Universität Kassel
Fachgebiet Verteilte Systeme
Prof. Dr. Kurt Geihs

Projekt: Distributed Genetic Programming Framework (DGPF)
Projektleiter: Dipl. Inf. Thomas Weise

Implementierung von Simulated Annealing in einem DGPF

von Marc Kirchhoff

Inhaltsverzeichnis

1. Einleitung

Ziel des von mir bearbeiteten Projektes war es das Distributed Genetic Programming Framework (DGPF) [1][2][3] um ein weiteres heuristisches Suchverfahren, den so genannten Simulated Annealing Algorithmus [5][7], zu erweitern.
Das DGPF ist ein auf Genetischen Algorithmen basierendes, verteiltes Framework zum automatischen Erzeugen von Programmen für Sensornetzwerke.
Diese Ausarbeitung gibt einen Überblick über das DGPF, den zu implementierenden Algorithmus und die Implementierung an sich.
In Kapitel 2 wird zunächst auf Genetische Algorithmen und auf die ihnen zugrunde liegenden Konzepte eingegangen. Kapitel 3 gibt einen Überblick über das eben bereits erwähnte Distributed Genetic Programming Framework, zeigt seinen grundlegenden Aufbau und die wichtigsten Klassen.
Kapitel 4 schließlich stellt das so genannten Simulated Annealing vor und zeigt eine mögliche Implementierung innerhalb des DGPFs.

2. Genetische Algorithmen

Dieses Kapitel gibt einen Überblick über genetische Algorithmen und die ihnen zugrunde liegenden Konzepte.

Genetische Algorithmen lehnen sich an die Theorie der biologischen Evolution an, wobei allerdings nicht versucht wird die Evolution möglichst genau nachzubilden - es werden lediglich die zugrunde liegenden Konzepte übernommen. Um aber eine Vorstellung von der Idee, die hinter den Genetischen Algorithmen steht, zu bekommen, ist es notwendig wenigstens eine sehr grobe Vorstellung von den Vorgängen der Evolution zu besitzen.
Die zwei herausstechensten Merkmale der Evolution (nach der Evolutionstheorie von Charles Darwin), die von den genetischen Algorithmen in sehr abstrakter Weise übernommen werden, sind:

1.) die Fähigkeit sich anzupassen
2.) die natürliche Auslese.

Um die eigenen Überlebenschancen zu erhöhen sind Lebewesen in der Lage sich im Laufe der Evolution an veränderte Lebensbedingungen anzupassen. Dabei entstehen durch Veränderungen der Gene immer neue Generationen von Lebewesen die die vorherigen Generationen in ihrer Überlebensfähigkeit übertreffen.
Das zweite wesentliche Merkmal der Evolution ist die natürliche Auslese, die zum Aussterben der schwachen und zur Vermehrung der starken Lebewesen führt.

Wie schon erwähnt basieren Genetische Algorithmen auf diesen zwei Grundprinzipien.
Ein weiteres Merkmal das die Genetischen Algorithmen in sehr abstrakter Weise von der Evolution übernommen haben ist die Art und Weise wie sich Lebewesen durch Veränderung an ihre Umwelt anpassen. Die Veränderung der Gene findet auf zwei Arten statt: Kreuzung (Crossover) und Mutation. Neuartige Gene entstehen zum einen durch eine Vermischung der Elterngene und zum anderen verändert sich das Erbgut durch äußere Einflüsse selbst.
Die Evolution ist ein ständiger aus folgenden Schritten bestehender Kreislauf [5]:

1.) Neuerschaffung
2.) Überlebensprüfung
3.) Vermehrung der Besten
4.) Veränderung des Vorhandenen (durch Crossover und Mutation)

Genetische Algorithmen orientieren sich an diesen Schritten.
Ein Einsatzgebiet von Genetische Algorithmen, an dem sich dieser Ablauf beispielhaft erklären lässt, ist das automatisierte Erzeugen von Programmen.
Der erste Schritt bei der Suche nach einem Programm zur Lösung eines bestimmten Problems besteht in der Generierung einer Startpopulation, bestehend aus Programmen mit zufällig generiertem Code (Individuen).

