnitt 5.3.1. Auch ant_1 ist dieselbe Menge geblieben und repräsentiert die Startposition der ersten Ameise. Das Tupel $ant_2 = ((26,21),(31,0))$ steht für Startposition und Blickrichtung der zweiten Ameise. Die Zahl cmd $\in \mathbb{N}$ gibt die maximale Befehlszahl an und wurde auf 600 gesetzt.

Somit ergibt sich die Parametermenge P_0 und die Funktionenschar F

$$\begin{aligned} P_0 = (\{&(0,1),(0,2),(0,3),(1,3),(2,3),(3,3),(4,3),(5,3),(5,4),(5,5),(5,6),(5,8),(5,9),\\ &(5,10),(5,11),(5,12),(6,12),(7,12),(8,12),(9,12),(11,12),(12,12),(13,12),\\ &(14,12),(17,12),(18,12),(19,12),(20,12),(21,12),(22,12),(23,12),(24,11),\\ &(24,10),(24,9),(24,8),(24,7),(24,4),(24,3),(25,1),(26,1),(27,1),(28,1),\\ &(30,2),(30,3),(30,4),(30,5),(29,7),(28,7),(27,8),(27,9),(27,10),(27,11),\\ &(27,12),(27,13),(27,14),(26,16),(25,16),(24,16),(21,16),(20,16),(19,16),\\ &(18,16),(15,17),(14,20),(13,20),(10,20),(9,20),(8,20),(7,20),(5,21),(5,22),\\ &(4,24),(3,24),(2,25),(2,26),(2,27),(3,29),(4,29),(6,29),(9,29),(12,29),\\ &(14,28),(14,27),(14,26),(15,23),(18,24),(19,27),(22,26),(23,23)\},\\ &((0,0),(0,1)),((26,21),(31,0)),600) \end{aligned}$$

und $\qquad F = \{\text{Food, Move, Left, Right, Merge}\} \qquad$ mit

$$\begin{aligned} \text{Food}:&\quad P \to P \times \{1,2\} \times \{1\} \times \{2\}\\ \text{Left}:&\quad P \to P \times \{1\} \times \{1\} \times \{1\}\\ \text{Right}:&\quad P \to P \times \{1\} \times \{1\} \times \{1\}\\ \text{Move}:&\quad P \to P \times \{1\} \times \{1\} \times \{1\}\\ \text{Merge}:&\quad P \to P \times \{1\} \times \{2\} \times \{1\} \end{aligned}$$

sowie

$$\text{Food}: \quad (\text{food}, (a_1^1, a_1^2), (a_2^1, a_2^2), \text{cmd}) \mapsto (\text{food}, (a_1^1, a_1^2), (a_2^1, a_2^2), \text{cmd}) \times j \times 1 \times 2$$

$$\text{mit} \quad j = \begin{cases} 1 & \textit{falls}((a_k^1 + a_k^2)\ \textit{mod}\ 32) \in \text{food}^1 \\ 2 & \textit{sonst} \end{cases}$$

$$\text{Left}: \quad (\text{food}, (a_1^1, a_1^2), (a_2^1, a_2^2), \text{cmd}) \mapsto (\text{food}, (a_1^1, \hat{a}_1^2), (a_2^1, \hat{a}_2^2), \widehat{\text{cmd}}) \times 1 \times 1 \times 1$$

[1] Die Modulo-Rechnung wird auf beiden Komponenten des Tupels $(a_k^1 + a_k^2)$ einzeln durchgeführt

$$\hat{a}_k^2 = \begin{cases} (31,0) & \text{falls } a_k^2 = (0,1) \\ (0,31) & \text{falls } a_k^2 = (31,0) \\ (1,0) & \text{falls } a_k^2 = (0,31) \\ (0,1) & \text{falls } a_k^2 = (1,0) \end{cases} \qquad \hat{a}_{3-k}^2 = a_{3-k}^2, \qquad \widehat{\text{cmd}} = \text{cmd} - 1$$

$$\text{Right}: \quad (\text{food}, (a_1^1, a_1^2), (a_2^1, a_2^2), \text{cmd}) \mapsto (\text{food}, (a_1^1, \hat{a}_1^2), (a_2^1, \hat{a}_2^2), \widehat{\text{cmd}}) \times 1 \times 1 \times 1$$

$$\hat{a}_k^2 = \begin{cases} (1,0) & \text{falls } a_k^2 = (0,1) \\ (0,1) & \text{falls } a_k^2 = (31,0) \\ (31,0) & \text{falls } a_k^2 = (0,31) \\ (0,31) & \text{falls } a_k^2 = (1,0) \end{cases} \qquad \hat{a}_{3-k}^2 = a_{3-k}^2, \qquad \widehat{\text{cmd}} = \text{cmd} - 1$$

$$\text{Move}: \quad (\text{food}, (a_1^1, a_1^2), (a_2^1, a_2^2), \text{cmd}) \mapsto (\widehat{\text{food}}, (\hat{a}_1^1, a_1^2), (\hat{a}_2^1, a_2^2), \widehat{\text{cmd}}) \times 1 \times 1 \times 1$$

$$\begin{aligned} \hat{a}_k^1 &= (a_k^1 + a_k^2)\ mod\ 32, & \hat{a}_{3-k}^1 &= a_{3-k}^1, \\ \widehat{\text{food}} &= \text{food} \setminus \{(a_k^1 + a_k^2)\ mod\ 32\}, & \widehat{\text{cmd}} &= \text{cmd} - 1 \end{aligned}$$

$$\text{Merge}: \quad (\text{food}, (a_1^1, a_1^2), (a_2^1, a_2^2), \text{cmd}) \mapsto (\text{food}, (a_1^1, a_1^2), (a_2^1, a_2^2), \text{cmd}) \times 1 \times 2 \times 1$$

Hierbei bezeichnet k immer die aktuell bearbeitete Ameise.

Tabelle 11.1 zeigt die Initialisierungen der GP-Variablen. Diese Experimente wurden ausgeführt, bevor die genetischen Operatoren kontextsensitiv ausgewählt werden konnten (Abschnitt 7.4) und bevor das Konzept der *dynamischen Demes* aus Abschnitt 9.1 verwirklicht wurde. Für die genetischen Operatoren in diesem Test wurde deshalb das *Idp*-Verfahren aus Abschnitt 7.2.4 zusammen mit der Schrittweitensteuerung aus Abschnitt 8.2 verwendet.

Parameter	Wert
Populationsgröße	1000
Fitnessauswertungen	200000
Turniergröße	4/2
Graphgröße	25
Crossover	*random*
Läufe	200
Besonderheiten	Schrittweitensteuerung adaptive Operatorwahrscheinlichkeiten (*Idp*) **keine kontextsensitiven Operatoren** zyklische Graphen

Tabelle 11.1: GP-Variablen für das *Artificial Ant*-Problem mit zwei Ameisen

11.1.2 Bewertung des Graphen

Die Auswertung eines GP-Graphen folgt der Beschreibung aus Abschnitt 3.5.2. Der Algorithmus der ersten Ameise beginnt beim Eingangsknoten 1, die zweite Ameise startet beim Eingangsknoten 2.

Das *Artificial-Ant*-Problem sieht keine Möglichkeit für ein Abbruchkriterium nach dem Finden sämtlicher Nahrungsstücke vor. Aus diesem Grund laufen beide Threads des Algorithmus so lange, bis die Anzahl zulässiger *Move-*, *Left-* und *Right*-Befehle aufgebraucht wurde. Erreicht einer der Threads den Ausgangsknoten, fängt er wieder bei seinem Eingangsknoten neu an.

Bei den gewählten Grundfunktionen ist es möglich, dass sich beide Threads in einer Endlosschleife befinden. In so einem Fall wird die Auswertung des Graphen abgebrochen und dem Individuum der schlechteste aller möglichen Fitnesswerte zugewiesen. Eine Endlosschleife kann entstehen, wenn ein Zyklus im Graphen nur aus Eingangs-, Ausgangs-, *Food-* oder *Merge*-Knoten besteht, da der Befehlszähler *cmd* bei der Ausführung der entsprechenden Funktionen nicht reduziert wird. Eine Endlosschleife ist auf folgende Weise leicht zu erkennen: Wurden mehr Knoten ohne Veränderung der Ameisenstellung in Folge betrachtet als der GP-Graph insgesamt Knoten hat, muss ein Zyklus für den entsprechenden Thread vorliegen.

Die Fitness eines GP-Graphen setzt sich aus mehreren Kriterien zusammen:

1. die Anzahl der am Ende nicht gefundenen Nahrungsstücke, also die Kardinalität der Menge *food*
2. die Anzahl der Befehle, die bis zur letzten Nahrungsaufnahme ausgeführt wurden
3. die Größe des GP-Graphen

Die Anzahl der aufgenommenen Nahrungsstücke liegt zwischen 0 und 89. Je mehr Nahrungsstücke gefunden wurden, desto kleiner wird diese Zahl. Es spielt hierbei keine Rolle, welche der Ameisen die Nahrung aufgenommen hat.

Der Wert *cmd* aus der Parametermenge P der Grundfunktionen wird nach jeder aufgenommenen Nahrung gespeichert. Aus dem letzten dieser Werte ergibt sich nach Beendigung des Graphdurchlaufs die Anzahl der Befehle, die bis zum Aufnehmen der letzten Nahrung benötigt wurden. Je geringer diese Anzahl ist, desto weniger Umwege haben die Ameisen gemacht. Die Ameisen dürfen maximal 600 *Move-*, *Left-* oder *Right-* Anweisungen durchführen. Die Anzahl der von beiden Ameisen zusammen ausgeführten Bewegungen geht durch 10000 dividiert als zweites Kriterium in die Fitness ein. Haben die Ameisen beispielsweise 86 Nahrungsstücke aufgenommen – also drei übersehen – und dafür 550 Bewegungen benötigt, ergibt sich eine Fitness von $3 + 550/100000 = 3.055$.

Die Größe eines GP-Graphen wird als sekundäres Fitnesskriterium verwendet. Haben in einem Turnier zwei Individuen die ansonsten gleiche Fitness, gilt das kleinere als das bessere der Individuen. Wie später zu sehen sein wird, können so Introns stark reduziert bzw. komplett beseitigt werden.

11.1.3 Ergebnisse

Abbildung 11.2 zeigt die Ergebnisse der 200 GP-Läufe. In 70 Prozent aller Fälle konnte ein paralleler Algorithmus gefunden werden, bei dem alle Nahrungsstücke aufgesammelt wurden. Im Diagramm ist für jeden GP-Lauf die Fitness des besten Individuums nach 200000 Fitnessauswertungen aufgetragen. Um die Grafik übersichtlicher zu gestalten, wurden die Läufe nach der Fitness des besten Individuums sortiert.

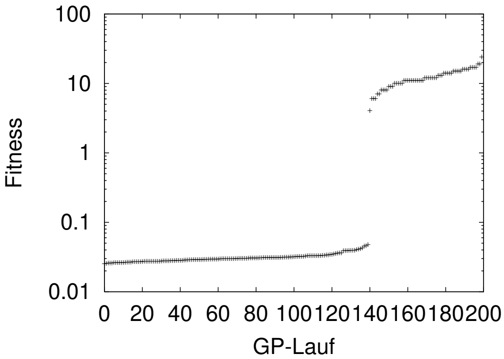
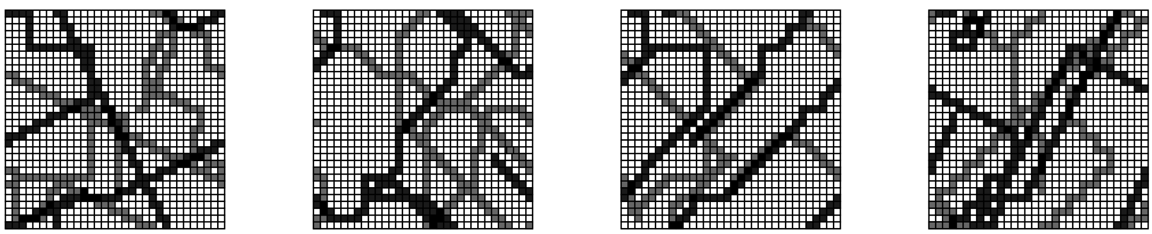


Abbildung 11.2: Die Fitnesswerte des jeweils besten Individuums aller 200 GP-Läufe beim *Santa Fé*-Pfad für zwei Ameisen.

Die Fitness des jeweils besten Individuums der 140 erfolgreichen GP-Läufe liegt zwischen 0.0253 und 0.0478. Der Durchschnitt dieser Werte liegt bei 0.031042. Dies bedeutet, dass die Ameisen zusammen nach im Schnitt 310.42 Bewegungen alle 89 Nahrungsstücke aufgesammelt haben.

Bei den 60 GP-Läufen, die zu keiner Lösung führten, übersahen die Ameisen zwischen 4 und 24 Nahrungsstücke – im Durchschnitt 10.4. Abbildung 11.3 zeigt exemplarisch die Ergebnisse von vier gescheiterten GP-Läufen.



Food left: 4 | Food left: 10 | Food left: 15 | Food left: 19

Abbildung 11.3: Die Spuren der jeweils besten Ameisenpaare aus vier gescheiterten GP-Läufen

Mit einer Ausnahme verfolgt keine einzige Ameisenpaarung aus der jeweils letzten Generation der gescheiterten Läufe den Santa Fé-Pfad. Vielmehr laufen alle Ameisen quer über das Feld und finden hierbei eher zufällig die Nahrungsstücke. Bei einem Lauf, bei dem sich beide Ameisen im besten Individuum an den Pfad halten, ist bei längerer Laufzeit das Finden einer Lösung zu erwarten. In allen Läufen, die zu einer Lösung führten, wurde bei einer ähnlichen Situation nach spätestens 20000 weiteren Funktionsauswertungen eine korrekte Lösung gefunden. In den meisten Fällen reichten sogar 2000 weitere Auswertungen aus.

Die Evolution verlief in allen erfolgreichen GP-Läufen zweistufig. In der ersten Phase wurde eine Lösung gefunden, bei der alle Nahrungsstücke aufgesammelt wurden. Danach schloss sich eine Phase an, in der die Anzahl der von den Ameisen benötigten Gehbewegungen reduziert wurde.

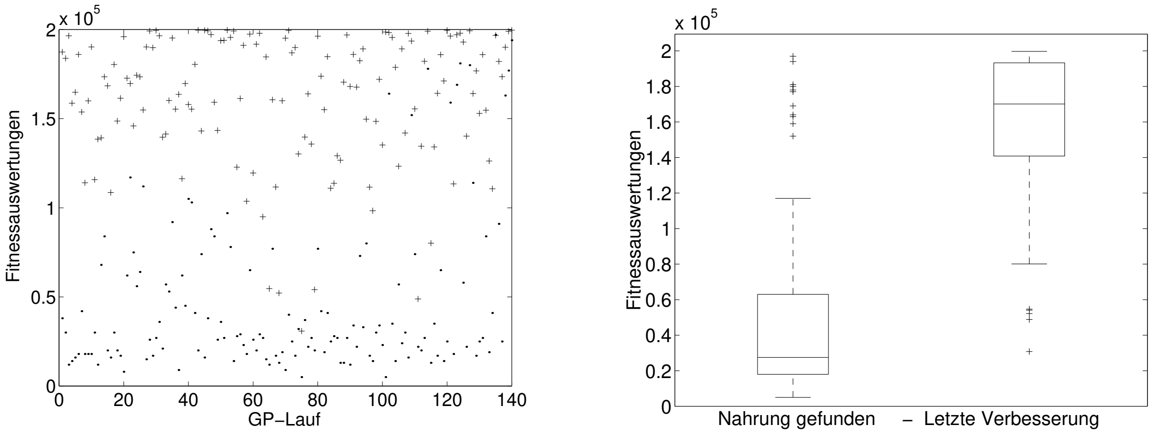


Abbildung 11.4: Die Anzahl der Fitnessauswertungen in den 140 erfolgreichen Läufe, nach denen erstmals ein Individuum gefunden wurde, das alle 89 Nahrungsstücke finden konnte sowie die Anzahl der Fitnessauswertungen nach denen das jeweils beste Individuum der Läufe entstand.

Abbildung 11.4 verdeutlicht diese Zweiteilung. Im linken Diagramm sind für alle 140 erfolgreichen Läufe die Zeitpunkte eingetragen, in denen der Übergang von der ersten in die zweite Phase stattfindet. Für jeden Lauf sind zwei Markierungen vorhanden. Die untere gibt an, zu welchem Zeitpunkt der erste Algorithmus gefunden wurde, der alle Nahrungsstücke findet. Die obere Markierung steht für den Zeitpunkt, zu dem die letzte Fitnessverbesserung des Laufs eingetreten ist. Dies ist also der Zeitpunkt, zu dem sich die Anzahl der benötigten Ameisenbewegungen zum letzten Mal reduziert hat.

Das rechte Boxplot-Diagramm verdeutlicht, zu welchen Zeitpunkten innerhalb der GP-Läufe die beiden Phasen hauptsächlich stattfanden. Im linken Boxplot sind die Zeitpunkte der 140 erfolgreichen Läufe zusammengefasst, zu denen das erste Mal ein Individuum alle Nahrungsstücke eingesammelt hat. Die Begrenzungen der Box stehen hierbei für das untere Quartil, den Meridian und das obere Quartil dieser 140 Werte. Im rechten Boxplot sind die letzten Fitnessverbesserungen innerhalb der erfolgreichen Läufe zusammengefasst.

Die Abbildung verdeutlicht, dass innerhalb der GP-Läufe relativ schnell Algorithmen gefunden wurden, mit denen sämtliche Nahrungsstücke aufgenommen werden. In der Hälfte aller erfolgreichen Läufe war dies bereits nach 28000 Fitnessauswertungen der Fall.

Die meiste Zeit der Läufe wird für eine Verkürzung der Wege benötigt, die die beiden Ameisen zurücklegen. Der rechte Boxplot, der die Zeitpunkte der letzten Wegverkürzungen zusammenfasst, macht deutlich, dass diese Optimierung in den meisten Läufen noch nicht abgeschlossen ist, da nur bei wenigen Läufen eine Stagnation der letzten erreichten Fitness über einen längeren Zeitraum zu beobachten ist.

Da beide Ameisen insgesamt nur 600 Schritte machen dürfen, ist es nicht möglich, alle Nahrungsstücke mit einem periodischen Schrittmuster einzusammeln. Die Begrenzung auf 25 Knoten in den Graphen verhindert außerdem ein Auswendiglernen des kompletten Pfades.

Daher ist zu erwarten, dass die Ameisen, die sämtliche Nahrungsstücke finden, alle einer Variante des Grundalgorithmus aus Abbildung 11.5 folgen: Eine Ameise betrachtet die drei Felder vor, rechts und links von sich. Liegt auf einem dieser Felder ein Nahrungsstück, bewegt sie sich in diese Richtung. Sind alle drei Felder leer, bewegt sie sich ein Feld in die ursprüngliche Richtung vor und beginnt die Nahrungssuche aufs Neue.