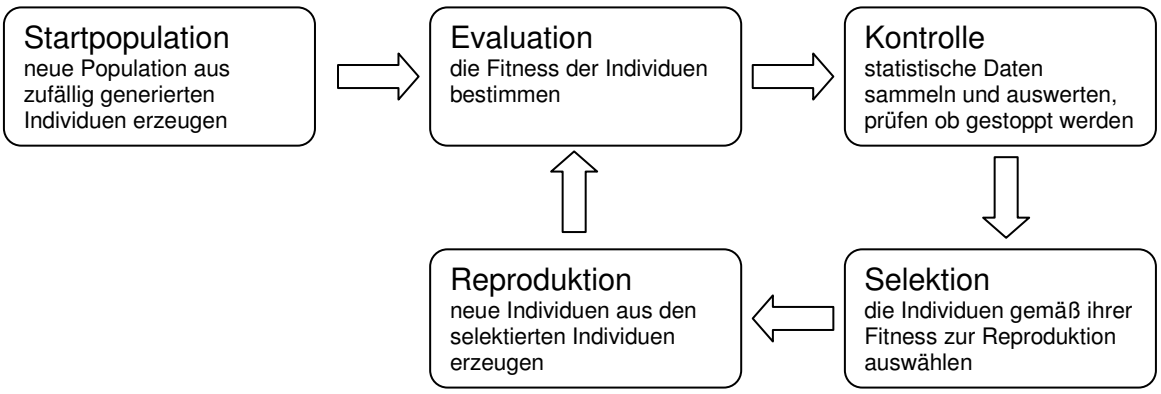

Abbildung 1: Funktionsweise genetischer Algorithmen

Im nächsten Schritt werden diese zufällig generierten Individuen getestet, wobei ihnen dann anhand der Testergebnisse (inwiefern stimmen die gefunden Ergebnisse mit den erwarteten Ergebnissen überein?) und verschiedener anderer Kriterien durch Fitnessfunktionen ein oder mehrere Fitnesswerte zugewiesen werden. Verwendet man mehrere Fitnessfunktionen, so nennt man den Genetischen Algorithmus „multiobjektiv“. Beispielsweise könnte neben der Richtigkeit des Ergebnisses auch die Programmgröße oder der benötigte Speicher von Bedeutung sein.

Im nächsten Schritt werden statistische Daten gesammelt und ausgewertet. Anhand dieser Daten wird dann entschieden ob eine ausreichend gute Lösung gefunden wurde und ob die Suche gestoppt werden kann.
Anschließend werden anhand der Fitnesswerte die besten Individuen zur weiteren Reproduktion ausgewählt (siehe Abbildung 1).
Im nächsten Schritt werden aus diesen Individuen mittels Mutation oder Crossover wieder neue Individuen erzeugt.
Dieser Prozess wird nun solange wiederholt bis ein ausreichend gutes Programm gefunden wurde.
Diese Vorgehensweise kann natürlich nicht nur zum Erzeugen von Programmen, sondern für eine Vielzahl von Optimierungsproblemen eingesetzt werden. Ein weiteres Beispiel für ein mit Genetischen Algorithmen lösbares Optimierungsproblem wäre z.B. das Finden des globalen Minimums einer mathematischen Funktion. Bei den Individuen würde es sich dann um einzelne Zahlen oder Vektoren handeln, die einen umso besseren Fitnesswert zugewiesen bekämen je näher sie sich an dem globalen Minimum befänden.

3. DGPF

3.1 Überblick

Bei dem DPGF (Distributed Genetic Programming Framework) handelt es sich um ein sich noch in der Entwicklung befindendes Framework, das in der Lage ist auf Basis von randomisierten, heuristischen Suchverfahren automatisch Programmcode für bestimmte Problemstellungen zu erzeugen.
Das in Java geschriebene, vollständig (in Englisch) dokumentierte Framework steht unter der „Lesser General Public License“ (LGPL) und ist daher frei erhältlich. Heruntergeladen werden kann das Programm inklusive Programmcode unter [2]. Weiterführende Dokumente und aktuelle Informationen findet man unter [1].
Das übergeordnete Hauptziel dieses Projektes ist es ein Framework zur automatischen Erzeugung von Algorithmen für Sensornetze bereitzustellen. Dabei stützt sich das DGPF hauptsächlich auf genetischen Algorithmen, ist allerdings nicht an diese gebunden, so dass auch andere heuristische Suchverfahren wie z.B. Hill Climbing oder das weiter unten im Detail beschriebene Simulated Annealing eingesetzt werden können.
Der modulare Aufbau des DGPF erlaubt den Einsatz in verschiedensten Anwendungsgebieten [4].
Neben der lokalen Ausführung (siehe Abbildung 2, A)) bietet das Framework auch mehrere Möglichkeiten, die Berechnungen auf mehreren Rechnern verteilt ausführen zu lassen.
Es existieren aktuell drei Verfahren zur verteilten Berechnung:

1.) Client/Server (Abbildung 2, B))
2.) Peer-to-Peer (Abbildung 2, C))

3.) Hybride Verteilung (Abbildung 2, D))

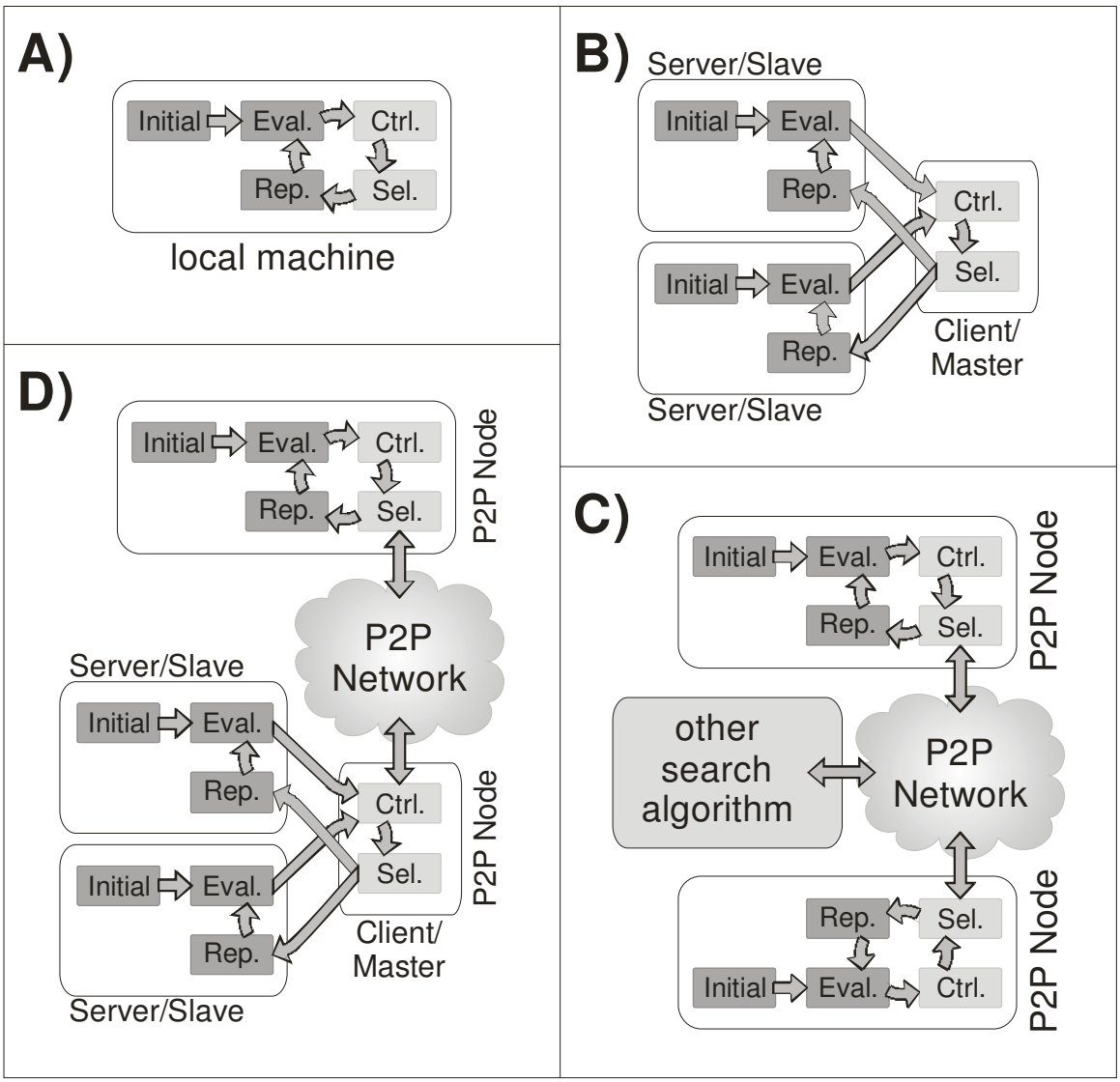

Abbildung 2: Ausführungsarten

1.) Bei dem Client/Server-Modell werden die Evaluation und die Replikation (siehe Bild 2, B)) vom Client auf einen oder mehrere Server ausgelagert. Die Kontrolle und die Selektion werden aber weiterhin vom Client ausgeführt. Der Grund warum gerade diese beiden Schritte auf den Server ausgelagert werden liegt darin, dass die Evaluation zum Beispiel bei der automatischen Erzeugung von Programmen die Simulation dieser Programme beinhaltet, was unter Umständen sehr Zeit- und Rechenaufwendig sein kann. Stehen mehrere Server zur Verfügung, so erlaubt dieses Verteilungsverfahren des Weiteren die parallele Abarbeitung dieser Schritte auf verschiedenen Rechnern, was unter Umständen sehr große Geschwindigkeitsvorteile zur Folge haben kann.

2.) Die verteilte Berechnung innerhalb eines Peer-to-Peer-Netzes erlaubt das Erzeugen von sehr großen Populationen. Dabei existiert eine beliebige Anzahl von Knoten die alle an demselben Problem arbeiten.