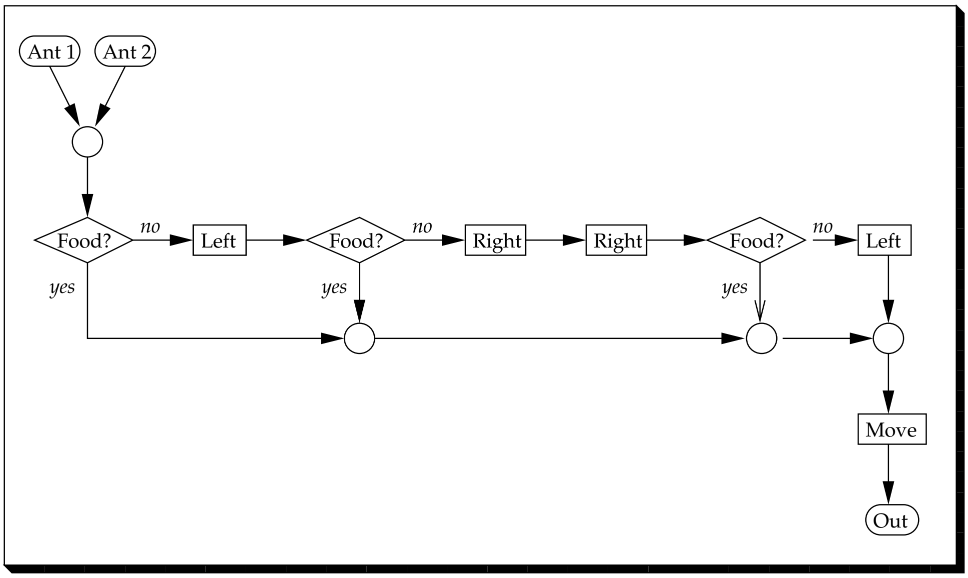


Abbildung 11.5: Eine mögliche graphische Repräsentation des Grundalgorithmus der Ameisen für den *Santa Fé*-Pfad

Wenn sich beide Ameisen an dieses Grundschema halten, benötigen sie für das Finden aller Nahrungsstücke 392 oder 396 Schritte: Im Laufschema der Ameisen ist es ohne Bedeutung, ob sie sich zuerst nach links oder rechts drehen. In beiden Fällen werden auf dem *Santa Fé*-Pfad alle Nahrungsstücke gefunden. Die Anzahl der Linksdrehungen, die Ameise 2 durchführen muss, unterscheidet sich allerdings von der Zahl ihrer Rechtsdrehungen. Je nach Reihenfolge der beiden Drehungen im Algorithmus variiert somit die Anzahl der benötigten Schritte.

Tabelle 11.2 verdeutlicht, aus welchen Bewegungstypen die Pfade der beiden Ameisen bestehen. Wenn eine Ameise ein Nahrungsstück vor sich hat, benötigt sie einen Schritt, um dieses aufzunehmen. Liegt weder vor noch neben ihr ein Nahrungsstück, braucht sie für

Bewegungsmuster	Ameise 1	Ameise 2
Nahrungsaufnahme	55	34
Linksdrehung	4	5
Rechtsdrehung	4	7
Lücke im Pfad	14	39

Tabelle 11.2: Der Pfad der beiden Ameisen setzt sich aus den hier aufgeführten einzelnen Bewegungen zusammen. *Lücke im Pfad* bedeutet hierbei, dass die Ameise, nachdem sie nach rechts und links gesehen hat, in die ursprüngliche Richtung weiterläuft.

die nächste Fortbewegung fünf Schritte – vier Drehbewegungen und das abschließende Vorwärtsbewegen. Liegt entweder zur linken oder zur rechten ein Nahrungsstück, braucht die Ameise eine oder drei Bewegungen. Dies hängt davon ab, ob die Seite, auf der die Nahrung liegt, durch den Algorithmus zuerst (eine Drehung) oder als zweites (drei Drehungen) betrachtet wird.

Bei den gefundenen Lösungsalgorithmen benötigen nur die wenigsten genau 392 oder 396 Schritte. Der erste Grund hierfür ist, dass die Ameise, die zuerst alle ihre Nahrungsstücke gefunden hat, trotzdem weiterläuft, bis auch die langsamere Ameise ebenfalls die Nahrung vollständig aufgesammelt hat. Diese überzähligen Schritte werden bei der Fitnessberechnung mitgezählt.

Von den erfolgreichen Individuen benötigen 126 weniger als 390 Schritte. Dies bedeutet, dass die evolvierten Algorithmen keine exakte Implementierung des oben beschriebenen allgemeinen Schemas sein können.

Um dies genauer zu untersuchen, wurde etwa ein Drittel aller evolvierten Graphen eingehender untersucht. Hierbei ergab sich, dass alle erfolgreichen Algorithmen zwar am Grundalgorithmus angelehnt sind, die Ameisen jedoch durch den Verzicht auf überflüssige Drehungen beschleunigt werden.

Ein typisches Beispiel hierfür ist in Abbildung 11.6 dargestellt. Es handelt sich um das beste Individuum aus einem der Läufe, dessen Ameisen insgesamt 300 Schritte zur Nahrungsaufnahme benötigen. Die Grundstruktur bezüglich der Drehbefehle und der *Food*-Abfragen entspricht der aus Abbildung 11.5. Wird allerdings ein Nahrungsstück gefunden, werden mehrere Vorwärtsbewegungen auf einmal durchgeführt:

- Wird nach eine Linksdrehung Nahrung gefunden, geht die Ameise automatisch zwei Felder vorwärts.

- Wird nach einer Rechtsdrehung Nahrung gefunden, so geht die Ameise automatisch vier Felder vorwärts.

- Liegen zwei Nahrungsstücke hintereinander, geht die Ameise automatisch drei Felder weiter.

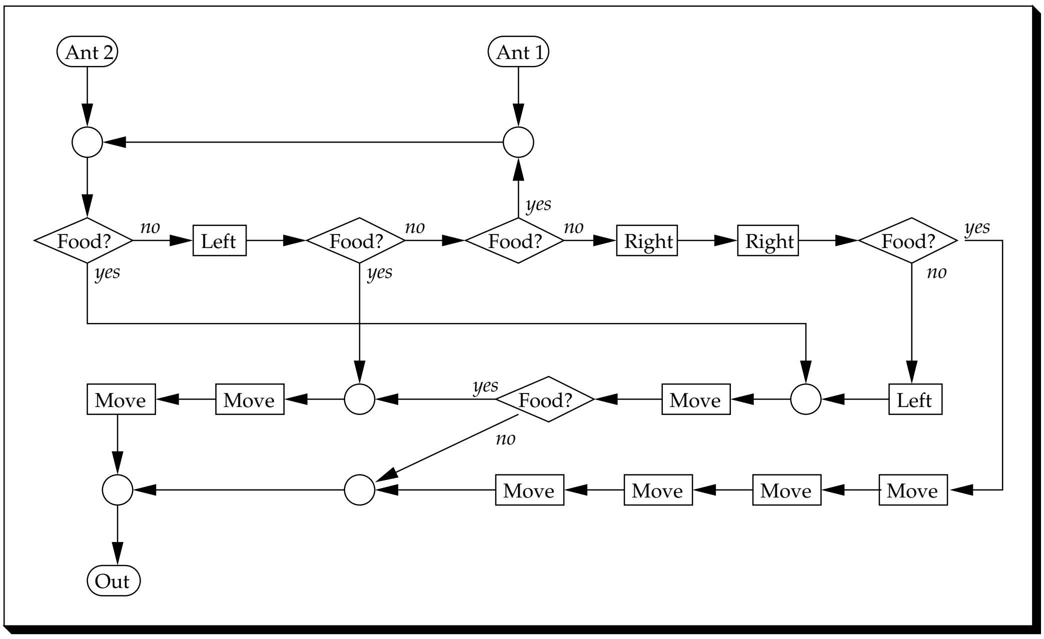


Abbildung 11.6: Ein evolviertes Individuum mit Fitness 0.0300

- Wurde zwar keine Nahrung in einem der benachbarten Feldern gefunden, befindet sich aber ein Nahrungsstück auf dem übernächsten Feld, geht die Ameise noch ein Feld über dieses hinaus.

All diese beschleunigten Vorwärtsbewegungen sind auf dem Santa-Fé-Pfad möglich, gehen aber über den erwarteten Grundalgorithmus hinaus. Jedes Mal, wenn durch die zusammenhängenden *Move*-Anweisungen Felder ohne Nahrung übergangen werden, hat die Ameise vier Drehbewegungen eingespart.

Ein weiterer Unterschied der meisten evolvierten Individuen zum Grundschema aus Abbildung 11.5 ist der Einstiegspunkt in den Algorithmus. Abbildung 11.7 gibt hierfür ein Beispiel. Es handelt sich hierbei um das beste Individuum aus einem der erfolgreichen Läufe. Um das Problem des verschobenen Einstiegspunktes deutlicher zu machen, wurden sämtliche Introns entfernt.

Bei beiden Ameisen wird zuerst überprüft, ob ein Nahrungsstück im Feld voraus liegt. Ist dies nicht der Fall, erfolgt eine Rechtsdrehung, gefolgt von einer Vorwärtsbewegung. Danach befindet sich die Ameise an der Stelle des Algorithmus, die im Grundalgorithmus aus Abbildung 11.5 am Anfang steht. Der Algorithmus kann deshalb nur funktionieren, wenn beide Ameisen direkt vor einem Nahrungsstück starten. Diese Eigenschaft ist beim *Santa Fé*-Pfad erfüllt. Von den untersuchten GP-Individuen hatten 53 alle Nahrungsstücke gefunden. Bei diesen Algorithmen wurde die beschriebene Eigenschaft alleine in 27 Fällen ausgenutzt. Dies verdeutlicht sehr anschaulich, dass sowohl die Problemstellung als auch die Fitnessfunktionen für eine angestrebte Generalisierung exakt definiert werden müssen.

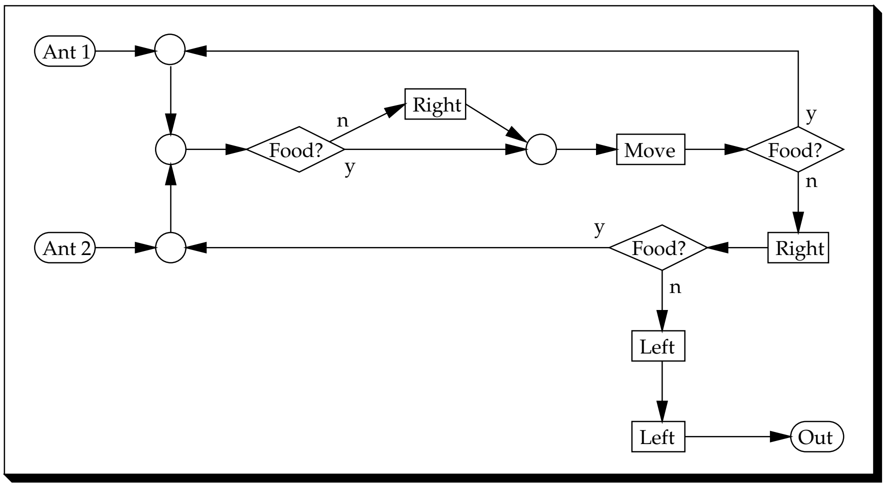


Abbildung 11.7: Ein von Introns befreites Individuum, bei dem der Start des Grundalgorithmus nicht am Anfang des Graphen liegt. Die Fitness beträgt 0.0392.

11.1.4 Gemeinsam genutzter Code

Die Gesamtzahl von 25 Knoten innerhalb eines Graphen zwingt die beiden Ameisen, Teile des Graphen gemeinsam zu nutzen. Es ist zwar denkbar, dass der Grundalgorithmus oder eine seiner beschleunigten Varianten für jede Ameise einzeln in den Knoten codiert ist. Allerdings ist dies sehr unwahrscheinlich und bei den untersuchten Individuen nicht vorgekommen.

Wenn sich derselbe Algorithmus innerhalb des Graphen parallel zweimal entwickelt, ist es für die Fitness des Individuums besser, den einen Teil zu löschen und durch einen Verweis auf die zweite Realisierung zu ersetzen, da kleinere Graphen bei sonst gleicher Fitness bevorzugt werden. Ist die Fitness der Implementierung einer der beiden Ameisen etwas besser als die der anderen, würde das Löschen des schlechteren Teiles ebenfalls zu einer Fitnessverbesserung führen.

Somit ergeben sich folgende Fragen, die zumindest qualitativ im Folgenden beantwortet werden sollen:

1. Wie groß ist der Anteil der Knoten, die exakt einer Ameise zugeordnet werden können?

2. Benutzen beide Ameisen den gemeinsamen Code auf dieselbe Weise?

Die Analyse von erfolgreichen Individuen ergab, dass sich keine dieser Fragen eindeutig beantworten lässt, da alle denkbaren Kombinationen vertreten waren.

Zur Beantwortung der ersten Frage muss zunächst geklärt werden, welche Knoten einer Ameise zugeordnet werden können. Dies führt direkt zur Frage, welche Knoten als Introns aufgefasst werden können. Betrachtet man die Definition aus [9], so sind alle Introns Eigenschaften des Genotyps, die keinen Einfluss auf die Fitness eines Individuums haben. Folgt man dieser Definition und nimmt weiter an, dass der Graph eines Individuums als Genotyp aufgefasst wird und der Weg, den eine Ameise abläuft, der Phänotyp ist, so bleiben als Introns für eine Ameise nur die Knoten des Graphen, die während der Ausführung des Algorithmus von dieser nicht erreicht werden können.[1] Abbildung 11.8 gibt hierfür zwei Beispiele. Im linken Beispiel ist der untere *Food*-Knoten kein Intron. Ein Entfernen des Knotens würde zwar keine direkte Veränderung des Pfades der Ameise bewirken, der nachfolgende *Move*-Knoten würde aber einen Schritt früher ausgeführt. Durch diesen Schritt könnte die eine Ameise ein Nahrungsstück aufnehmen, das im ursprünglichen Fall vorher Ziel einer *Food*-Abfrage der anderen Ameise gewesen wäre. Somit kann die Entfernung des Knotens zu komplett anderen Pfaden auf dem Feld führen.

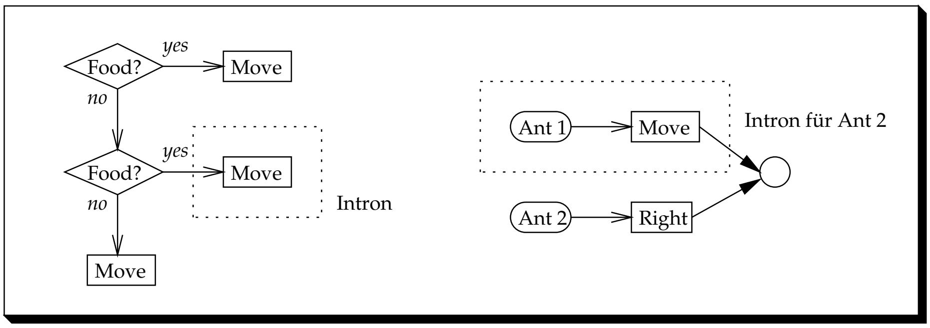


Abbildung 11.8: Zwei Beispiele für Introns innerhalb eines Individuums. Im zweiten Beispiel ist der gekennzeichnete Bereich nur für die zweite Ameise ein Intron, da sie diesen Code niemals ausführt. Betrachtet man den parallelen Algorithmus als Ganzen, so sind diese Knoten kein Intron, da sie von Ameise 1 benutzt werden.

Somit können alle Knoten, die für genau eine Ameise Introns darstellen, der jeweils anderen Ameise als exklusive Knoten zugeordnet werden. Introns, die sich auf den kompletten Algorithmus beziehen, zählen in dieser Betrachtung zum gemeinsamen Teilgraphen beider Ameisen.

In Abbildung 11.9 sind die Ergebnisse der Analyse von 52 untersuchten erfolgreichen Individuen dargestellt. Eine Untersuchung von weiteren Individuen ist an dieser Stelle nicht erforderlich, da bereits qualitative Aussagen gemacht werden können und quantitative Aussagen nicht ohne weiteres auf andere Probleme übertragbar wären.

Im Diagramm sind auf der x-Achse die Anzahl der Schritte abgetragen, die ein Ameisenpaar zum Einsammeln sämtlicher Nahrungsstücke benötigt. Der Wert der y-Achse ist die Summe der Knoten, die exklusiv einer der beiden Ameise zugeordnet werden können. Da bei allen Graphen zwei Eingangsknoten vorhanden sind, die den Start der beiden Ameisen repräsentieren, werden diese nicht mitgezählt.

[1] Die in [9] als Beispiel für Introns angegebene *Left/Right*-Kombination ist in der gewählten Problemmodellierung kein Intron, da zusätzliche Drehungen eine Veränderung der Fitness zur Folge haben.

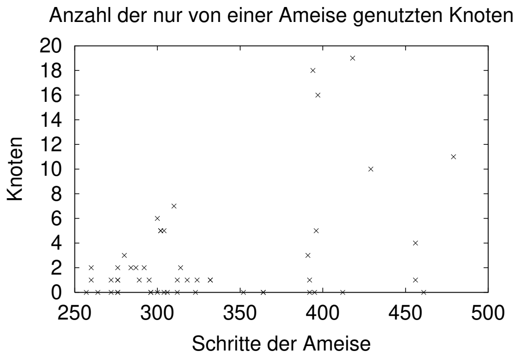


Abbildung 11.9: Die Anzahl der Knoten, die in den untersuchten erfolgreichen Individuen nur von einer Ameise benutzt werden. Die Eingangsknoten der beiden Ameisen werden nicht mitgezählt.

Individuen, die deutlich weniger Schritte als die vom Grundalgorithmus vorgegebenen machen, haben nur sehr wenige Knoten, die eindeutig einer Ameise zuzuordnen sind. Bei insgesamt neun der Individuen war der Algorithmus beider Ameisen sogar vom Einstiegspunkt an identisch, da die beiden Eingangsknoten der Ameisen durch einen *Merge*-Knoten miteinander verbunden waren. Bei Ameisenpaarungen, die mindestens 392 Schritte oder mehr benötigen, werden in der Tendenz weniger Knoten von beiden Ameisen gleichermaßen benutzt.

Somit lässt sich die Frage nach dem Anteil gemeinsam benutzten Codes qualitativ so beantworten, dass bei guten Individuen Code eher gemeinsam benutzt wird als bei schlechten.

Für die Beantwortung der Frage nach der Verwendung des gemeinsamen Codes zerfällt die analysierte Algorithmenmenge grundsätzlich in zwei Gruppen: die guten Individuen mit großem gemeinsamen Codeanteil und die schlechteren, bei denen viele Knoten einer einzelnen Ameise zuzuordnen sind. Die wenigen Ausnahmen werden anschließend gesondert betrachtet, sind aber aufgrund ihrer geringen Anzahl nicht aussagekräftig.

Individuen mit kleinem gemeinsamen Codeanteil Bei diesen Individuen ist es in der Regel so, dass eine Ameise über einen Großteil des Graphen verfügt, während die andere nur relativ wenige Knoten exklusiv benutzt. In diesen Fällen entspricht der Teilalgorithmus der einen Ameise dem Grundalgorithmus aus Abbildung 11.5, während die zweite Ameise am Einsammeln der Nahrungsstücke nicht beteiligt ist. Abbildung 11.10 zeigt ein entsprechendes Individuum.

Die erste Ameise verfolgt den Grundalgorithmus, der hier durch drei *Move*-Befehle nach einer Linksdrehung und vier *Move*-Befehle nach einer Rechtsdrehung variiert wird. Des Weiteren fängt der Algorithmus mit einem *Move*-Befehl an, so dass eine Ausführung auf einem Pfad scheitern könnte, bei dem das erste Nahrungsstück links oder rechts von der Ameise liegen würde. Die zweite Ameise dreht sich ausschließlich im Kreis. Das Individuum benötigt insgesamt 394 Schritte. Die Bewegungen der zweiten Ameise werden ge-

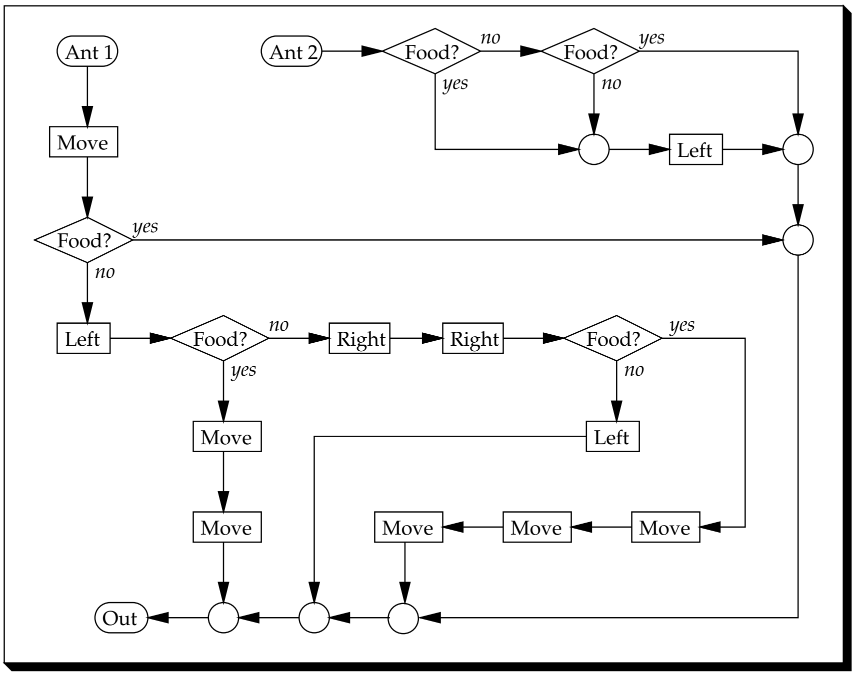


Abbildung 11.10: Ein Algorithmus, bei dem eine Ameise den gesamten Pfad abläuft, während sich die zweite im Kreis dreht.

genüber der Standardzahl von 392 bzw. 396 Schritten durch die Kombination mehrerer *Move*-Knoten der ersten Ameise kompensiert. Es ist wichtig, dass die zweite Ameise während der Ausführung des Algorithmus möglichst viele *Food*- und *Merge*-Knoten betrachten muss, durch welche die Anzahl der Schritte nicht erhöht wird. Auf diese Weise wird die Anzahl der Linksdrehungen der Ameise gesenkt, die ausgeführt werden müssen, bis die andere Ameise alle Nahrungsstücke eingesammelt hat. Eine Entfernung eines der beiden *Food*-Knoten der zweiten Ameise würde somit zu einer größeren Schrittzahl und einer schlechteren Fitness führen.