Jeder Knoten führt lokal alle Schritte aus (siehe Bild 2, C)), mit der Besonderheit, dass die Knoten während der Selektionsphase untereinander Individuen austauschen können. Dadurch ergibt sich die Möglichkeit die Gesamtevolution durch Steuerung dieses Individuenaustausches positiv zu beeinflussen. Eine relativ einfache aber sinnvolle Strategie wäre z.B. die Emigrationsrate der Knoten zu erhöhen die besonders gute Individuen hervorbringen.

3.) Bei dem dritten Verfahren, der hybriden Verteilung (siehe Bild 2, D)), handelt es sich um eine Mischung aus dem Client/Server- und dem Peer-to-Peer-Ansatz. Dabei existiert ein wie unter 2.) beschriebenes Peer-to-Peer-Netz, wobei die Knoten innerhalb dieses Netzes die Möglichkeit haben zur Geschwindigkeitssteigerung die Evolution und die Reproduktion auf ein oder mehrere Server auszulagern.

3.2 Der Aufbau des DGPF

Dieser Abschnitt gibt einen Überblick über den grundlegenden Aufbau des Distributed Genetic Programming Frameworks und seine wichtigsten Pakete und Klassen.

Abbildung 1 zeigt den grundlegenden Aufbau des Distributed Genetic Programming Frameworks.

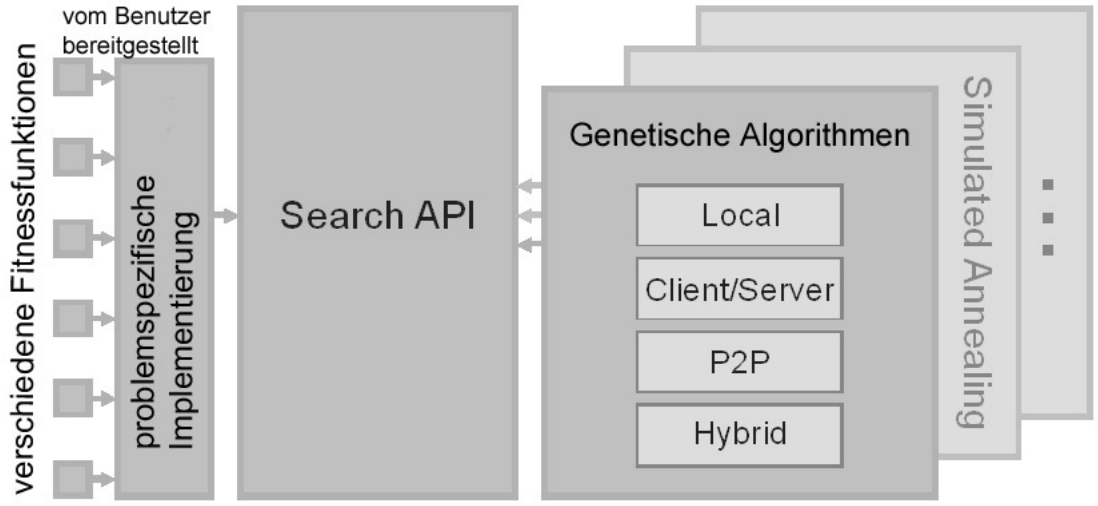

Abbildung 3: Grundaufbau des DGPF

Der Search Algorithms Layer (Search API) (org.dgpf.search) enthält alle Basisklassen die benötigt werden um heuristische Suchalgorithmen zu implementieren und zu verteilen.
Der nachfolgende Abschnitt gibt einen kurzen Überblick über die wichtigsten Klassen dieser Schicht.
Eine der wichtigsten Basisklassen ist die Klasse „SearchEngine“ aus dem Package *org.dgpf.search.api*. Die Search Engine stellt eine generelle Abstraktion für alle Suchalgorithmen dar, sie enthält als Interface alle grundlegenden Methoden für das Starten/Stoppen, für die Event-Generierung und für die Konfiguration einer Suche. Zur Implementierung eines Suchalgorithmus ist es also notwendig eine generelle Oberklasse von dieser Klasse abzuleiten, welche die Basisfunktionalitäten des jeweiligen Suchalgorithmus implementiert (z.B. die Klasse *GeneticEngine* in dem Package *org.dgpf.search.algorithms.ga* oder die von mir implementierte Klasse *SAEngine* im *org.dgpf.search.algorithms.sa* Package). Von dieser Oberklasse werden dann wiederum jeweils weitere spezielle Klassen für die lokale oder verteilte Ausführung abgeleitet (z.B. *LocaleSAEngine* (*org.dgpf.search.algorithms.sa.local*) und *CSSAEngine* (*org.dgpf.search.algorithms.sa.cs*) für die lokale und für die verteilte Client/Server Ausführung).

Das DGPF unterteilt eine Suche in mehrere Runden oder Updates. Z.B. ist jede neue Generation eines Genetischen Algorithmus ein solches Update. Das Besondere dabei ist, dass das Framework mittels so genannter Adaptionsstrategien das dynamische Anpassen und Verändern der Suchparameter nach jedem Update erlaubt. Es ist also möglich die Suchparameter manuell oder automatisch anhand der gesammelten Daten anzupassen. Die Basisklassen für diese Adaptionsstrategien befinden sich im Package *org.dgpf.search.api.adaption*, müssen aber natürlich je nach verwendetem Algorithmus überschrieben werden. Ein Beispiel für eine solche Adaptionsstragie ist z.B. die von mir implementierte Klasse *DefaultSAAdapter* die anhand bestimmter Kriterien die Temperatur (siehe Kapitel 4), mit der der Simulated Annealing Algorithmus arbeitet, nach jeder Runde erneut anpasst.

Aktuell sind drei verschiedene Suchalgorithmen in dem Framework implementiert:

1.) Genetische Algorithmen
2.) Simulated Annealing
3.) Hill Climbing

1.) Was genetische Algorithmen sind, auf welchen Grundkonzepten sie aufbauen und wie sie funktionieren wurde bereits in Kapitel 2 erläutert, weshalb hier nicht noch einmal auf ihre Funktionsweise eingegangen werden soll.
Die Implementierung innerhalb des DGPFs befindet sich in dem Package *org.dgpf.search.algorithms.ga*. Dieses Package enthält neben der Such-Engine, den Parametern und verschiedenen anderen grundlegenden Klassen noch

mehrere Unterpackages die die Implementierungen zur verteilten Ausführung beinhalten.

2.) Der Simulated Annealing Algorithmus und dessen Implementierung wird in Kapitel 4 im Detail erläutert, weshalb hier nicht weiter darauf eingegangen werden soll.