Individuen mit großem gemeinsamen Codeanteil. In dieser Kategorie lässt sich kein eindeutiges Schema erkennen. In vielen Fällen wird der gemeinsame Teilgraph auf identische Weise genutzt, in anderen wiederum werden mehrere aufeinanderfolgende *Move*-Anweisungen von den beiden Ameisen an verschiedenen Stellen des Grundalgorithmus eingefügt. Im Beispiel aus Abbildung 11.11 läuft die zweite Ameise bei einem direkt vor ihr liegenden Nahrungsstück drei Felder vorwärts, während sich Ameise 1 in einer entsprechenden Situation nur ein Feld weiterbewegt. Haben beide Ameisen kein Nahrungsstück vor sich, verfolgen sie ansonsten denselben Weg durch den Graphen.

Die initiale Fortbewegung um 3 Felder ist auf dem *Santa Fè*-Pfad nur auf dem Teilstück der zweiten Ameise möglich. Die erste Ameise würde nach der zweiten Rechtskurve auf dem langen senkrechten Teilstück die nächste Abzweigung verpassen. Die Kette von drei

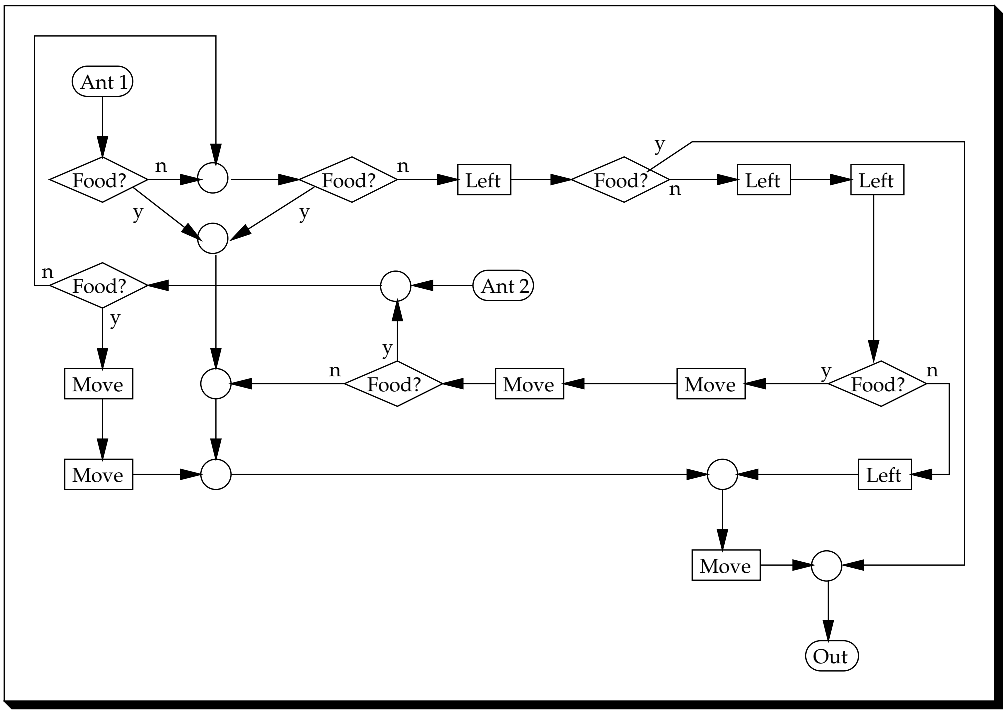


Abbildung 11.11: Zwei Ameisen benutzen die Knoten eines Graphen für unterschiedliche Zwecke

hintereinander liegenden *Move*-Anweisungen wird zusätzlich von beiden Ameisen noch an einer weiteren Stelle des Algorithmus zur schnelleren Fortbewegung benutzt.

An diesem Beispiel ist zu sehen, dass GP bereits bei kleinen Graphen den gemeinsamen Teilgraphen auf die unterschiedlichsten Weisen nutzt. Teile können mehrmals benutzt oder an unterschiedlichen Stellen des Grundschemas eingesetzt werden. Des Weiteren kann der Einstieg der beiden Ameisen in den Grundalgorithmus an unterschiedlichen Stellen des Graphen erfolgen. Diese Beobachtungen ziehen sich durch alle analysierten Individuen.

Es zeigt sich somit, dass GP in der Lage ist, innerhalb der vorgegebenen Randparameter – wie etwa der Graphgröße – für beide Ameisen individuelle Algorithmen zu erzeugen, die gemeinsame Teilgraphen auf unterschiedlichste Weise benutzen können.

11.1.5 Generalisierung

In diesem Abschnitt sollen die Möglichkeiten von GP untersucht werden, eine generalisierende Lösung für das *Artificial Ant*-Problem mit zwei Ameisen zu finden. Hierzu ist zuerst zu klären, was unter einer *generalisierenden Lösung* zu verstehen ist.

In den vorigen Abschnitten hat sich bereits gezeigt, dass bei der vorgegebenen Problemstellung sehr spezielle Algorithmen erzeugt werden, die besondere Eigenschaften des *Santa Fé*-Pfades ausnutzen. Es wird daher zusätzlich untersucht, ob eine Modifizierung der

Aufgabenstellung zu allgemeineren Algorithmen führen kann.

Generalisierung bezeichnet die Möglichkeit, dass ein GP-Individuum, welches mit einem bestimmten Testdatensatz trainiert wurde, auch auf anderen Datensätzen gute Lösungen liefert. Für das *Artificial Ant*-Problem bedeutet dies somit, dass die Algorithmen, die mit Hilfe des *Santa Fé*-Pfades erzeugt wurden, in der Lage sind, andere Pfade zu verfolgen.

Der *Santa Fé*-Pfad ist nur eine Möglichkeit, Nahrungsstücke auf einem zweidimensionalen Feld zu verteilen. Die allgemeinste Variante wäre es, j Nahrungsstücke auf einem $m \times n$ Felder großen Areal zufällig zu verteilen. Zwischen dem einfachen Ablaufen des *Santa Fé*-Pfades und dem Finden aller zufällig verteilten Nahrungsstücke, muss eine Grenze festgelegt werden, bis zu der ein *generalisierendes GP-Individuum* alle Nahrungsstücke finden muss. Abbildung 11.12 gibt hierzu mehrere Beispiele.

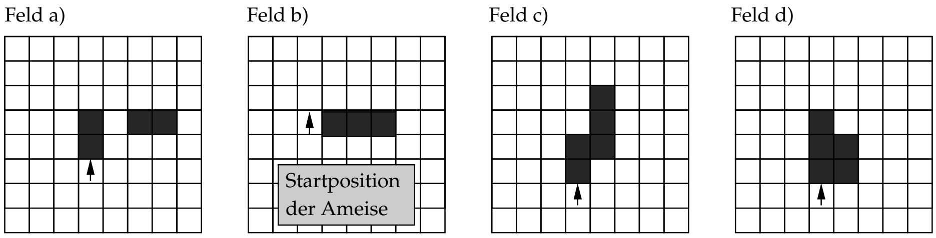


Abbildung 11.12: Verschiedene Positionierungen von Nahrungsstücken, die so nicht im *Santa Fé*-Pfad vorkommen

Alle hier gezeigten Pfadfragmente sind im *Santa Fé*-Pfad in dieser Form nicht vorhanden.

- Damit im ersten Beispiel alle vier Nahrungsstücke gefunden werden können, müsste die Ameise in der Lage sein, auch das übernächste Feld zur Rechten kontrollieren zu können. Dies kommt beim *Santa Fé*-Pfad nicht vor und würde auch vom Grundalgorithmus aus Abbildung 11.5 nicht berücksichtigt.

- Im zweiten Beispiel befindet sich das erste Nahrungsstück beim Start der Ameise seitlich von dieser. Beim *Santa Fé*-Pfad hingegen liegt es für beide Ameisen direkt vor diesen und viele der gefundenen Lösungen nutzen diese Eigenschaft aus. Die Ameisen dieser Lösungen würden zunächst an dem Pfadanfang vorbeilaufen und, falls der Pfad noch einmal ihre Bewegungsrichtung kreuzt, an einer falschen Stelle innerhalb des Pfades mit der Nahrungsaufnahme anfangen.

- Feld c zeigt eine Kombination aus einer Rechtsdrehung, einer Vorwärtsbewegung und einer Linksdrehung. Mit dem Grundalgorithmus aus Abbildung 11.5 würde der Weg korrekt verfolgt. Im *Santa Fé*-Pfad kommt diese Kombination allerdings nicht vor. Nach einer Rechtsdrehung kann sich die Ameise immer mindestens vier Felder geradeaus bewegen, ohne an einer Biegung des Pfades vorbeizulaufen.

- Für das vierte Beispiel ist die Reihenfolge im Grundalgorithmus wichtig, in der die Ameise die Felder um sich herum auf Nahrung untersucht. Betrachtet die Ameise zuerst das Feld zu ihrer Rechten und danach erst das Feld direkt vor ihr, werden im

Beispiel alle fünf Nahrungsstücke gefunden. Wird allerdings, wie in Abbildung 11.5 vorgeben, zuerst das Feld vor der Ameise untersucht, läuft die Ameise an den beiden rechten Nahrungsstücken vorbei und kann diese – je nach weiterem Verlauf des Pfades – eventuell nicht mehr finden. Das hier vorgestellte Pfadfragment kommt im *Santa Fé*-Pfad nicht vor. Es stellt sich somit die Frage, ob die rechten Nahrungsstücke von einem als *generalisierend* bezeichneten Algorithmus für das *Artificial Ant*-Problem gefunden werden müssen. In diesem Fall wäre der *Santa Fé*-Pfad als Trainingspfad ungeeignet, da keine Eigenschaften gelernt werden können, die in den Testdaten nicht vorkommen.

An diesen Beispielen ist zu erkennen, dass der Begriff der Generalisierung beim *Artificial Ant*-Problem – je nach Sichtweise – unterschiedlich ausgelegt werden kann. Wenn ein *generalisierendes GP-Individuum* auch zufällig verteilte Nahrungsstücke auf einem Feld unbekannter Größe finden sollte, müsste das GP-System einen Algorithmus ermitteln, der das komplette Feld systematisch absucht. Der Pfad wäre hierbei unabhängig von der Verteilung der Nahrungsstücke. Dies lässt sich allerdings nicht mit der Aufgabenstellung des *Santa Fé*-Pfades in Einklang bringen, da die Anzahl der insgesamt zur Verfügung stehenden *Move-*, *Left-* und *Right*-Anweisungen für eine solche Strategie nicht ausreichend ist.

Wenn unter Generalisierung hingegen verstanden wird, dass der gefundene Algorithmus bestimmte Pfadfragmente korrekt abläuft, müssen diese alle im Testpfad vorhanden sein. Die Generalisierung bestünde dann darin, dass die Pfadfragmente in einer beliebigen Reihenfolge auftreten können.

Im Fall des *Santa Fé*-Pfades könnte der Grundalgorithmus aus Abbildung 11.5 als generalisierende Lösung verstanden werden. Da allerdings das dritte Pfadfragment aus Abbildung 11.12 nicht im Pfad vorkommt, enthalten viele der gefundenen Lösungen mehrere *Move*-Befehle hintereinander, wodurch bei der gewählten Fitnessfunktion die Fitness besser wird. Die gewünschte, generalisierende Lösung hat somit nicht die beste Fitness und wird entsprechend selten auftreten.

Zum Erhalt eines GP-Individuums, das den Grundalgorithmus für beide Ameisen benutzt, muss daher sowohl die Testdatenmenge als auch die Fitnessfunktion genau an die gewünschte Lösung angepasst werden. Im folgenden Beispiel wird deutlich, dass eine Änderung der Fitnessfunktion unter Beibehaltung des *Santa Fé*-Pfades nicht unbedingt zu mehr GP-Individuen mit dem Algorithmus aus Abbildung 11.5 führen muss.

Beispiel 11.1
Viele der gefundenen Lösungen benutzen eine Variante des Grundalgorithmus aus Abbildung 11.5. Die zwei Hauptunterschiede zu diesem sind zum einen die Verwendung mehrerer Move-Befehle hintereinander, zum anderen die verschobenen Einstiegspunkte in den Algorithmus. Da beide Ameisen beim Santa Fé-Pfad direkt vor einem Nahrungsstück starten, können beide bei allen drei Food-Knoten des Algorithmus in diesen einsteigen. Es ist sogar möglich, dass der Move-Knoten der erste Befehl ist, den die Ameisen bearbeiten.

In diesem Beispiel soll nur durch eine Veränderung der Fitnessfunktion erreicht werden, dass die Ameisen vor der ersten Food-Anweisung in den Algorithmus einsteigen. Hierzu geht die Blickrichtung einer Ameise in die Fitnessbewertung ein:

- *Wenn eine Ameise bei einem Durchlauf durch den Algorithmus kein Nahrungsstück aufgenommen hat und beim Erreichen des Out-Knotens in eine andere Richtung zeigt als bei ihrem Startknoten, wird einmalig ein Strafterm von 4.1 zur restlichen Fitness hinzuaddiert.*

Die Wirkungsweise dieses Strafterms soll am Beispiel des Algorithmus aus Abbildung 11.13 verdeutlicht werden. Der Anfang des Grundalgorithmus liegt bei diesem Individuum beim markierten Food-Knoten. Die diesem vorgelagerten Food-, Right- und Move- Knoten sind Teil der vorigen Iteration des Grundalgorithmus.

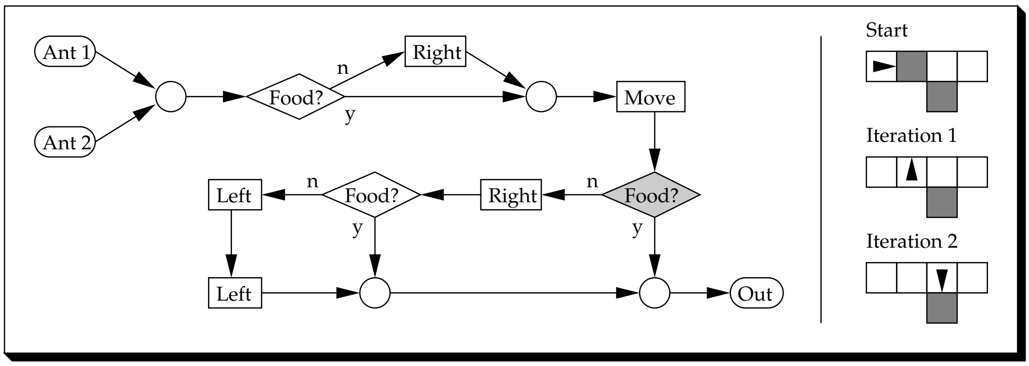


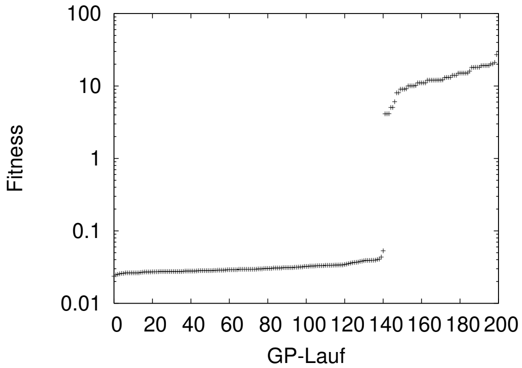
Abbildung 11.13: Ein Graph, bei dem der Einstieg in den Grundalgorithmus verschoben ist.

Solange die Ameise in jedem Algorithmendurchlauf ein Nahrungsstück aufsammelt, wirkt sich der neue Strafterm nicht auf die Fitness aus. Sobald eine Ameise allerdings die Richtung ändert und danach kein Nahrungsstück aufnimmt, verletzt das Individuum die neue Bedingung.

Wenn bei der Ausführung des Algorithmus die mit Start bezeichnete Stellung auftritt, wird sich die Ameise zu Beginn des Algorithmus – aufgrund der vorhandenen Nahrung – mittels Move vorwärts bewegen. Da vor ihr kein weiteres Nahrungsstück liegt, dreht sich die Ameise zunächst nach links und dann nach rechts und erreicht den Out-Knoten in der Stellung Iteration 1. Die Ameise zeigt nun zwar in eine andere Richtung als zu Beginn dieses Durchlaufs, konnte aber ein Nahrungsstück aufnehmen. Die Fitness wird somit noch nicht durch den Strafterm beeinflusst.

Im nächsten Durchlauf dreht sich die Ameise zunächst nach rechts und geht einen Schritt vorwärts. Die Ausführung des Algorithmus befindet sich nun am markierten Food-Knoten und somit am eigentlichen Anfang des Grundalgorithmus. Da zur Rechten jetzt ein Nahrungsstück liegt, dreht sich die Ameise in diese Richtung. Als nächstes erreicht der Programmfluss den Out-Knoten, es ergibt sich die Stellung Iteration 2. Da der Move-Befehl zur Nahrungsaufnahme erst zu Beginn der nächsten Iteration folgt, zeigt die Ameise in eine andere Richtung als am Anfang der Iteration, hat nun aber keine Nahrung aufgenommen. Somit ist die Bedingung für den Strafterm erfüllt und die Fitness des Individuums verschlechtert sich entsprechend.

Abgesehen von der geänderten Fitnessfunktion, wurden alle Parameter aus Abschnitt 11.1.1 übernommen. Es wurden wiederum 200 Läufe durchgeführt. Die Quantität der Lösungen hat sich hierbei nur marginal verändert. Wurden bei der ursprünglichen Problemstellung aus Abschnitt 11.1.1 in 140 Läufen Lösungsalgorithmen gefunden, so sind es mit der geänderten Fitnessfunktion 141 erfolgreiche Läufe.



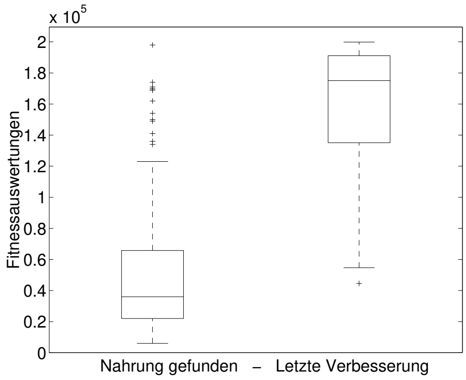


Abbildung 11.14: Die Fitnesswerte der besten Individuen der 200 Läufe und die beiden Phasen der Evolution bei zwei Ameisen mit veränderter Fitnessfunktion

In Abbildung 11.14 sind die Ergebnisse der Läufe analog zu den Abbildungen 11.2 und 11.4 dargestellt. Das linke Diagramm zeigt die Fitnesswerte des jeweils besten Individuums der 200 GP-Läufe. Die Unterschiede zum Problem mit der ursprünglichen Fitnessfunktion (Abb. 11.2) sind vernachlässigbar gering. Auch das Boxplot-Diagramm entspricht zu weiten Teilen dem aus Abbildung 11.4. Der linke Boxplot gibt wiederum an, nach wie vielen Fitnessauswertungen zum ersten Mal ein Individuum alle Nahrungsstücke aufgenommen hat. Das rechte zeigt den Zeitpunkt, zu dem das beste Individuum eines Laufs entstanden ist. In beiden Boxplots sind die 141 erfolgreichen Läufe eingegangen.