3.) Bei dem Hill Climbing Algorithmus handelt es sich um eine sehr einfache Methode zur Lösung von Optimierungsproblemen.
Zur Suche nach dem Maximum einer gegebenen Zielfunktion $f : S \to \Re$ durchläuft der Algorithmus folgende Schritte [5]:

1.) Wähle einen Startpunkt $s_0 \in S$
2.) Wähle $s_1 \in S$ in der „Nähe“ von s_0
3.) Falls $f(s_1) \geq f(s_0)$, wähle s_1 als neuen Startpunkt und fahre mit 2.) fort; ansonsten fahre mit s_0 als Startpunkt fort und wähle ein neues s_1 in 2.)

Der große Vorteil dieses Algorithmus ist seine Einfachheit.
Allerdings besitzt Hill Climbing auch eine Reihe von Problemen, so „verfängt“ er sich z.B. sehr leicht in lokalen Optima ohne je das globale Optimum zu erreichen. Der im nächsten Abschnitt diskutierte Simulated Annealing Algorithmus stellt eine Verbesserung dieses Algorithmus dar, der in der Lage ist dieses Problem zumindest teilweise zu umgehen.
Zum Auftakt des Projektes wurde dieser Algorithmus, zum besseren Verständnis der Zusammenhänge, gemeinsam mit dem Projektleiter in das DGPFs implementiert.
Die Implementierung, zusammen mit den üblichen Klassen und Methoden zur verteilten Ausführung, befindet sich in dem Package *org.dgpf.search.algorithms.hc* und seinen Unterpackages.

4. Simulated Annealing

4.1 Einleitung:

Der Simulated Annealing Algorithmus ist die Weiterentwicklung eines von Metropolis veröffentlichen Verfahrens zur Berechnung des Verhaltens einzelner Teilchen in einer Schmelze [5].
Die Grundidee dieses Algorithmus kommt folglich aus der Physik und basiert auf der Tatsache, dass sich in einem Metallgitter befindende Atome bei zu schnellem Abkühlen unter Umständen in energetisch ungünstigen Positionen „verfangen“, was zu Material von schlechter Qualität führen kann. Erfolgt die Abkühlung

hingegen verhältnismäßig langsam, so haben die Atome genug Zeit einen Zustand möglichst niedriger Energie einzunehmen. Sie bleiben somit nicht in einem lokalen Minimum (einem Zustand höherer Energie) „hängen“, was die Erzeugung von qualitativ hochwertigem Material erlaubt.
Ohne weiter auf die physikalischen Details eingehen zu wollen, handelt es sich bei Simulated Annealing um einen aus diesem Optimierungsproblem abgeleiteten Algorithmus zur Bestimmung von globalen Minima. Wobei es möglich ist durch geschickte Wahl und Reduzierung der Temperatur die Suche positiv oder unter Umständen auch negativ zu beeinflussen.
Es ist ein besonderes Merkmal dieses Algorithmus, dass er sich selten in lokalen Minima „verfängt“.
Bleiben Algorithmen dagegen leicht in lokalen Optima hängen, so müssen unter Umständen relativ viele Durchläufe mit unterschiedlichen Startpunkten durchgeführt werden um mit Sicherheit sagen zu können, dass es sich bei einem gefundenen Funktionswert auch wirklich um das globale Minimum handelt.
Um dieses Problem zu umgehen, akzeptiert Simulated Annealing mit einer bestimmten Wahrscheinlichkeit auch schlechtere Funktionswerte, was es ermöglicht (mit einer Wahrscheinlichkeit größer 0) aus lokalen Minima wieder herauszufinden. Es ist sogar bewiesen, dass dieses Verfahren für unendliche kleine Änderungen des Temperaturwertes T immer das globale Minimum findet.