Die Unterschiede der Ergebnisse zu denen mit ursprünglicher Fitnessfunktion werden erst deutlich, wenn die erzeugten GP-Individuen im Einzelnen betrachtet werden. Hierzu wurden die Algorithmen von einem Viertel aller gefundenen Lösungen untersucht.

Aus einem Fitnesswert unter eins folgt zwangsläufig, dass alle Lösungsalgorithmen einen Weg gefunden haben, den neuen Strafterm der Fitnessfunktion zu umgehen. Die ursprüngliche Intention, dies durch einen korrekten Einstiegspunkt in den Algorithmus zu realisieren, kam allerdings nur in einem Teil der Lösungen vor.

Bei den 35 untersuchten Individuen gab es insgesamt sechs Algorithmen, die – abgesehen von Sequenzen aus mehreren Move*-Knoten – dem Grundalgorithmus aus Abbildung 11.5 genau entsprachen. Des Weiteren gibt es zusätzlich zwölf Algorithmen, die bei dem im Grundalgorithmus am Ende des Graphen stehenden* Move*-Knoten beginnen. Dies setzt zwar für den untersuchten Pfad voraus, dass die Ameise das erste Nahrungsstück direkt vor sich hat, entspricht aber ansonsten einer Lösung, die bei dem Hinzufügen des Strafterms zu erwarten war.*

Bei den restlichen vierzehn Lösungen liegt der Einstiegspunkt in den Grundalgorithmus jedoch weiterhin an einer falschen Stelle, also zum Beispiel bei der 180-Grad-Drehung der Ameise. Die Vermeidung des Strafterms erfolgt in all diesen Fällen durch eine Schleife innerhalb des Algorithmus, die kurz vor dem Out-Knoten wieder in die Nähe des Algorithmenbeginns verzweigt. Abbildung 11.15 gibt ein Beispiel hierfür. Die Schleife, die sonst durch die wiederholte Ausführung des Graphen vorgegeben ist, wird somit explizit in den Algorithmus aufgenommen.

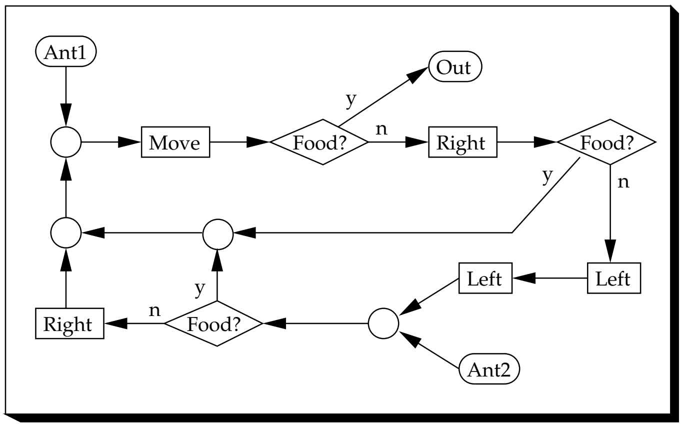


Abbildung 11.15: Ein Graph, bei dem der Einstieg in den Grundalgorithmus verschoben ist.

Bei dem abgebildeten Algorithmus handelt es sich um eine von Introns und zusätzlichen Move-Knoten bereinigte Darstellung einer Lösung. Insgesamt wurden drei Move-, zwei Food- und zwei Merge-Knoten weggelassen. Der Unterschied zum Grundalgorithmus liegt in den Einstiegspunkten der beiden Ameisen und in dem im Algorithmus explizit vorhandenen Zyklus. Beim Folgen des Santa-Fé-Pfades entsteht für keine der Ameisen eine Situation, in der der Ausgangsknoten erreicht würde und gleichzeitig noch keine Nahrung aufgenommen wurde. Ab dem zweiten Durchlauf nehmen beide Ameisen beim Erreichen des Move-Knotens automatisch Nahrung auf, da beim Verlassen des Graphen über den Out-Knoten sichergestellt ist, dass sich auf dem folgenden Feld ein Nahrungsstück befindet.

Am Beispiel wird deutlich, dass sowohl die Testdaten als auch die Fitnessfunktion im Hinblick auf eine Generalisierung genauestens ausgewählt werden müssen. Denn GP-Systeme tendieren zu Lösungen, die Aufgrund der Wahl von GP-Operatoren, Fitnessfunktion und Testdaten wahrscheinlich sind und nicht zu den Lösungen, die der Benutzer zwar gerne erhalten möchte, die er aber in den Testdaten nicht genau genug spezifiziert hat.

Um eine Generalisierung für das *Artificial Ant* Problem des Beispiels 11.1 zu forcieren, könnte ein zweiter Testpfad herangezogen werden. Hier wäre wiederum der *Santa Fé*-Pfad denkbar, allerdings würden die Ameisen zu Beginn in eine andere Richtung zeigen, so dass sie sich vor der ersten Nahrungsaufnahme erst drehen müssten. Dies würde die Wahrscheinlichkeit erhöhen wird, dass beide Ameisen den Grundalgorithmus verwenden und der Einstiegspunkt in diesen korrekt gewählt wird.

Die Kombination mehrerer *Move*-Knoten hintereinander könnte vermieden werden, indem deren Vorkommen durch einen Strafterm in die Fitnessfunktion einfließt. Alternativ können weitere kurze Pfade als Testdatensatz benutzt werden, bei denen die Ameisen *Rechts/Links*-Kombinationen ablaufen müssen.

Zusammenfassend ergeben sich folgende Punkte:

- Generalisierung ist keine Selbstverständlichkeit, die von einem GP-System ohne weiteres Zutun erwartet werden kann.
- Wenn bekannt ist, was eine generalisierende Lösung von einer nicht-generalisierenden unterscheidet, sollte dieses Wissen in Fitnessfunktion und Testdatensätze einfließen.
- Die beste Umschreibung für Generalisierung beim *Artificial Ant*-Problem ist das Ablaufen von Pfaden, die sich aus beliebig hintereinander angeordneten Pfadfragmenten ergeben. Diese Fragmente müssen anhand der Testdaten gelernt werden.

11.2 Parallele Bearbeitung von Bitvektoren

In diesem Abschnitt soll untersucht werden, ob GP in der Lage ist, bei parallelen Algorithmen einerseits Kontrollstrukturen zu evolvieren, andererseits Arbeit paritätisch auf mehrere Threads zu verteilen.

11.2.1 Problembeschreibung

Als Testproblem wurde die parallele Bearbeitung eines Bitvektors durch zwei Threads gewählt. Aufgabe des Algorithmus ist es, alle Bits, die den Wert 0 repräsentieren, auf 1 zu setzen.

Für beide Threads wurde ein Befehlssatz verwendet, der eine einfachen Registermaschine darstellt. Beide Prozessoren verfügen über einen eigenen Akkumulator, in dem die Berechnungen stattfinden sowie über ein eigenes Hilfsregister *local*. Zusätzlich haben beide Threads Zugriff auf ein gemeinsames Hilfsregister *shared*, auf ein konstantes Register *size*, das die Länge des Bitvektors enthält, auf Konstanten sowie auf den Bitvektor. Alle Register sind für ganze Zahlen ausgelegt.

Rechenoperationen werden immer im Akkumulator ausgeführt und benutzen zusätzlich ein Hilfsregister. Es stehen die folgenden Funktionen zur Verfügung: Addition, Subtraktion, Multiplikation, Laden des Wertes eines Hilfsregister in den Akkumulator, Schreiben des Akkumulators in ein (beschreibbares) Hilfsregister, Vergleich des Akkumulators mit einem Hilfsregister auf *kleiner als*, Vergleich des über den Akkumulator indizierten Elements des Bitvektors auf 0 sowie ein Bitflip-Befehl für ein Bitvektorelement.

Die Auswahl des für die nächsten Befehle zu benutzenden Hilfsregisters erfolgt über die Auswahlbefehle *Local*, *Shared*, *Size* und *Const*. Alle folgenden Rechenoperationen werden –

bis zum nächsten dieser Befehle – mit dem entsprechenden Hilfsregister ausgeführt. Soll ein Wert in ein Hilfsregister geschrieben werden, bei dem es sich um eine Konstante oder die Länge des Bitvektors handelt, wird statt dessen in das zuletzt ausgewählte beschreibbare Hilfsregister geschrieben.

Beim Start des Algorithmus sind Akkumulator, *Local*- und *Shared*-Register mit 0 initialisiert und das Hilfsregister *Local* ist für die folgenden Rechenoperationen aktiv. Für die Graphen wird eine algorithmische Semantik verwendet.

Als Testdatensatz wurden lediglich zwei komplementäre Bitvektoren der Länge 11 verwendet, $bv_1 = \{00101110010\}$ und $bv_2 = \{11010001101\}$.

Die Fitnessfunktion ist multikriteriell. Hauptkriterium ist, dass nach der Ausführung des Algorithmus der Bitvektor nur aus Einsen besteht. Weiter soll sichergestellt werden, dass beim Zugriff auf Elemente des Bitvektors mit dem Wert des Akkumulators als Index keine Werte außerhalb des gültigen Bereichs verwendet werden. Es soll also nicht auf das dreizehnte Element des Vektors zugegriffen werden, obwohl dieser nur aus elf Bits besteht.

Ein zweites, schwächer gewichtetes Kriterium besteht darin, dass alle Elemente des Bitvektors nur einmal betrachtet werden sollen. Ziel des Algorithmus muss es sein, beide Threads so aufeinander abzustimmen, dass sie sich die Arbeit teilen und überflüssige Zugriffe auf den Bitvektor vermeiden. Hierzu dient als drittes Kriterium der Fitnessfunktion ein Term, der die Anzahl der Feldelemente in Relation setzt, die beide Threads bearbeitet haben. Haben beide Threads jeweils die Hälfte der Bits bearbeitet, liegt der Wert bei null,[1] wurden alle Bits von einem Thread bearbeitet, ist der Wert gleich eins.

Für eine formale Darstellung des Problems ergibt sich somit eine Parametermenge, die aus folgenden Komponenten besteht: dem Akkumulator, den beschreibbaren Hilfsregistern, einem der Bitvektoren, Verweisen auf das aktive Hilfsregister und auf das aktive beschreibbare Hilfsregister, der Anzahl der Zugriffe auf den Bitvektor, der Anzahl der korrekten Bitflips durch die beiden Threads, der Anzahl der Verletzungen der Bitvektorgrenzen sowie der Konstanten, die durch den letzten Konstantenknoten repräsentiert wurde.

$P = (bits, accu, local, shared, activereg, activewreg, bitsaccess, okflips, idxfault, cnst)$

mit

$$\begin{aligned}
bits &\in \{bv_1, bv_2\}\\
accu, local, okflips &= (0,0)\\
activereg, activewreg &= \text{‘}local\text{'}\\
shared, bitsaccess, idxfault, cnst &= 0
\end{aligned}$$

Die Menge der Grundfunktionen sieht dabei wie folgt aus:

[1] Der Wert ist nicht gleich null, da die Anzahl der zu flippenden Bits im gewählten Beispiel ungerade ist.

$F = \{Local, Shared, Size, Const, Load, Store, Add, Sub, Mul, Ifzero, Ifless, Toggle, Nop\}$ mit

$$\begin{aligned}
\text{Local, Shared, Size, Load, Store} &: \ P \to P \times \{1\} \times \{1\} \times \{1\} \\
\text{Const} &: \ P \times [0, 10[\to P \times \{1\} \times \{1\} \times \{1\} \\
\text{Toggle, Add, Sub, Mul} &: \ P \to P \times \{1\} \times \{1\} \times \{1\} \\
\text{Ifzero, Ifless} &: \ P \to P \times \{1, 2\} \times \{1\} \times \{2\} \\
\text{Nop} &: \ P \to P \times \{1\} \times \{2\} \times \{1\}
\end{aligned}$$

Da die Grundfunktionsmenge sehr umfangreich ist, die Funktionen selbst aber relativ einfach, werden hier nur die näher aufgeführt, die zum Herleiten der übrigen Funktionen nötig sind. Die Definitionen beziehen sich o.B.d.A. auf den ersten Thread:

Local : $(\ldots, activereg, activewreg, \ldots) \mapsto (\ldots, \widehat{activereg}, \widehat{activewreg}, \ldots) \times 1 \times 1 \times 1$

$$\text{mit} \quad \widehat{activereg} = \text{`}local' \quad \text{und} \quad \widehat{activewreg} = \text{`}local'.$$

Const : $(\ldots, activereg, \ldots, cnst) \times var \mapsto (\ldots, \widehat{activereg}, \ldots, \widehat{cnst}) \times 1 \times 1 \times 1$

$$\text{mit} \quad \widehat{activereg} = \text{`}const' \quad \text{und} \quad \widehat{cnst} = var.$$

Store : $(\ldots, (local_1, local_2), shared, \ldots) \mapsto (\ldots, (\widehat{local_1}, local_2), \widehat{shared}, \ldots) \times 1 \times 1 \times 1$

$$\text{mit} \begin{cases} \widehat{local_1} = accu_1 \text{ und } \widehat{shared} = shared & \text{falls } activewreg = \text{`}local' \\ \widehat{local_1} = local_1 \text{ und } \widehat{shared} = accu_1 & \text{falls } activewreg = \text{`}shared' \end{cases}$$

Add : $(\ldots, (accu_1, accu_2), \ldots) \mapsto (\ldots, (\widehat{accu_1}, accu_2), \ldots) \times 1 \times 1 \times 1$

$$\widehat{accu_1} = \begin{cases} accu_1 + local_1 & \text{falls } activereg = \text{`}local' \\ accu_1 + shared & \text{falls } activereg = \text{`}shared' \\ accu_1 + sizeof(bitvec) & \text{falls } activereg = \text{`}size' \\ accu_1 + cnst & \text{falls } activereg = \text{`}const' \end{cases}$$

Toggle : $(bits, \ldots, bitsaccess, (okflips_1, okflips_2), idxfault, \ldots) \mapsto$

$$(\widehat{bits}, \ldots, \widehat{bitsaccess}, (\widehat{okflips}_1, okflips_2), \widehat{idxfault}, \ldots) \times 1 \times 1 \times 1$$

mit

$$\begin{cases} \widehat{bits}[accu_1] = 1 - bits[akku_1], \quad \widehat{idxfault} = idxfault & \text{falls } accu_1 \in [0, sizeof(bits)[\\ \widehat{bitsaccess} = bitsaccess + 1 & '' \\ \widehat{bits} = bits, \widehat{bitsaccess} = bitsaccess, \widehat{idxfault} = idxfault + 1 & \text{sonst} \end{cases}$$

$$\begin{cases} \widehat{okflips_1} = okflips_1 + 1 & \text{falls ein Bit zum ersten Mal von 0 auf 1 flippt} \\ \widehat{okflips_1} = okflips_1 & \text{sonst} \end{cases}$$

$$\underline{\text{Ifless}}: \quad P \mapsto P \times next \times 1 \times 2$$

$$\text{mit} \quad \begin{cases} next = 1 & \text{falls } (activereg = \text{`}local\text{'} \text{ und } accu_1 < local_1) \\ & \quad \text{oder } (activereg = \text{`}shared\text{'} \text{ und } accu_1 < shared) \\ & \quad \text{oder } (activereg = \text{`}const\text{'} \text{ und } accu_1 < cnst) \\ & \quad \text{oder } (activereg = \text{`}size\text{'} \text{ und } accu_1 < sizeof(bitvec)) \\ next = 2 & sonst \end{cases}$$

$$\underline{\text{Ifzero}}: \quad (\ldots, bitsaccess, \ldots, idxfault, \ldots) \mapsto$$
$$(\ldots, \widehat{bitsaccess}, \ldots, \widehat{idxfault}, \ldots) \times next \times 1 \times 2$$

$$\text{mit} \quad \begin{cases} next = 1 & \text{falls } accu_1 \in [0, sizeof(bits)[\text{ und } bits[accu_1] = 0 \\ next = 2 & \text{sonst} \end{cases}$$

$$\text{und} \quad \begin{cases} \widehat{idxfault} = idxfault & \text{falls } accu_1 \in [0, sizeof(bits)[\\ \widehat{bitsaccess} = bitsaccess + 1 & \\ \widehat{idxfault} = idxfault + 1 & \text{sonst} \\ \widehat{bitsaccess} = bitsaccess & \end{cases}$$

Zur Berechnung der Fitness eines Individuums werden für beide Bitvektoren die Fitnesswerte fit_1 und fit_2 getrennt voneinander berechnet und danach addiert. Hierzu wird jeweils der entsprechende Bitvektor bv_1 oder bv_2 als Komponente in die Parametermenge P eingesetzt. Anschließend wird mit der folgenden Formel die Einzelfitness für den Bitvektor berechnet:

$$fit_i = 1000 \times zeroes(bits) + 300 \times idxfault + (bitsaccess - sizeof(bits)) + proportion(okflips)$$

Hierbei ist $zeroes(bits)$ die Anzahl der Nullen im Bitvektor nach Terminierung des Algorithmus, $sizeof(bits)$ die Länge des Bitvektors und $proportion(okflips)$ das Verhältnis der von beiden Threads korrekt gesetzten Bits:

$$proportion((okflips_1, okflips_2)) = \begin{cases} \dfrac{|okflips_1 - okflips_2|}{okflips_1 + okflips_2} \times 0.9999 & \text{falls } okflips_1 + okflips_2 > 0 \\ 0 & \text{sonst} \end{cases}$$

Die beste Fitness, die sich für das Problem ergeben kann, ist 0.19998, da beim zweiten Bitvektor eine ungerade Anzahl von Bits geflippt werden muss und somit der Proportionalterm nicht Null werden kann.

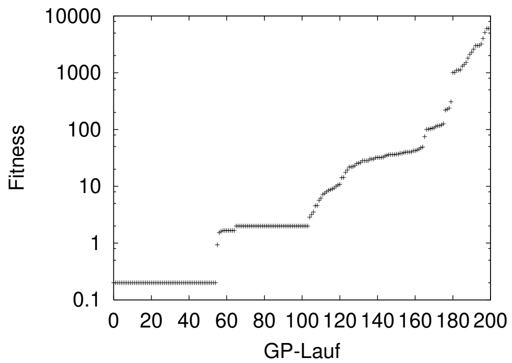
Die weiteren Parameter des Testproblems sind in Tabelle 11.3 aufgeführt. Wie auch die Experimente für das *Artificial Ant*-Problem mit zwei Ameisen, wurden die Testläufe ohne kontextsensitive Operatoren (Abschnitt 7.4) und Demes (Abschnitt 9.1) durchgeführt. Es wurde eine Kombination aus dem *Idp*-Verfahren (Abschnitt 7.2.4) und Schrittweitensteuerung (Abschnitt 8.2) eingesetzt.

Parameter	Wert
Populationsgröße	100
Fitnessauswertungen	200000
Turniergröße	4/2
Graphgröße	20
Crossover	*random*
Läufe	200
Besonderheiten	Schrittweitensteuerung adaptive Operatorwahrscheinlichkeiten (*Idp*) **keine kontextsensitiven Operatoren** zyklische Graphen

Tabelle 11.3: GP-Variablen für das *Bitvektor*-Problem

11.2.2 Ergebnisse

Abbildung 11.16 zeigt die Ergebnisse der 200 GP-Läufe. Im linken Diagramm sind die Fitnesswerte der jeweils besten Individuen der einzelnen Läufe abgebildet. Die Läufe wurden bezüglich dieser Fitness sortiert.



Gruppe	Anzahl	Fitness
1	55	0.19998
2	10	0.93324 1.6665
3	39	1.9998
4	62	2.86658 74.9999
5	14	100.59994 308.26654
6	20	1009.39996 6002.9998

Abbildung 11.16: Die Fitnesswerte des jeweils besten Individuums aller 200 GP-Läufe beim *Bitvektor*-Problem im Einzelnen und in Gruppen eingeteilt.

Die Menge der GP-Läufe lässt sich, basierend auf den Fitnesswerten des jeweils besten Individuums, in sechs Gruppen einteilen. Sie sind in der rechten Tabelle von Abbildung 11.16 dargestellt. In der mit *Fitness* überschriebenen Spalte befinden sich die Fitnesswerte des besten und schlechtesten Individuums einer Gruppe. Ist nur ein Wert vorhanden, haben alle Individuen dieselbe Fitness. Die erste Gruppe umfasst 55 Läufe. In diesen wurden jeweils Individuen mit der Fitness 0.19998 gefunden. Der Großteil der Algorithmen dieser Individuen funktioniert nach demselben Prinzip: Ein Thread betrachtet die geraden Indizes des Bitvektors, der andere die ungeraden. Als Abbruchkriterium wird die *Size*-Funktion benutzt. Diese Algorithmen sind somit generalisierend, sie sind ebenfalls für Bitvektoren beliebiger Länge einsetzbar. Abbildung 11.17 zeigt einen für diese Gruppe typischen Algorithmus.

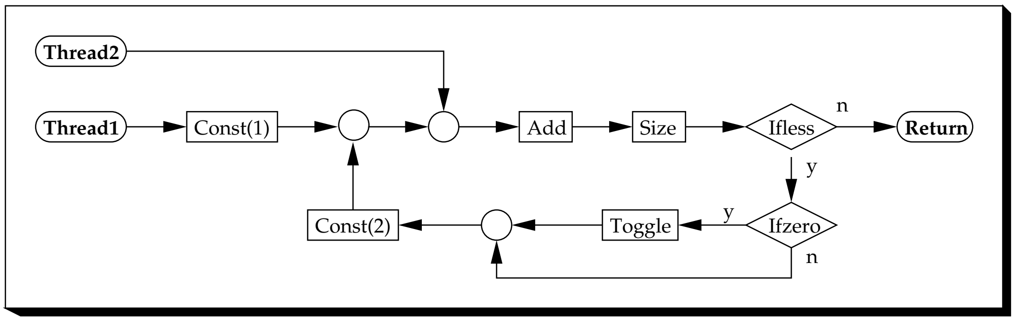


Abbildung 11.17: Ein Algorithmus für das *Bitvektor*-Problem mit zwei Threads, bei dem jeder die Hälfte der Bits untersucht.

Lediglich zwei der besten Individuen unterschieden sich im grundsätzlichen Aufbau von den Übrigen. Als Abbruchkriterium wurde nicht der *Size*-Knoten verwendet, sondern eine Addition aus mehreren *Const*-Werten. Diese Individuen sind entsprechend nur für Bitvektoren geeignet, die dieselbe Länge wie die Testvektoren besitzen.

In der dritten Gruppe befinden sich alle Individuen, die eine Fitness von 1.9998 besitzen. Diese Fitness bedeutet, dass alle Bits genau einmal betrachtet wurden und alle Nullen geflippt wurden. Allerdings wurde dies ausschließlich von einem der beiden Threads ausgeführt. Während der eine Thread alle Bits nacheinander betrachtete und einen Vergleich des aktuellen Index mit der *Size*-Funktion als Abbruchkriterium benutzte, terminierte der zweite umgehend[1].

In fünf Fällen wurde das Abbruchkriterium über die Addition von Konstanten an Stelle der *Size*-Funktion erreicht. Während diese Algorithmen somit auf eine bestimmte Länge des Bitvektors festgelegt sind, können die Restlichen insofern generalisieren, als dass sie ohne Änderung für Bitvektoren beliebiger Länge anwendbar sind. Allerdings konnte die Arbeit nicht gleichmäßig auf zwei Threads verteilt werden. Somit ist bei diesen Algorithmen die Parallelisierung fehlgeschlagen. Abbildung 11.18 zeigt einen typischen Graphen aus dieser Gruppe.

In den Gruppen zwei und vier befinden sich Individuen, deren Algorithmus das Testproblem zwar löst, die dies aber durch teilweise – aus algorithmischer Sicht – ‚unschöne' Methoden erreichen. Im besten Fall bearbeitet ein Thread den kompletten Bitvektor abzüglich eines Elementes mittels einer Schleife, die ein Element zu früh terminiert. Dieses Bit wird dann vom zweiten Thread betrachtet, der nach diesem einen Schritt beendet wird. Im ungünstigen Fall wird ein Register, das als Index für den Bitvektor dient, in einer Schleife um verschiedene *Const*-Knoten-Werte variiert. Hierbei liegt kein systematisches Vorgehen vor und Elemente des Bitvektors werden oft mehrfach betrachtet. Auch kann die ergebende Reihe von Indizes von der aktuellen Belegung des Bitvektors abhängen, so dass diese Individuen bei anderen Bitvektoren gleicher Länge das Problem nicht lösen könnten.

[1] In zwei Fällen endete einer der Threads in einer Endlosschleife.

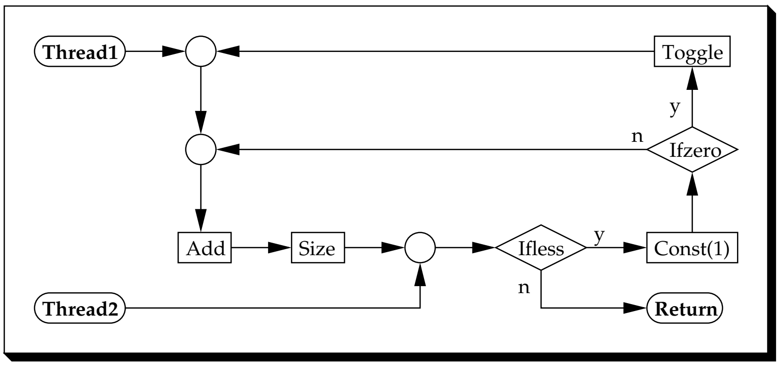


Abbildung 11.18: Ein typischer Algorithmus für Individuen der dritten Gruppe. Durch die Initialisierung des Akkus und der Register mit dem Wert Null terminiert der zweite Thread umgehend.

Die Individuen aus den Gruppen fünf und sechs lösen das Testproblem entweder überhaupt nicht oder versuchen während des Ablaufs auf nicht existierende Elemente des Bitvektors[1] zuzugreifen.

Zusammenfassend ergeben sich folgende Resultate:

1. GP ist auch bei diesem Testproblem in der Lage parallele Algorithmen zu evolvieren. In 53 Läufen – also in 26.5 Prozent aller Fälle – wurden Lösungen evolviert, bei denen beide Threads jeweils die Hälfte des Bitvektors betrachten.

2. Die Lösungen können bezüglich ihrer Generalisierungseigenschaften in drei Klassen unterteilt werden. Die erste Klasse enthält alle Individuen, deren Algorithmus auf sämtliche Bitvektoren beliebiger Länge angewendet werden kann. Dies sind 53 Individuen aus Gruppe eins, fünf der zweiten Gruppe und 34 aus Gruppe drei – zusammen also 46 Prozent aller besten Individuen.

   Die zweite Klasse besteht aus den Individuen, deren Algorithmen beliebige Bitvektoren einer bestimmten Länge bearbeiten können. Diese schwache Form der Generalisierung gilt für die zwei verbliebenen Individuen der ersten Gruppe, für ein Individuum der zweiten und für vier Individuen der dritten Gruppe – insgesamt also 3.5 Prozent.

   Alle übrigen Individuen repräsentieren entweder keine Lösung oder nur einen Algorithmus, der für die beiden getesteten Bitvektoren das korrekte Ergebnis liefert, jedoch im Allgemeinen bei anderen Vektoren versagt.

3. Die Individuen der ersten beiden Generalisierungsklassen enthalten innerhalb der Graphen Kontrollstrukturen, wie sie sonst in imperativen Programmiersprachen üblich sind. Die in Abbildung 11.17 zu sehende Knotenkombination *Ifzero/Toggle/Merge* entspricht einer *If/Then*-Bedingung und die Kombination aus der

[1] Zum Beispiel Elemente mit negativem Feldindex

Addition eines festen Wertes mit folgender Vergleichsoperation entspricht einer Schleife mit festem Inkrement. Des Weiteren ist zu sehen, dass diese beiden Strukturen ineinander geschachtelt sind. Bei nicht generalisierenden Individuen ist diese Strukturierung in den meisten Fällen nicht vorhanden. Sollten diese Algorithmen direkt in einer Hochsprache wie *C* implementiert werden, so müssten *Goto*-Konstrukte eingesetzt werden. Die Programme würden entsprechend unlesbar.

Speziell aus dem dritten Punkt ergibt sich für zukünftige Untersuchungen die Frage, ob es einen Zusammenhang zwischen generalisierenden Algorithmen und der Evolution von Kontrollstrukturen mittels GP gibt. Ist es möglich, durch das Erzwingen von bekannten Kontrollstrukturen in den Individuengraphen die Wahrscheinlichkeit von generalisierenden Algorithmen zu erhöhen? Sind umgekehrt gut strukturierte Lösungen zu erwarten, wenn Möglichkeiten zur Verfügung stehen, durch die mit hoher Wahrscheinlichkeit generalisierende Algorithmen evolviert werden? Hieraus ergibt sich direkt die Frage, ob in diesem Fall ein GP-System nicht nur zum Lösen von Optimierungs- oder Klassifikationsproblemen herangezogen, sondern auch in der Softwareentwicklung eingesetzt werden kann.

Kapitel 12

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde das GP-System *GGP* entwickelt, das den Genotyp eines Individuums als Graph modelliert. Hierdurch unterscheidet es sich von anderen Ansätzen, bei denen der Genotyp ein Baum [57] oder eine lineare Sequenz von Befehlen [7] ist. Die Motivation für die Arbeit war die geplante Entwicklung eines GP-Systems zur Modellierung und Regelung dynamischer Systeme der Regelungstechnik. Schaltungen der Regelungstechnik lassen sich auf natürliche Weise als Graphen darstellen. Somit lag es nahe, den gewohnten Baumansatz durch einen graphbasierten Ansatz zu ersetzen. Bis zu diesem Zeitpunkt war es üblich, Graphen in Bäume zu überführen [72, 61] und dabei eine Einschränkung des Suchraums in Kauf zu nehmen [72, 61]. Somit wurden immer potenzielle Lösungen ausgeschlossen.

Die neue Repräsentationsform der Individuen machte es erforderlich, neue genetische Operatoren zu entwickeln. Diese müssen die Graphen in geeigneter Weise verändern können. Sie sind in Abschnitt 12.1 zusammengefasst.

Die Fähigkeiten des neuen GP-Systems wurden anhand von mehreren Benchmarks analysiert. Durch genauere Untersuchungen der Lösungen konnten Erweiterungen des Systems entwickelt werden, die zu einer deutlichen Steigerung der Performanz führten. Abschnitt 12.2 fasst alle Ergebnisse zusammen und vergleicht sie mit Werten aus der Literatur.

Im letzten Abschnitt dieses Kapitels wird aufgezeigt, für welche Anwendungen das GP-System zukünftig verwendet werden kann. Zusätzlich werden Ideen für weitere Verbesserungen vorgestellt.

12.1 Das GP-System

Das GP-System *GGP* benutzt gerichtete Multigraphen als Genotyp der Individuen. Jedem Knoten wird hierbei eine Funktion zugeordnet. Durch diese Funktion ist die Anzahl der Eingangs- und Ausgangskanten jedes Knotens eindeutig bestimmt.

Das System unterstützt zwei unterschiedliche Arten von Semantiken für Graphen, einen funktionalen und einen algorithmischen Ansatz. Beim funktionalen repräsentieren die Kanten des Graphen die Funktionswerte, die von den Funktionen der Knoten berechnet werden. Beim algorithmischen Ansatz wird der Graph als Flussdiagramm interpretiert. Die Werte, die mit den Funktionen der Knoten berechnet werden, befinden sich in externen Variablen. Die Kanten repräsentieren den Programmfluss.

Baumbasierte GP-Systeme verwenden genetische Operatoren wie *Mutation* und *Crossover*. Diese sind für *GGP* nicht ausreichend bzw. nicht anwendbar. Aus diesem Grund mussten neue Operatoren entwickelt werden. Diese können einen Knoten variieren (*Knoten mutieren*, *Knoten einfügen*, *Knoten löschen* und *Knoten verschieben*), spezielle Teilgraphen verändern (*Pfad einfügen*, *Pfad löschen* und *Zyklus*) oder das Genom zweier Individuen miteinander rekombinieren (*randomCrossover*). Graphen im *GGP*-System müssen weitere Bedingungen erfüllen, die über die Definition eines Multigraphen hinausgehen. Der Crossover-Operator sollte diese berücksichtigen. Somit kann auf sonst nötige Reparaturmechanismen verzichtet werden.

Es wurden drei verschiedene Crossover-Operatoren entwickelt. Bei allen lassen sich Fälle konstruieren, in denen der Operator keinen geforderten Graphen erzeugen kann. Wenn jedoch zwischen Testproblemen mit zyklischen und azyklischen Graphen unterschieden wird, kann für beide Klassen ein geeigneter Operator angegeben werden (*randomCrossover* bzw. *acycCrossover*). Die Fehlerrate für diese ist jeweils so gering, dass sie für GP-Läufe vernachlässigbar ist. Der dritte Crossover-Operator (*generalCrossover*) erzielt für beide Problemklassen akzeptable, jedoch etwas schlechtere Ergebnisse.

Wenn ein neues Individuum erzeugt werden soll, wird genau ein genetischer Operator auf die Elter-Individuen angewandt. Die Wahrscheinlichkeiten für die jeweilige Auswahl dieses Operators werden beim Programmstart über Parameter festgelegt. Eine adaptive Anpassung dieser Werte während eines GP-Laufs führte zu keinen signifikant besseren Resultaten.

Große Ergebnisverbesserungen konnten durch den kontextsensitiven Einsatz der genetischen Operatoren erzielt werden: Abhängig von den Graphen der Elter-Individuen werden verschiedene Operatoren verboten und zu einem späteren Zeitpunkt des GP-Laufs wieder zugelassen.

In weiteren Versuchsreihen wurden mehrere Operatoren auf einen Graphen angewendet, bevor die neue Fitness des Individuums berechnet wurde. Zunächst wurde ein stark kausaler Zusammenhang zwischen der Anzahl hintereinander angewandter Operatoren und der Größe der Fitnessveränderung nachgewiesen. Dies galt für alle Testprobleme. Die Häufigkeit der Fitnessverbesserungen nahm jedoch überproportional ab, wenn die Anzahl der sequenziell angewandten Operatoren zu groß wurde. Der GP-Algorithmus wurde deshalb so erweitert, dass bei jeder Graphvariation zwischen ein und drei Operatoren angewandt werden. Auf diese Weise ergaben sich signifikant bessere Ergebnisse bei den Testproblemen. Eine Adaptation dieser Anzahl während eines GP-Laufs konnte nur bei einigen Benchmarks die Resultate verbessern. Der hierdurch entstehende Mehraufwand stand jedoch in keinem Verhältnis zu den (nicht signifikanten) Verbesserungen.

Des Weiteren wurde die Populationsdiversität untersucht. Es stellte sich heraus, dass bereits nach wenigen Fitnessauswertungen alle Graphen von demselben Individuum der Ursprungspopulation abstammen. Zur Diversitätserhaltung wurden Demes in das GP-System integriert. Die Anzahl der Demes kann hierbei während eines GP-Laufs verändert werden. Beim Erstellen neuer Demes werden neue Individuen erzeugt. Auf diese Weise steht der Population wieder neues genetisches Material zur Verfügung, die Diversität steigt. Durch diesen Ansatz, der als *dynamische Demes* bezeichnet wird, war insgesamt eine signifikante Verbesserung der Testergebnisse möglich.

Bei vielen Problemstellungen können Fitnessberechnungen den Großteil der Rechenzeit ausmachen, die ein GP-Lauf benötigt. In diesen Fällen ist es sinnvoll, doppelte Berechnungen von identischen Graphen zu vermeiden. Das GP-System wurde um eine entsprechende Funktionalität erweitert. Zusätzlich können isomorphe Graphen erkannt werden und brauchen somit ebenfalls nicht erneut berechnet zu werden. Bei der Untersuchung der Testprobleme ergab sich, dass – je nach Problem – zwischen 22 und 72 Prozent aller Fitnessauswertungen überflüssig sind. Gerade im Aufgabenbereich *Modellierung und Regelung dynamischer Systeme* kann somit ein großer Teil der Rechenzeit eingespart werden.

In einem abschließenden Anwendungskapitel wurden mit *GGP* parallele Algorithmen evolviert. Als Beispiel diente die Erweiterung des *Artificial Ant*-Problems um eine zweite Ameise. Der Algorithmus dieser Ameise befindet sich im selben Graphen wie der der ersten Ameise, jedoch mit einem anderen Startknoten. Die Lösungsindividuen wurden eingehend auf ihre Fähigkeit zur Generalisierung untersucht. Es ergab sich, dass durch den *Santa-Fé*-Pfad Lösungsgraphen bevorzugt werden, die der gewünschten Generalisierungsfähigkeit der Individuen widersprechen.

Zum Abschluss wurden die Ergebnisse aus dem Bereich *Modellierung und Regelung dynamischer Systeme* vorgestellt, die mit dem Vorgängersystem von *GGP* erzielt wurden.

12.2 Ergebnisse

Während die Resultate dieser Arbeit im Abschnitt 12.1 qualitativ zusammengefasst sind, werden hier die quantitativen Ergebnisse für die sechs Testprobleme aufgeführt. Die Abbildungen 12.1 bis 12.6 zeigen jeweils die Ergebnisse der wichtigsten Testreihen aus je 200 GP-Läufen. In den Diagrammen wurden nur selbst durchgeführte Testreihen berücksichtigt. Vergleiche mit anderen, aus der Literatur bekannten Ergebnissen werden gesondert aufgeführt, da die Fitnesswerte oft unter anderen Voraussetzungen erzielt wurden. Dies kann eine andere Anzahl von GP-Läufen oder eine abgewandelte Aufgabenstellung sein.

Allen Resultaten ist gemein, dass die Ergebnisse, die mit *GGP* und den zugehörigen Erweiterungen erzielt wurden, deutlich besser sind als die des baumbasierten GP-Systems.

Beim *Polynom*-Problem (Abbildung 12.1) konnte die durchschnittliche Fitness des besten Individuums eines Laufs auf unter 2 Prozent der Fitness beim baumbasierten GP-System verringert werden. Ohne die Verbesserungen der Kapitel 7 bis 9 war das graphbasierte System schlechter als baumbasierte. Dies lag zu großen Teilen daran, dass anstelle von Graphvariationen häufig Reproduktionen der Elter-Individuen durchgeführt wurden. Dies wurde mit der kontextsensitiven Operatorwahl *Std+* verhindert. Durch die Verwendung von *dynamischen Demes* konnte zusätzlich die Diversität der Populationen so erhöht werden, dass das Verlassen lokaler Optima vereinfacht wurde.

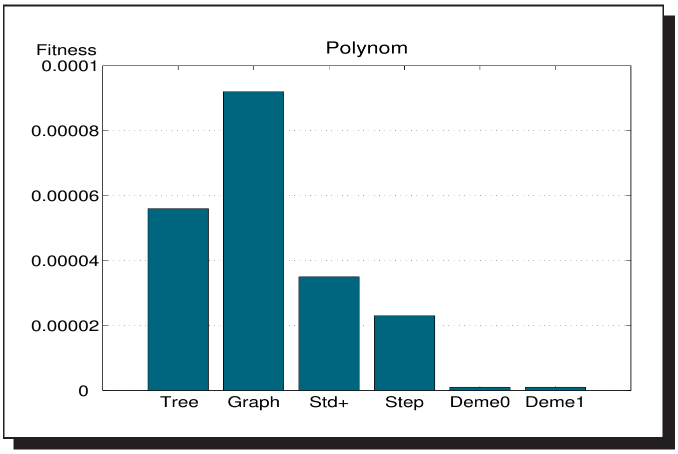


Abbildung 12.1: Die durchschnittlichen Fitnesswerte der Testreihen zum *Polynom*-Problem

In der Literatur wird die Fitness beim *Polynom*-Problem in den meisten Fällen anders berechnet als in dieser Arbeit: Die Fitness eines Individuums entspricht der Anzahl der Stützpunkte, bei denen der Wert der ermittelten Funktion um mehr als 0.01 bzw. 0.05 vom tatsächlichen Wert abweicht [77]. Oft wird sogar nur der *Effort*-Wert als Resultat einer Testreihe angegeben. Diese Fitnesseinteilung ist zu grob und wurde deshalb nicht verwendet.

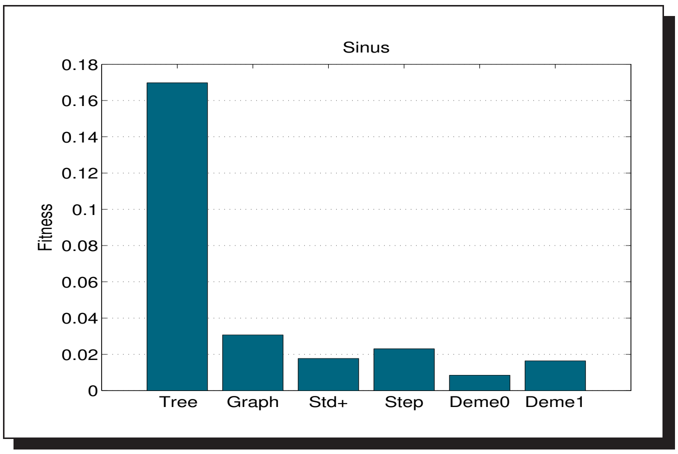


Abbildung 12.2: Die durchschnittlichen Fitnesswerte der Testreihen zum *Sinus*-Problem

Beim *Sinus*-Problem ist es nicht möglich, die gesuchte trigonometrische Funktion aus den vorgegebenen arithmetischen exakt zu berechnen. Es kann nur eine Näherungslösung evolviert werden. Dies unterscheidet die Aufgabenstellung vom *Polynom*-Problem. Das graphbasierte System ist von vornherein signifikant besser als das baumbasierte (Abbildung 12.2). Die durchschnittliche Fitness des besten Individuums bei *GGP* liegt bereits bei nur einem Fünftel der Fitness vom Baumansatz. Durch gleichzeitige Verwendung der *kontext-sensitiven Operatorwahl* und *dynamischer Demes* (*Deme0*) kann die Fitness sogar auf etwa ein zwanzigstel gesenkt werden. Bei diesem Problem ist die Ausführung mehrerer Operatoren vor einer Fitnessberechnung nicht von Nutzen: Die Variationen durch einzelne Operatoren sind bereits groß genug, um lokale Optima zu verlassen.

In der Literatur wird das Problem auf sehr unterschiedliche Weise definiert. So wird zum Beispiel, wie beim *Polynom*-Problem, die Anzahl der nicht gut angenäherten Stützpunkte als Fitness verwendet. In vielen Fällen wird der Sinus auf einem anderen Definitionsintervall betrachtet (z.B. $-\pi$ bis π, [109]). Ein Vergleich kann mit der Veröffentlichung von KANTSCHIK und BANZHAF erfolgen, wo das Problem auf *Linear Tree-GP* angewendet wurde [49]. Rechnet man die dort aufgeführten Werte so um, dass die Fitnesswerte vergleichbar sind, ergibt sich eine durchschnittliche Fitness von *0.062*. Dieser Wert ist deutlich besser als der mit Baum-GP erreichte durchschnittliche Wert, jedoch schlechter als der in dieser Arbeit erzielte.

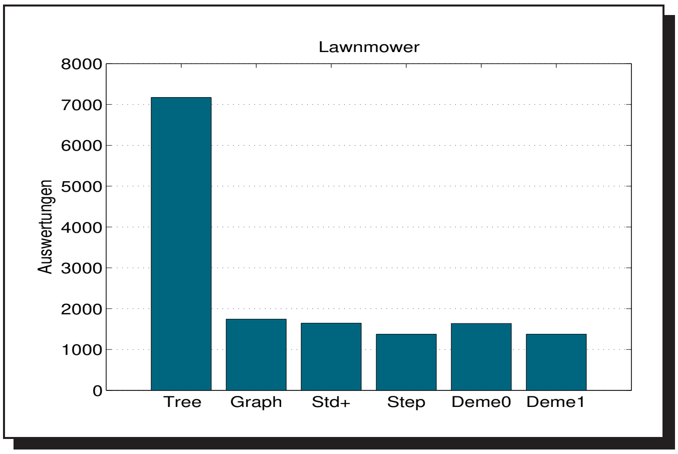


Abbildung 12.3: Die durchschnittlich benötigte Anzahl von Fitnessauswertungen der Testreihen zum *Lawnmower*-Problem

Das *Lawnmower*-Problem erweist sich für alle GP-Systeme als sehr einfach. Es können in fast allen Läufen Lösungen gefunden werden. Der eigentliche Unterschied der GP-Systeme besteht darin, wie viele Fitnessauswertungen durchschnittlich benötigt werden, bis die erste Lösung gefunden wird. Diese Zahlen werden in Abbildung 12.3 miteinander verglichen. Das baumbasierte System ist signifikant langsamer als die graphbasierten Varianten. Die Benutzung von Demes ist überflüssig, da die Lösungen bereits gefunden sind, bevor der Deme-Mechanismus in den Testläufen zum ersten Mal aktiviert werden kann.

In der Literatur werden Ergebnisse zum *Lawnmower*-Problem oft nur durch den *Effort*-Wert

Effort	GP-Typ	Quelle
100000	tree	Poli [90]
58000	stack based	Bruce [16]
11000	tree + ADF	Poli [90]
10000	tree + culture	Spector & Luke [110]
5000	parallel distributed	Poli [90]
3311	*GGP* (*Step*)	Abschnitt 8.2

Tabelle 12.1: *Effort*-Werte zum *Lawnmower*-Problem

dargestellt. Aus Gründen der Vollständigkeit sind sie in Tabelle 12.1 zusammengefasst. Wegen der statistischen Ungenauigkeit des Wertes (Abschnitt 5.4) werden sie jedoch nicht weiter kommentiert.

Das *Artificial Ant*-Problem galt für GP-Systeme bisher als sehr schwierig [66]. Dies bestätigt sich mit dem schlechten Ergebnis des baumbasierten Systems (Abbildung 12.4): Nur in 7 Prozent aller Läufe wurde eine Lösung gefunden. Der in der Literatur bis jetzt gewählte Ausweg bestand darin, das Problem zu vereinfachen. Somit erhielten auch herkömmliche GP-Systeme eine Chance, zumindest eine Variante des Problems zufriedenstellend zu lösen [65, 62].

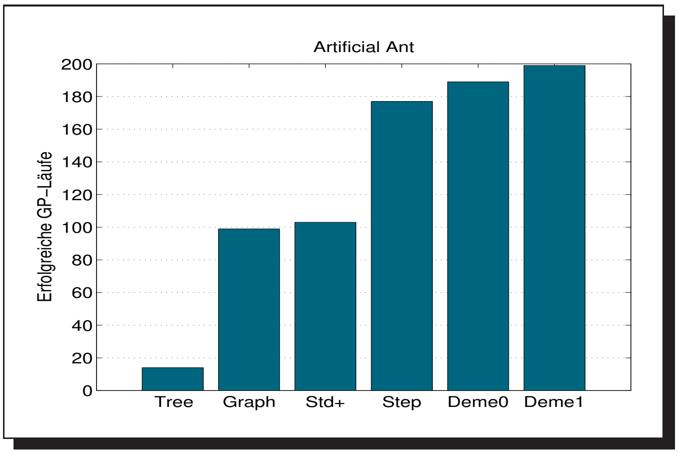


Abbildung 12.4: Die Anzahl der erfolgreichen GP-Läufe in den Testreihen zum *Artificial Ant*-Problem

Mit dem graphbasierten System *GGP* ist es gelungen, das *Artificial Ant*-Problem mit dem *Santa Fé*-Pfad ohne Änderung des Suchpfades zu lösen. Die Aufgabenstellung ist dieselbe geblieben. Lediglich die Kodierung und Bewertung der gefundenen Lösungen wurde verändert:

- Das Genom besteht aus Graphen anstelle von Bäumen.
- Durch eine Begrenzung der Graphgröße wird der Suchraum eingeschränkt.

- Die Fitnessfunktion wurde verschärft: Schnellere Ameisen werden bevorzugt.

Die Ergebnisse in Abbildung 12.4 belegen, dass mit *GGP* und den Erweiterungen aus den Kapiteln 7 bis 9 wesentlich häufiger Lösungen gefunden werden als mit baumbasierten Ansätzen.

Tabelle 12.2 enthält weitere Ergebnisse aus der Literatur.

Training Technique	Solutions	Runs
Original	35	200
Strict Hill Climbing [66]	8	50
Evolutionary Programming [20]	47	59
Scalar speed [66]	9	50
1 food ahead [66]	19	50
5 food ahead [66]	71	200
1ptXO, original fitness function [66]	8	100
Grammatical Evolution (100000 Auswertungen) [85]	56	100
GGP mit *Deme1*	**199**	**200**

Tabelle 12.2: Ergebnisse aus der Literatur für das *Artificial Ant*-Problem

Das *Two-Boxes*-Problem stellt für graphbasiertes GP ebenfalls kein Problem dar. Abbildung 12.5 zeigt, dass die Verfahren *Deme0* und *Deme1* immer eine Lösung finden konnten. Beim baumbasierten GP-System lag die Erfolgsquote hingegen bei nur 40 Prozent.

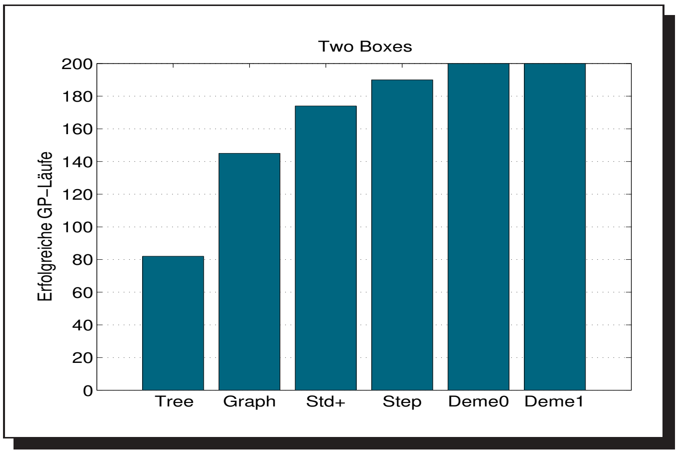


Abbildung 12.5: Die Anzahl der erfolgreichen GP-Läufe in den Testreihen zum *Two-Boxes*-Problem

Tabelle 12.3 zeigt die Ergebnisse aus zwei anderen aktuellen Veröffentlichungen. In diesen wurde versucht, das *Two-Boxes*-Problem mit neuen Methoden für lineares [40] und baumbasiertes GP [118] zu lösen. In beiden Veröffentlichungen wurden unterschiedliche Testreihen mit je 50 GP-Läufen durchgeführt. Die Spalte *Solutions* der Tabelle gibt an, wie viele der

Läufe in den Testreihen erfolgreich waren. Es zeigt sich, dass nur eine der siebzehn Testreihen mit graphbasiertem GP mithalten kann.

Publication	Solutions (in 50 Runs)
M. I. HEYWOOD & A. N. ZINCIR-HEYWOOD [40]	12, 14, 46, 4, 4, 2, 10, 2
M. D. TERRIO & M. I. HEYWOOD [118]	2, 0, 6, 18, 9, 1, 2, 0, 2
GGP mit *Deme0* (Abschnitt 9.1.3)	**50**

Tabelle 12.3: Ergebnisse aus der Literatur für das *Two-Boxes*-Problem

Beim *Klassifikations*-Problem konnte die Anzahl der durchschnittlichen Fehlklassifizierungen mit Hilfe von graphbasiertem GP auf etwa ein Drittel reduziert werden (Abbildung 12.6). Wie bereits beim *Sinus*-Problem bringt eine Anwendung der *Step*-Erweiterung keine Vorteile. Es sind die beiden einzigen Testprobleme, bei denen die Grundfunktionen lokale, reellwertigen Parameter besitzen. Der Einsatz von größeren Schrittweiten scheint somit nicht nötig zu sein, wenn im Genom der Individuen reelle Zahlen vorhanden sind, die durch den *Mutations*-Operator verändert werden können. Bei ausschließlich parameterlosen Grundfunktionen führt die Verwendung von *Step* hingegen immer zu besseren Ergebnissen.

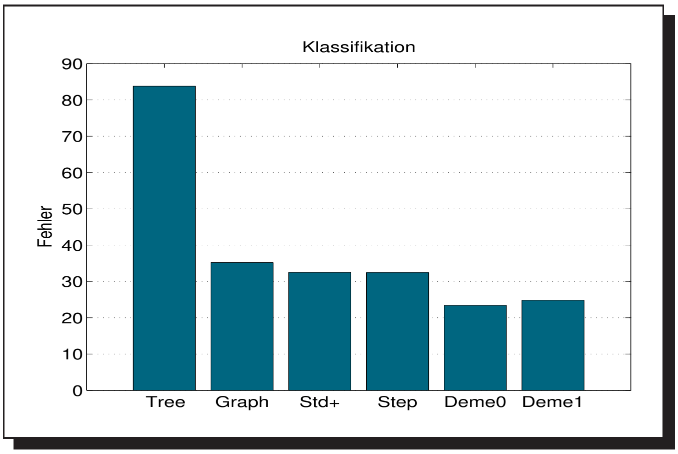


Abbildung 12.6: Die durchschnittliche Zahl von Fehlklassifikationen in den Testreihen zum *Klassifikations*-Problem

Das *Klassifikations*-Problem wurde in der Literatur bisher nur selten verwendet. Die Werte aus der Veröffentlichung von BRAMEIER und BANZHAF [14] können nicht zum Vergleich herangezogen werden, da andere Grundfunktionen verwendet wurden. Des Weiteren waren wesentlich mehr Fitnessauswertungen erlaubt. Vergleichbar sind die Ergebnisse aus [49] für *Linear Tree GP*. Für dieses Verfahren wurde umgerechnet eine durchschnittliche Zahl von 19.8 Fehlklassifizierungen ermittelt. Dieses Ergebnis ist etwas besser als das mit graphbasiertem GP erzielte – allerdings wurden nur 30 Testläufe durchgeführt.

Abschließend lässt sich festhalten, dass alle Testprobleme von der Verwendung von graphbasiertem GP gegenüber baumbasiertem profitieren. Neben den besseren Ergebnissen bezüglich der Fitness ist *GGP* in der Lage, mit sehr kleinen Graphen zu arbeiten. Dadurch enthalten Individuen nur sehr wenige Introns.

Von den zusätzlichen Erweiterungen des GP-Systems hat sich die Verwendung der *kontextsensitiven Operatorwahl* und *dynamischer Demes* in allen Fällen als sinnvoll erwiesen[1], die Adaptation von Operatorwahrscheinlichkeiten nur in Einzelfällen. Das sequenzielle Ausführen mehrerer genetischer Operatoren erlaubte größere Schrittweiten im Suchraum. Es erwies sich immer dann als sinnvoll, wenn keine der erlaubten Knotenfunktionen über eigene reelle Parameter verfügte.

12.3 Ausblick

In dieser Arbeit wurde das graphbasierte GP-System *GGP* vorgestellt. Bei den verwendeten Testproblemen erwies es sich als gute Alternative zum baumbasierten GP-Ansatz. Neben den besseren Fitnesswerten ist hier besonders hervorzuheben, dass das System mit sehr kleinen Datenstrukturen als Genotyp arbeiten kann. Hieraus ergibt sich zwangsläufig, dass Introns nur in sehr begrenztem Umfang vorkommen können.

Für die zukünftige Arbeit an und mit dem System sind zwei Richtungen möglich: Zum einen sind weitere Verbesserungen des GP-Systems selbst möglich, zum anderen kann das System nun auf praxisrelevante Probleme angewendet werden.

Systemverbesserungen

- Das System verwendet momentan eine multikriterielle Fitnessfunktion. Neben der Fitnessbewertung, die sich aus der Semantik eines Graphen ergibt, wird als zweites Kriterium die Größe des Genoms betrachtet: Je kleiner ein Graph bei gleichem Fitnesswert ist, desto besser. Bei den in dieser Arbeit verwendeten Testproblemen hat sich dieses Verfahren bewährt. Es konnten immer gute Lösungen gefunden werden.

  Es kann jedoch vorkommen, dass ein GP-Lauf in einem lokalen Optimum verbleibt, weil der Graph des zugehörigen Individuums sehr klein geworden ist. Oftmals sind in solchen Situationen Graphen mit besserer Fitness wesentlich größer. Wenn durch einen schnell einsetzenden *Takeover*-Effekt zusätzlich die Diversität der Population verloren geht, ist auch der *Crossover*-Operator meist nicht mehr in der Lage, das benötigte Individuum mit besserer Fitness zu erzeugen.

  Ein Beispiel für eine entsprechende Situation tritt oftmals beim 5-Parity-Problem auf, ein BOOLEsches Klassifikationsproblem mit 32 Testcases [91]. In der Regel sind keine 100 Fitnessauswertungen nötig, um ein Individuum zu erzeugen, das 17 der 32 Testcases richtig klassifiziert. Nach wenigen weiteren Auswertungen enthält das beste

[1] Beim *Lawnmower*-Problem waren die Demes nicht erfolgreich, weil die Lösungen bereits vor dem ersten Einsetzen des Mechanismus gefunden wurden

Individuum nur noch drei *OR* und ein *AND*-Gatter (oder umgekehrt). Dieses lokale Optimum wird im Normalfall für den Rest des GP-Laufs nicht mehr verlassen.

Eine adaptive Verwendung der Graphgröße als Fitnesskriterium wäre ein Ausweg aus dieser Situation: Wenn bei stagnierender Fitness die Graphen zu klein würden, dürfte die Größe der Individuen nicht mehr in die Fitness eingehen. Um die Chancen auf ein Verlassen des lokalen Optimums zu erhöhen, könnten zusätzlich neutrale Netze [8] zur Steigerung der Diversität eingesetzt werden.

- Bevor für ein Problem GP-Läufe durchgeführt werden können, müssen Parameter wie die Populationsgröße, Turniergröße oder Operatorwahrscheinlichkeiten festgelegt werden. Diese variieren von Problem zu Problem und müssen durch Voruntersuchungen oder mit Hilfe der eigenen Erfahrung vorgegeben werden. Im Normalfall ergeben sich somit suboptimale Parametersätze.

  Es wäre wünschenswert, GP-Parameter adaptiv zur Laufzeit des GP-Systems zu bestimmen. Alternativ könnte nach einer Klassifikation verschiedener Problemtypen gesucht werden. Für jede dieser Problemklassen könnten im nächsten Schritt Richtlinien zur Parameterwahl erarbeitet werden.

- Die Isomorphiebetrachtungen aus Kapitel 10 dienen bisher nur zur Vermeidung überflüssiger Fitnessauswertungen. ROSCA schlug bereits 1995 in [96] vor, Isomorphien als Diversitätsmaß zu benutzen. Er meinte jedoch, dass die Berechnungen zu kostenintensiv wären. Mit der hier vorgestellten Implementierung ist es nun doch möglich, Untersuchungen in dieser Richtung durchzuführen.

Anwendungsgebiete

Der Fokus der bisherigen Arbeit lag auf der Entwicklung der Grundlagen für ein graphbasiertes GP-System. Hierzu wurden ausschließlich Testprobleme eingesetzt, deren Verhalten in traditionellen GP-Systemen bereits ausreichend bekannt war. Die Ergebnisse lieferten, aufgrund ihrer geringen Komplexität, Rückschlüsse für Weiterentwicklungen des Systems.

Das GP-System hat jetzt ein Stadium erreicht, in dem es mit komplexeren Problemstellungen verwendet und weiterentwickelt werden kann. Die Ergebnisse der Testprobleme haben gezeigt, dass mit dem System für viele Problemklassen gute Ergebnisse erzielt werden können. Die Hauptrichtung zukünftiger Forschung sollte jedoch auf Probleme zielen, deren Lösungen auf natürliche Weise als Graph dargestellt werden können. Die folgende Aufstellung gibt einige Beispiele:

Dynamische Systeme der Regelungstechnik Grundlage und Motivation für das graphbasierte System *GGP* war das Vorgängersystem *SCADS* (Anhang A, [69]), von dem mehrere Prinzipien übernommen und weiterentwickelt, andere jedoch nicht weiter betrachtet wurden.

Die Controller der Regelungstechnik bieten ein gutes Beispiel für den Einsatz von *GGP*: Erstens lassen sich Controller als gerichtete Multigraphen darstellen. Zweitens haben Bausteine der Regelungstechnik eine fixe Zahl von Ein- und Ausgängen und drittens ist das Ziel des Schaltungsentwurfs, kompakte Regelungen mit möglichst wenigen Schaltelementen zu entwerfen. Alle diese Kriterien sprechen für einen Einsatz von *GGP*. Es ist einerseits das einzige graphbasierte GP-System, das die Kantenzahlen einzelner Knoten bei genetischen Operationen konstant hält und andererseits kleine Individuen mit wenigen Introns erzeugt.

Es wäre weiterhin zu erforschen, ob der Einsatz hybrider Techniken, wie sie bei *SCADS* eingesetzt wurden, die Evolution verbessern könnten: Die lokalen Parameter einzelner Schaltelemente – zum Beispiel die Verzögerungsdauer eines Totzeitgliedes – können in einem gesonderten Optimierungsschritt mittels anderer Verfahren zur Parameteroptimierung (Evolutionsstrategien, Simulated Annealing, etc.) berechnet werden.

Diese Trennung von Struktur- und Parameteroptimierung könnte durch Verwendung der Isomorphie-Erweiterungen von *GGP* (Kapitel 10) unterstützt werden. Doppelte Optimierungen derselben Struktur würden auf diese Weise vermieden.

Analog können Schaltungen aus dem Bereich der Transistortechnik [59] untersucht werden. Ein Einsatz für Problemstellungen des *Evolvable Computing* mit *CGP* nach MILLER [78] ist ebenfalls denkbar.

Parallele Algorithmen Mit *GGP* lassen sich Algorithmen als Flussdiagramm darstellen und evolvieren. Dies wurde zunächst am Beispiel des *Artificial Ant*-Problems (Abschnitt 5.3.1) gezeigt. In Kapitel 11 wurde das Verfahren auf parallele Algorithmen erweitert. Die resultierenden Programme wurden genauer analysiert. Zukünftig kann dieser Ansatz auf schwerere Probleme angewendet werden oder über weitere Analysen anhand von Testbenchmarks verbessert werden.

Quantenalgorithmen Das Evolvieren von Quantenalgorithmen mittels Genetischer Programmierung wurde erstmals von WILLIAMS und GRAY durchgeführt. Ziel ist es, mit Qubits in ihrer Superposition logische Operationen auszuführen. Auf diese Weise soll die Berechnung eines zu lösenden Problems durchgeführt werden [70].

DEUTSCH zeigte in [23], wie sich Quantenalgorithmen analog zu digitalen Schaltungen als Schaltkreis aus Quantengattern darstellen lassen. Diese Darstellung entspricht wiederum einem gerichteten Graphen, wie er von *GGP* benutzt wird. Bisherige Ansätze verwenden zur Repräsentation der Algorithmen Bäume, lineare Genome (beide [111]) oder *Linear-Tree GP* [68]. Die Repräsentation durch Graphen könnte der Suche nach weiteren Quantenalgorithmen neue Impulse verleihen.

Anhang A

Modellierung und Regelung dynamischer Systeme

Im Rahmen des vom Bundesministerium für Bildung und Forschung geförderten Projektes *Genetisches Programmieren für Modellierung und Regelung dynamischer Systeme* wurde in Zusammenarbeit mit der Technischen Universität Berlin und der *DaimlerChrysler AG* das GP-System *SCADS* zur selbstorganisierenden Modellierung dynamischer und nichtlinearer Systeme sowie zur Reglerstrukturierung entworfen. Während die GP-Komponenten die Grundlage für das in dieser Arbeit vorgestellte graphbasierte GP-System *GGP* darstellen, integrierten die Partner des Fachgebiets Bionik und Evolutionstechnik der TU-Berlin Evolutionsstrategien, mit deren Hilfe die Parameter einer Reglerstruktur parallel zur Strukturoptimierung ebenfalls optimiert wurden [69].

Das Projekt knüpfte an das in einer Kooperation zwischen der *TH Darmstadt* und der *Daimler Benz AG* entwickelte System SMOG an [72], welches zur Lösung linearer Probleme der Regelungstechnik ausgelegt war, bei der internen Darstellung jedoch mit Bäumen arbeitete. Auf die Unterschiede zwischen dem graphbasierten Ansatz und anderen baumbasierten wurde bereits in Abschnitt 2.3 eingegangen. Baumbasierte Ansätze finden sich unter anderem in [88, 60, 51, 17].

In *SCADS* standen zum Zeitpunkt des Verfassens des Endberichtes *Knoten einfügen/mutieren/löschen*, *Pfad einfügen/löschen* und *Zyklus* als genetische Operatoren zur Verfügung. Die Semantik der erzeugten Graphen war funktional und als Grundfunktionen wurden – je nach Problemstellung – aus der Regelungstechnik bekannte lineare und nichtlineare Blöcke, wie etwa Verzögerungs-, Verstärkungs- oder Totzeitglieder sowie statische Nichtlinearitäten verwendet.

An dieser Stelle sollen beispielhaft eine Modellierung und eine Regelung eines dynamischen Systems vorgestellt werden, die mit *SCADS* erzeugt wurden. Die Modellierung wurde aus [100] übernommen, die Regelung aus der Diplomarbeit von SEEGATZ [104].

- In [100] zeigen SANTIBÁÑEZ KOREF *et al.*, wie mit Hilfe von *SCADS* die harmonischen

Funktionen

$$x'(t) = -y(t) \text{ mit } x(0) = 1$$
$$y'(t) = x(t) \text{ mit } y(0) = 0$$

evolviert werden können. Hierzu wird eine Rahmenstruktur erzeugt, in der die zu evolvierende GP-Struktur eingebettet wird (Abb. A.1) und die Fitnessfunktion

$$Q(x,y) = \int_0^{T_{\max}} (out_1(t) - y(t))^2 + (out_2(t) - x(t))^2 \mathrm{d}t$$

verwendet.

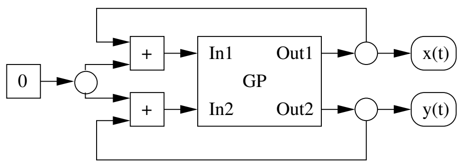


Abbildung A.1: Der Rahmen, in den das zu evolvierende GP-Individuum eingebettet wird.

Das beste Individuum, das im Rahmen der Veröffentlichung vorgestellt wurde, ist in Abbildung A.2 dargestellt und besitzt eine Fitness von 1.5×10^{-9}.

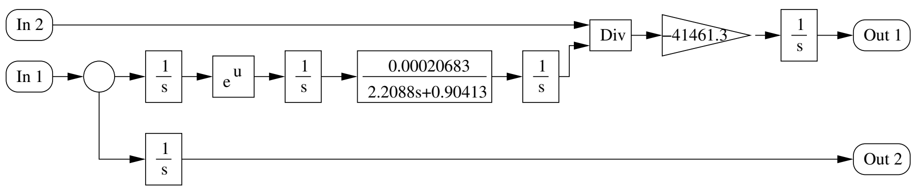


Abbildung A.2: Das beste Individuum, das die harmonischen Funktionen repräsentiert

- Im Rahmen seiner Diplomarbeit wurde von SEEGATZ versucht, mit Hilfe von *SCADS* eine Regelung für ein Flugzeugtriebwerk zu evolvieren [104]. Das nichtlineare Modell des Triebwerks wurde von der *DaimlerChrysler AG* zur Verfügung gestellt. Wegen des enorm hohen Rechenaufwands konnte allerdings nur ein Szenarium evolviert werden, die Nichtlinearitäten der Strecke wurden hierbei nicht weiter berücksichtigt.

  Als Fitnesskriterium wurden mehrere Ansätze verfolgt, von denen sich der Entwurf eines Sollbereichs für eine Sprungantwort mit aufgeprägter Störung als am günstigsten erwiesen hat. Zusätzlich wurde auf die Rückkopplung der Regelgröße über eine Zufallszahl eine Störgröße aufgeprägt.

  Es konnte ein Individuum gefunden werden, das eine für die Aufgabenstellung optimale Fitness hatte. Die Sprungantwort lag während der Regelung im Sollbereich und erfolgte in weniger als 0.35 Sekunden. Des Weiteren fand kein Überschwingen statt.

Das Fazit der Diplomarbeit war, dass *SCADS* (und somit der graphbasierte GP-Ansatz, der in dieser Arbeit mit *GGP* verfolgt wird) gut zur Identifikation von Modellen geeignet ist, die Berechnung von nichtlinearen Reglern im Moment aber noch durch zu hohe Simulationsdauer für einzelne Modelle zu rechenintensiv ist. Ein erster Schritt zur Verkürzung der Simulationsdauer wäre die Berücksichtigung der Isomorphie-Betrachtungen aus Kapitel 10.

Über den Autor

Von 1991 bis 1998 Studium der Informatik mit Nebenfach Mathematik an der Universität Dortmund, Abschluss Diplom-Informatiker. Von 1998 bis 2001 wissenschaftlicher Mitarbeiter des Informatik Centrum Dortmund in der Abteilung Complexe Adaptive Systeme und Anwendungen (CASA). Von 2001 bis 2002 wissenschaftlicher Mitarbeiter in der Arbeitsgruppe von Prof. Dr. Wolfgang Banzhaf am Lehrstuhl für Systemanalyse der Universität Dortmund. Seit 2002 Mitarbeiter der Firma Visual Systems Software & Consulting GmbH.

Schwerpunkte der wissenschaftlichen Arbeit liegen auf den Gebieten des DNA-Computings sowie der Genetischen Programmierung unter besonderer Berücksichtigung von (Selbst-)Adaptation, starker Kausalität sowie graphbasierten Genotypen.

Publikationen

Artikel in Zeitschriften

[82] Niehaus J., C. Igel und W. Banzhaf [2004] „Graph Genetic Programming and Neutrality“, *in Vorbereitung*

Konferenzbeiträge

[80] Niehaus J. und W. Banzhaf [2001] „Adaption of Operator Probabilities in Genetic Programming“, *Genetic Programming, 4th European Conference, EuroGP 2001* (Hrsg.: J. F. Miller *et al.*), Springer-Verlag, Berlin, vol. 2038 of LNCS, 325–336

[81] Niehaus J. und W. Banzhaf [2003] „More on Computational Effort Statistics in Genetic Programming“, *Genetic Programming, 6th European Conference, EuroGP 2003* (Hrsg.: C. Ryan *et al.*), Springer-Verlag, Berlin, vol. 2610 of LNCS, 164–172

Berichte

– Niehaus J. [1998] *DNA-Computing: Bewertung und Simulation*, Interner Bericht der Systems Analysis Research Group, No.SYS-6.98, Universität Dortmund, Fachbereich Informatik

[69] Lohnert F., A. Schütte, J. Sprave, I. Rechenberg, I. Boblan, U. Raab, I. Santibanez Koref, W. Banzhaf, R. E. Keller, J. Niehaus und H. Rauhe [2001] „Genetisches Programmieren für Modellierung und Regelung dynamischer Systeme“, *Schlussbericht des mit Mitteln des Bundesministeriums für Bildung und Forschung geförderten Vorhabens GEPROG*

Literaturverzeichnis

[1] Andre D. und A. Teller [1996] „A Study in Program Response and the Negative Effects of Introns in Genetic Programming“, *Genetic Programming 1996: Proceedings of the First Annual Conference* (Hrsg.: J. R. Koza *et al.*), MIT Press, 12–20

[2] Angeline P. J. [1994] „Genetic Programming and Emergent Intelligence“, *Advances in Genetic Programming* (Hrsg.: K. E. Kinnear Jr.), MIT Press, Cambridge, MA, 75–98

[3] Angeline P. J. und J. B. Pollack [1994] „Coevolving high-level representations“, *Artificial Life III* (Hrsg.: C. G. Langton), Addison-Wesley, Santa Fe, 55–71

[4] Angeline P. J. [1995] „Adaptive and Self-adaptive Evolutionary Computations“, *Computational Intelligence: A Dynamic Systems Perspective*, IEEE Press, 152–163

[5] Angeline P. J. [1996] „An Investigation into the Sensitivity of Genetic Programming to the Frequency of Leaf Selection During Subtree Crossover“, *Genetic Programming 1996: Proceedings of the First Annual Conference* (Hrsg.: J. R. Koza *et al.*), MIT Press, 21–29

[6] Angeline P. J. [1996] „Two Self-Adaptive Crossover Operators for Genetic Programming“, *Advances in Genetic Programming 2* (Hrsg.: P. J. Angeline *et al.*), MIT-Press, 89–110

[7] Banzhaf W. [1993] „Genetic Programming for pedestrians“, *Proceedings of fifth the International Conference on Genetic Algorithms* (Hrsg.: S. Forrest), Morgan Kaufmann, 628–638

[8] Banzhaf W. [1994] „Genotype-phenotype-mapping and neutral variation – a case study in genetic programming“, *Parallel Problem Solving from Nature – PPSN III* (Hrsg.: Y. Davidor *et al.*), Springer-Verlag, Berlin, vol. 866 of LNCS, 322–332

[9] Banzhaf W., P. Nordin, R. E. Keller und F. D. Francone [1998] *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.*, Morgan Kaufmann, dpunkt.verlag

[10] Banzhaf W. und W. B. Langdon, [2002] „Some Considerations on the Reason for Bloat“, *Genetic Programming and Evolvable Machines*, vol. 3(1), Kluwer Academic Publishers, 81–91

[11] Bersano-Begey T. F. [1997] „Controlling Exploration, diversity and escaping local optima in GP: Adapting weights of training sets to model resource consumption“, *Late Breaking Papers at the 1997 Genetic Programming Conference*, (Hrsg.: J. R. Koza), Stanford University Bookstore, 7–10

[12] Bettenhausen K. D., P. Marenbach, S. Freyer, H. Rettenmaier und U. Nieken [1995] „Self-organizing structured modelling of a biotechnological fed-batch fermentation by means of genetic programming“, *Proc. Int. Conf. on Genetic Algorithms in Engineering Systems: Innovations and Applications*, vol. 414 of IEE Conference Publication, IEE, Sheffield, UK, 481–486.

[13] Brameier M., W. Kantschik, P. Dittrich und W. Banzhaf [1998] „SYSGP - A C++ library of different GP variants“, *Internal Report of SFB 531*, Universität Dortmund, Fachbereich Informatik, ISSN 1433-3325

[14] Brameier M. und W. Banzhaf [2001] „Evolving teams of predictors with linear genetic programming.“, *Genetic Programming and Evolvable Machines*, vol. 2(4), Kluwer Academic Publishers, Boston, 381–407

[15] Brameier M. und W. Banzhaf [2002] „Explicit control of diversity and effective variation distance in linear genetic programming“, *Proceedings of the 5th European Conference, EuroGP 2002* (Hrsg.: J. A. Foster *et al.*), Springer-Verlag, Berlin, vol. 2278 of LNCS, 37–49

[16] Bruce W. S. [1997] „The Lawnmower Problem Revisited: Stack-Based Genetic Programming and Automatically Defined Functions“, *Genetic Programming 1997: Proceedings of the Second Annual Conference* (Hrsg.: J. R. Koza *et al.*), Morgan Kaufmann, 52–57

[17] Brucherseifer E., P. Bechtel, S. Freyer und P. Marenbach [2001] „An Indirect Block-Oriented Representation for Genetic Programming“, *Genetic Programming, 4th European Conference, EuroGP 2001* (Hrsg.: J. F. Miller *et al.*), Springer-Verlag, Berlin, vol. 2038 of LNCS, 268–279

[18] Burke E., S. Gustafson und G. Kendall [2002] „A Survey and Analysis of Diversity Measures in Genetic Programming“ *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002* (Hrsg.: W. B. Langdon *et al.*), Morgan Kaufmann, San Francisco, 716–723

[19] Busch J., J. Ziegler, W. Banzhaf, A. Ross, D. Sawitzki und C. Aue [2002] „Automatic generation of control programs for walking robots using genetic programming“, *Proceedings of the 5th European Conference, EuroGP 2002* (Hrsg.: J. A. Foster *et al.*), Springer-Verlag, Berlin, vol. 2278 of LNCS, 258–267

[20] Chelapilla K. [1997] „Evolutionary Programming with tree mutations: Evolving computer programs without crossover“, *Genetic Programming 1997: Proceedings of the Second Annual Conference* (Hrsg.: J. R. Koza *et al.*), Morgan Kaufmann, 431–438

[21] Christensen S. und F. Oppacher [2002] „An analysis of Koza's computational effort statistic for genetic programming.“, *Proceedings of the 5th European Conference, EuroGP 2002* (Hrsg.: J. A. Foster *et al.*), Springer-Verlag, Berlin, vol. 2278 of LNCS, 182–191

[22] de Jong E., R. Watson und J. Pollack [2001] „Reducing bloat and promoting diversity using multiobejective methods“, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2001* (Hrsg.: L. Spector *et al.*), Morgan Kaufmann, San Francisco, 11–18

[23] Deutsch D. [1989] „Quantum Computational Networks“, *Proceedings of the Royal Society of London Series A*, vol. 425, 73–90

[24] Deutsch L. P. und J.-L. Gailly [1996] *ZLIB Compressed Data Format Specification version 3.3*, Request for Comments: RFC1950, `http://www.faqs.org/ftp/rfc/rfc1950.pdf`

[25] Deutsch L. P. [1996] *DEFLATE Compressed Data Format Specification version 1.3*, Request for Comments: RFC1951, `http://www.faqs.org/ftp/rfc/rfc1951.pdf`

[26] Deutsch L. P. [1996] *GZIP file format specification version 4.3*, Request for Comments: RFC1952, `http://www.faqs.org/ftp/rfc/rfc1952.pdf`

[27] D'haeseleer P. [1994] „Context preserving crossover in genetic programming“, *Proceedings of the 1994 IEEE World Congress on Computational Intelligence, vol. 1*, IEEE Press, 256–261

[28] D'haeseleer P. und J. Bluming [1994] „Effects of Locality in Individual and Population Evolution“, *Advances in Genetic Programming* (Hrsg.: K. E. Kinnear jr.), MIT Press, Cambridge, MA, 177–198

[29] Diestel R. [2000] *Graphentheorie*, Springer-Verlag, Heidelberg 1996, 2000

[30] Ekárt A. und S. Z. Németh [2001] „Selection Based on the Pareto Nondomination Criterion for Controlling Code Growth in Genetic Programming“ *Genetic Programming and Evolvable Machines*, vol. 2(1), Kluwer Academic Publishers, 61–73

[31] Eshelman L. J. und J. D. Schaffer [1993] „Crossover's niche“, *Proceedings of Fifth the International Conference on Genetic Algorithms* (Hrsg.: S.Forrest), Morgan Kaufmann, 9–14

[32] Fogel, L., A. Owens und M. Walsh [1996] *Artificial Intelligence through Simulated Evolution*, John Wiley & Sons, New York

[33] Gathercole C. und P. Ross [1994] „Dynamic training subset selection for supervised learning in genetic programming“, *Parallel Problem Solving from Nature – PPSN III* (Hrsg.: Y. Davidor *et al.*), Springer-Verlag, Berlin, vol. 866 of LNCS, 312–321

[34] Goldberg D. E. und K. Deb [1991] „A comparative analysis of selection schemes used in genetic algorithms“, *Foundations of Genetic Algorithms* (Hrsg.: G. J. E. Rawlins), Morgan Kaufmann, San Mateo, 69–93

[35] Groß R., K. Albrecht, W. Kantschik und W. Banzhaf [2002] „Evolving chess playing programs“, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002* (Hrsg.: W. B. Langdon *et al.*), Morgan Kaufmann, San Francisco, 740–747

[36] Grötzner, M [2000], *Eine überstrukturfreie Methode zur Synthese verfahrenstechnischer Prozesse*, Fortschritt-Berichte VDI Reihe 3, Verfahrenstechnik, vol. 622, VDI-Verlag, Düsseldorf

[37] Gruau, F. [1996] „Artificial cellular development in optimization and compilation“, *Towards Evolvable Hardware* (Hrsg.: E. Sanchez *et al.*), Springer-Verlag, Berlin, vol. 1062 of LNCS, 48–75

[38] Güting, R. H. [1992] *Datenstrutkuren und Algorithmen*, B. G. Teubner, Stuttgart

[39] Hansen N. und A. Ostermeier [2001] „Completely Derandomized Self-Adaptation in Evolution Strategies“, *Evolutionary Computation*, vol. 9(2), 159–195

[40] Heywood M. I. und A. N. Zincir-Heywood [2002] „Dynamic Page Based Crossover in Linear Genetic Programming“, *IEEE Transactions on Systems, Man, and Cybernetics: Part B - Cybernetics*, IEEE-Press, 380–388

[41] Hinterding R., Z. Michalewicz und T. Peachy [1996] „Self-Adaptive Genetic Algorithm For Numeric Functions“, *Parallel Problem Solving from Nature (PPSN IV), International Conference on Evolutionary Computation* (Hrsg.: H.-M. Voigt *et al.*), Springer-Verlag, Berlin, vol. 1141 of LNCS, 420–429

[42] Holland, J. [1992] *Adaptation in natural and artificial systems*, MIT Press, Cambridge, MA

[43] Hopcroft J. E. und J. D. Ullman [1990] *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*, Addison-Wesley, Bonn

[44] Iba H., H. de Garis und T. Sato [1994] „Genetic Programming Using a Minimum Description Length Principle“, *Advances in Genetic Programming* (Hrsg.: Kinnear jr. K. E.), MIT Press, Cambridge, MA, 265–284

[45] Igel C. [1998] „ Causality of Hierarchical Variable Length Representations“, *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, IEEE Press, 324–329

[46] Igel C. und Chellapilla K. [1999] „Investigating the Influence of Depth and Degree of Genotypic Change on Fitness in Genetic Programming“, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '99* (Hrsg.: W. Banzhaf *et al.*), Morgan Kaufmann, San Francisco, 1061–1068

[47] Igel C. und P. Stagge [2002] „Graph isomorphisms affect structure optimization of neural networks“, *International Joint Conference on Neural Networks 2002 (IJCNN)*, IEEE Press, 142–147

[48] Igel C. [2003] *Beiträge zum Entwurf neuronaler Systeme*, Shaker Verlag, Aachen

[49] Kantschik W. und W. Banzhaf [2001] „Linear-Tree GP and Its Comparison with Other GP Structures“, *Genetic Programming, 4th European Conference, EuroGP 2001* (Hrsg.: J. F. Miller *et al.*), Springer-Verlag, Berlin, vol. 2038 of LNCS, 302–312

[50] Kantschik W. und W. Banzhaf [2002] „Linear-Graph GP – A new GP Structure“, *Genetic Programming, 5th European Conference, EuroGP 2002* (Hrsg.: J. A. Foster *et al.*), Springer-Verlag, Berlin, vol. 2278 of LNCS, 83–92

[51] Keane M. A., J. Yu und J. R. Koza [2000] „Automatic Synthesis of Both Topology and Tuning of a Common Parameterized Controller for Two Families of Plants using Genetic Programming“, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2000* (Hrsg.: D. Whitley *et al.*), Morgan Kaufmann, San Francisco, 496–504

[52] Keijzer M. [1996] „Efficiently Representing Populations in Genetic Programming“, *Advances in Genetic Programming 2* (Hrsg.: P. J. Angeline *et al.*), MIT-Press, 259–278

[53] Kennedy J. und R. C. Eberhart [1995] „Particle swarm optimization“, *Proceedings of the 1995 IEEE International Conference on Neural Networks*, IEEE Service Center, Piscataway, NJ, IV: 1942–1948

[54] Kennedy C. J. und C. Giraud-Carrier [1999] „A Depth Strategy for Strongly Typed Evolutionary Programmming“, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '99* (Hrsg.: W. Banzhaf *et al.*), Morgan Kaufmann, San Francisco, 879–885

[55] Kirkpatrick S., C. D. Gelatt Jr. und M. P. Vecchi [1983] „Optimization by Simulated Annealing“, *Science*, no. 4598, 13 May 1983, 671–680

[56] Koza J. R. [1990] *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*, Stanford Univ.

[57] Koza J. R. [1992] *On the Programming of Computers by Natural Selection*, The M.I.T. Press, Cambridge, MA

[58] Koza J. R. [1994] *Genetic Programming II: Automatic Discovery of Reusable Programs*, The M.I.T. Press, Cambridge, MA

[59] Koza J. R., F. H. Bennet III, J. D. Lohn, F. Dunlap, D. Andre und M. A. Keane [1997] „Automated Synthesis of Computational Circuits using Genetic Programming“, *1997 IEEE International Conference on Evolutionary Computation*, IEEE Press, Piscataway, NJ, 447–452

[60] Koza J. R., F. H. Bennet III, M. A. Keane und D. Andre [1997] „Evolution of a Time-Optimal Fly-To Controller Circuit using Genetic Programming“, *Genetic Programming 1997: Proceedings of the Second Annual Conference* (Hrsg.: J. R. Koza *et al.)*, Morgan Kaufmann, San Francisco, 207–212

[61] Koza, J. R., M. A. Keane, J. Yu, F. H. Bennett III und W. Mydlowec [2000] „Automatic Creation of Human-Competitive Programs and Controllers by Means of Genetic Programming“, *Genetic Programming and Evolvable Machines*, vol. 1(1-2), Kluwer Academic Publishers, 121–164

[62] Kuscu I. [1998] „Evolving a generalised behavior: Artificial ant problem revisited“, *Seventh Annual Conference on Evolutionary Programming* (Hrsg.: V. W. Porto, *et al.*), Springer-Verlag, Berlin, vol. 1447 of LNCS, 799–808

[63] Langdon W. B. [1996] „Evolution of Genetic Programming Populations“, *Research Note RN/96/125*, University College London, Gower Street, London WC1E 6BT, UK

[64] Langdon W. B. [1997] „Fitness causes Bloat“, *Soft Computing in Engineering Design and Manufacturing* (Hrsg.: P. K. Chawdhry *et al.*), Springer-Verlag, London, 13–22

[65] Langdon W. B. [1998] „Better trained ants“, *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming* (Hrsg.: R. Poli *et al.*), CSRP-98-10, The University of Birmingham, UK, 11–13

[66] Langdon W. B. und R. Poli [1998] „Why ants are hard“, *Genetic Programming 1998: Proceedings of the Third Annual Conference* (Hrsg.: J. R. Koza *et al.*), Morgan Kaufmann, San Francisco, 193–201

[67] Langdon, W. B. und W. Banzhaf [2000] „Genetic Programming Bloat without Semantics“, *Parallel Problem Solving from Nature - PPSN VI 6th International Conference* (Hrsg.: M. Schoenauer *et al.*), Springer-Verlag, Berlin, vol. 1917 of LNCS, 201–210

[68] Leier A. und W. Banzhaf [2003] „Evolvability of Quantum Algorithms“, *5th Genetic and Evolutionary Computation Conference (GECCO-2003), to be published*

[69] Lohnert F., A. Schütte, J. Sprave, I. Rechenberg, I. Boblan, U. Raab, I. Santibanez Koref, W. Banzhaf, R. E. Keller, J. Niehaus und H. Rauhe [2001] „Genetisches Programmieren für Modellierung und Regelung dynamischer Systeme“, *Schlussbericht des mit Mitteln des Bundesministeriums für Bildung und Forschung geförderten Vorhabens GEPROG*

[70] Lloyd S. [1993] „A potentially realizable quantum computer“, *Science*, vol. 261, 1569–1571

[71] Luke S. und L. Panait [2002] „Is the perfect the enemy of the good?“, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002* (Hrsg.: W. B. Langdon *et al.*), Morgan Kaufmann, San Francisco, 820–828

[72] Marenbach P. [1997] „SMOG - Structured MOdel Generator. Ein Werkzeug zur selbstorganisierenden Modellbildung“, *Kurzdokumentation*, FG Regelsystemetheorie & Robotik, TH Darmstadt

[73] McKay B. D. [1978] „Computing automorphisms and canonical labellings of graphs“, *Combinatorial Mathematics, Lecture Notes in Mathematics, 686*, Springer-Verlag, Berlin, 223–232

[74] McKay B. D. [1981] „Practical graph isomorphism“, *Congressus Numerantium 30*, 45–87

[75] McKay B. D. [1990] „nauty User's Guide (version 1.5)“, *Tech. Rpt. TR-CS-90-02*, Dept. Computer Science, Austral. Nat. Univ

[76] McPhee N. F. und J. D. Miller [1995] „Accurate replication in genetic programming“, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)* (Hrsg.: L. Eshelman), Morgan Kaufmann, San Francisco, 303–309

[77] McPhee N. F. und N. J. Hopper [1999] „Analysis of genetic diversity through population history“, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '99* (Hrsg.: W. Banzhaf *et al.*), Morgan Kaufmann, San Francisco, 1112–1120

[78] Miller J. F. [1999] „An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming Approach“, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '99* (Hrsg.: W. Banzhaf *et al.*), Morgan Kaufmann, San Francisco, 1135-1142

[79] Miyazaki T. [1997] „The Complexity of McKay's Canonical Labeling Algorithm“, *Groups and Computation II, DIMACS Series on Discrete Mathematics and Theoretical Computer Science, vol. 28* (Hrsg.: L. Finkelstein und W. M. Kantor), Amer. Math. Soc., Providence, RI, 239–256

[80] Niehaus J. und W. Banzhaf [2001] „Adaption of Operator Probabilities in Genetic Programming“, *Genetic Programming, 4th European Conference, EuroGP 2001* (Hrsg.: J. F. Miller *et al.*), Springer-Verlag, Berlin, vol. 2038 of LNCS, 325–336

[81] Niehaus J. und W. Banzhaf [2003] „More on Computational Effort Statistics in Genetic Programming“, *Genetic Programming, 6th European Conference, EuroGP 2003* (Hrsg.: C. Ryan *et al.*), Springer-Verlag, Berlin, vol. 2610 of LNCS, 164–172

[82] Niehaus J., C. Igel und W. Banzhaf [2003] „Graph Isomorphisms in Genetic Programming“, *in Vorbereitung*

[83] Nordin, J.P. [1994] „A Compiling Genetic Programming System that Directly Manipulates the Machine-Code“ *Advances in Genetic Programming* (Hrsg.: K. E. Kinnear Jr.), MIT Press, Cambridge, MA, 311–331

[84] O'Neill M., C. Ryan und M. Nicolau [2001] „Grammar Defined Introns: An Investigation Into Grammars, Introns, and Bias in Grammatical Evolution“, *Proceedings of the*

Genetic and Evolutionary Computation Conference, GECCO 2001 (Hrsg.: L. Spector *et al.*), Morgan Kaufmann, San Francisco, 97–103

[85] O'Neill M., C. Ryan, M. Keijzer und M. Cattolico [2003] „Crossover in Grammatical Evolution“, *Genetic Programming and Evolvable Machines*, vol. 4(1), Kluwer Academic Publishers, 67–93

[86] O' Reilly [1997] „Using a Distance Metric on Genetic Programs to Understand Genetic Operators“, *Late Breaking Papers at the 1997 Genetic Programming Conference*, (Hrsg.: J. R. Koza), Stanford University Bookstore, 188–198

[87] Pereira F. B., P. Machado, E. Costa und A. Cardoso [1999] „Graph Based Crossover - A Case Study with the Busy Beaver Problem“, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '99* (Hrsg.: W. Banzhaf *et al.*), Morgan Kaufmann, San Francisco, 1149–1155

[88] Pohlheim H. und P. Marenbach [1996] „Generation of structured process models using genetic algorithms“, *Workshop on Evolutionary Computation AISB96* (Hrsg.: T. C. Fogarty), Springer-Verlag, Berlin, vol. 1143 of LNCS, 102–109

[89] Poli R. [1996] „Some Steps Towards a Form of Parallel Distributed Genetic Programming“, *Proceedings of the First On-line Workshop on Soft Computing*, Nagoya, 290-295

[90] Poli R. [1997] „Evolution of graph-like programs with parallel distributed genetic programming“, *Genetic Algorithms: Proceedings of the Seventh International Conference* (Hrsg.: T. Bäck), Morgan Kaufmann, 346–353

[91] Poli R., J. Page und W. B. Langdon [1999] „Smooth Uniform Crossover, Sub-Machine Code GP and Demes: A Recipe For Solving High-Order Boolean Parity Problems“, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '99* (Hrsg.: W. Banzhaf *et al.*), Morgan Kaufmann, San Francisco, 1162–1169

[92] Punch B. und D. Zongker, *lil-gp Genetic Programming System*, `http://garage.cse.msu.edu/software/lil-gp/lilgp-index.html`

[93] Rechenberg I. [1994] *Evolutionsstrategie'94*, Frommann-Holzboog, Stuttgart

[94] Rosca, J. P. und D. H. Ballard [1994] „Learning by adapting representations in genetic programming“, *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, IEEE Press, 407–412

[95] Rosca, J. P. [1995] „Genetic Programming Exploratory Power and the Discovery of Functions“, *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming* (Hrsg.: J. R. McDonnell *et al.*), MIT Press, 719–736

[96] Rosca, J. P. [1995] „Entropy-driven adaptive Representation“, *Proceedings of the Workshop on Genetic Programming: From Theory to Real World Application* (Hrsg.: J. P. Rosca), TR-NRL02, University of Rochester, Rochester NY, 23–32

[97] Rothermich J. und J. F. Miller [2002] „Studying the Emergence of Multicellularity with Cartesian Genetic Programming in Artificial Life“, *Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002)* (Hrsg.: E. Cantú-Paz), AAAI, 445 Burgess Drive, Menlo Park, CA 94025, 397-403

[98] Rudolph G. [2000] „Takeover Times and Probabilitites of Non-Generational Selection Rules“, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2000* (Hrsg.: D. Whitley *et al.*), Morgan Kaufmann, San Francisco, 903–910

[99] Sankhoff D. und J. B. Kruskal (Hrsg.) [1983] *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley Publishing Company, Inc.

[100] Santibáñez Koref I., I. Boblan, F. Lohnert und A. Schütte [2001] „Causality and Design of Dynamical Systems“, *Evolutionary Methods for Design, Optimization and Control, Procceedings of the EUROGEN2001 Conference* (Hrsg.: K.C. Giannakoglou *et al.*), VerlagInternational Center for Numerical Methods in Engineering (CIMNE), Barcelona, 229–234

[101] Schaudolph N.N. und R. K. Belew [1992] „Dynamic parameter encoding forgenetic algorithms“, *Machine Learning*, vol. 9(1), Kluwer Academic Publishers, Boston, 9–22

[102] Schaffer J. D. und A. Morishima [1987] „An adaptive crossover distribution mechanism for genetic algorithms“ *Proc. of the Second International Conference on Genetic Algorithms* (Hrsg.: J. J. Grefenstette), Lawrence Erlbaum Associates, Hillsdale, NJ, 36–40

[103] Schwefel H.-P. [1995] *Evolution and Optimum Seeking*, John Wiley, Chichester, UK

[104] Seegatz H. [2001] „Reglerentwurf mit Genetischer Programmierung“, *Diplomarbeit*, FHTW Berlin

[105] Shaefer C. G. [1987] „The ARGOT System: adaptive Representation Genetic Optimizing Technique“, *Proc. of the Second International Conference on Genetic Algorithms* (Hrsg.: J. J. Grefenstette), Lawrence Erlbaum Associates, Hillsdale, NJ, 50–58

[106] Smith P. W. H. und K. Harries [1998] „Code growth, explicitly defined introns, and alternative selection schemes“, *Evolutionary Computation*, vol. 6(4), 339–360

[107] Smith P. W. H. [1999] „Controlling Code Growth in Genetic Programming“, *Soft Computing and its Techniques* (Hrsg.: R. John und R. Birkenhead), Physica-Verlag, 166–171

[108] Soule T., J. A. Foster und J. Dickinson [1996] „Code Growth in Genetic Programming“, *Genetic Programming 1996: Proceedings of the First Annual Conference* (Hrsg.: J. R. Koza *et al.*), MIT Press, 215–223

[109] Soule T. und R. B. Heckendorn [2002] „An Analaysis of the Causes of Code Growth in Genetic Programming“, *Genetic Programming and Evolvable Machines*, vol. 3(3), Kluwer Academic Publishers, 283–309

[110] Spector L. und S. Luke [1996] „Cultural Transmission of Information in GP“, *Genetic Programming 1996: Proceedings of the First Annual Conference* (Hrsg.: J. R. Koza *et al.*), MIT Press, 209–214

[111] Spector L., H. Barnum, H. J. Bernstein und N. Samy [1999] „Quantum Computing Applications of Genetic Programming“, *Advances in Genetic Programming, Volume 3* (Hrsg.: L. Spector *et al.*), MIT Press, Cambridge, 135–160

[112] Tackett W. A [1994] *Recombination, Selection, and the Genetic Construction of Computer Programs*, PhD thesis, University of Southern California, Department of Electrical Engineering Systems

[113] Tanese R. [1989] „Distributed Genetic Algorithms“, *Proceedings of the Third International Conference on Genetic Algorithms* (Hrsg.: J. D. Schaffer), Morgan Kaufmann, 434–439

[114] Tanev I., T. Uozumi und K. Ono [2001] „Scalable architecture for parallel distributed implementation of genetic programming on network of workstations“, *Journal of Systems Architecture*, vol. 47, 557–572

[115] Teller A. und M. Veloso [1995] „Algorithm Evolution for Face Recognition: What Makes a Picture Difficult“, *International Conference on Evolutionary Computation*, IEEE Press, Piscataway, NJ, 608–613

[116] Teller A. und M. Veloso [1995] „Program Evolution for Data Mining“, *The International Journal of Expert Systems*, JAI Press, vol. 8(3), 216–236

[117] Teller A. und M. Veloso [1996] „PADO: A New Learning Architecture for Object Recognition“, *Symbolic Visual Learning* (Hrsg.: K. Ikeuchi *et al.*), Oxford University Press, 81–116

[118] Terrio M. D. und M. I. Heywood [2002] „Directing Crossover for Reduction of Bloat in GP“, *Proceedings of the 2002 IEEE Canadian Conference on Electrical & Computer Engineering*, IEEE Press, 1111–1115

[119] Wegener I. [1993] *Theoretische Informatik*, Leitfäden und Monographien der Informatik, B.G. Teubner, Stuttgart

[120] Whitley D., K. Mathias und P. Fitzhorn [1991] „Delta coding: An iterative search strategy for genetic algorithms“, *Proceedings of the Fourth International Conference on Genetic Algorithms* (Hrsg.: R. Belew *et al.*), Morgan Kaufman, San Mateo, 77–84

[121] Whitley D. [2001] „An Overview of Evolutionary Algorithms: Practical Issues and Common Pitfalls“, *Information and Software Technology*, vol. 43(14), 817–831

[122] Williams C. P und A. G. Gray [1998] „Automated Design of Quantum Circuits“, *Quantum Computing and Quantum Communications 98* (Hrsg: C. P. Williams), Springer-Verlag, Berlin, vol. 1509 of LNCS, 113–125

[123] Wu A. S. und R. K. Lindsay [1996] „A survey of intron research in genetics", *Parallel Problem Solving from Nature (PPSN IV), International Conference on Evolutionary Computation* (Hrsg.: H.-M. Voigt *et al.*), Springer-Verlag, Berlin, vol. 1141 of LNCS, 101–110

[124] Ziegler J. [2002] *On the influence of adaptive operator probabilities in genetic programming*, Technical Report Sys-03/02, Chair of Systems Analysis, University of Dortmund, 2002. ISSN 0941-4568.

[125] Zhang B. T. und H. Mühlenbein [1995] „Balancing Accuracy and Parsimony in Genetic Programming", *Evolutionary Computation*, vol. 3(1), 17–38