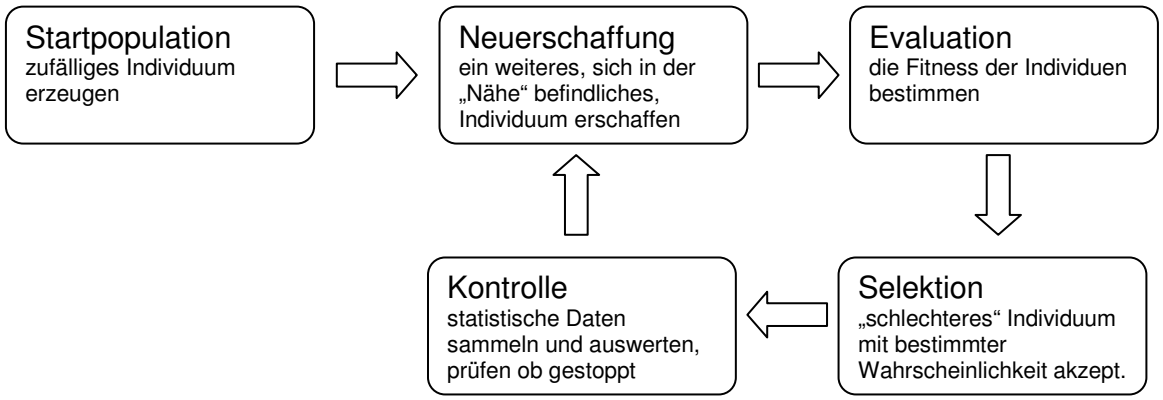

Abbildung 4: Funktionsweise von Simulated Annealing

Für eine Zielfunktion $f : S \to \Re$ arbeitet der Algorithmus folgendermaßen [5]:

1.) Wähle einen Startpunkt $s_0 \in S$.
2.) Wähle $s_1 \in S$ in der Nachbarschaft von s_0.
3.) Falls $f(s_1) \leq f(s_0)$, wähle s_1 als neuen Startpunkt und fahre mit (2.) fort.

Ist $f(s_1) > f(s_0)$, wähle s_1 mit Wahrscheinlichkeit $p = e^{\frac{\Delta f}{T}}$ mit $\Delta f := f(s_1) - f(s_0)$ als neuen Startpunkt, ansonsten behalte s_0 und fahre mit (2.) fort.

Der Wert p gibt also die Wahrscheinlichkeit an mit der der Algorithmus auch schlechtere Werte akzeptiert. Dabei ist p sowohl von der Temperatur T als auch von der Differenz Δf des aktuell berechneten und vorherigen Funktionswertes abhängig, wobei p umso kleiner wird je größer T oder umso kleiner Δf ist. Es ist also umso unwahrscheinlicher, dass der schlechtere Wert s_1 akzeptiert wird je weiter dieser von dem vorherigen Wert s_0 entfernt ist oder je kleiner die Temperatur ist.
Durch die Wahl der Temperatur T ist es also praktisch möglich festzulegen in welchem Maße Verschlechterungen akzeptiert werden.
Anfangs erlaubt man mit einem relativ großen T eine möglichst „weiträumige“ Suche, um diese dann im Laufe der Zeit durch das Absenken des Wertes T auf die erfolgsversprechenden Bereiche zu konzentrieren.
Durch eine geschickte Wahl des Anfangswertes und der Strategie für die Temperaturverringerung kann die Konvergenz der Suche positiv beeinflusst werden.
Bei den Temperaturverringerungsstrategien wird üblicherweise zwischen folgenden Arten unterschieden [5]:

Konstant: $T(n) = k$, wobei k eine beliebige Konstante ist und n die Anzahl der Iterationen

Arithmetisch: $T(n) = T(n-1) - k$, wobei k wieder eine beliebige Konstante ist. Allerdings muss darauf geachtet werden das die Anfangstemperatur $T(0)$ so groß gewählt wird, dass die Temperatur selbst nach der maximalen Anzahl von Iterationsschritten nicht negativ wird.

Geometrisch: $T(n) = \alpha(n) \cdot T(n-1)$, wobei α typischerweise zwischen 0,8 und 0,99 liegt.

4.2 Implementierung

Die Implementierung des Simulated Annealing Algorithmus befindet sich im Package *org.dgpf.search.algorithms.sa*, welches wiederum die folgenden Unterpackages enthält: *org.dgpf.search.algorithms.sa.adaption*, *org.dgpf.search.algorithms.sa.cs*, org.*dgpf.search.algorithms.sa.local*, *org.dgpf.search.algorithms.sa.p2p*, *org.dgpf.search.algorithms.sa.p2p.adaption* und *org.dgpf.search.algorithms.sa.ta*.
Das grundlegende Verhalten des Simulated Annealing Algorithmus, wie in Abschnitt 4.1 beschrieben, wird in der Klasse *SAEngine* implementiert. Zusammen mit der Container-Klasse für den Zustand (*SAStateBag*), ihren dynamischen Parametern (*SAParameters*) und weiteren Helferklassen befindet sie sich im Package *org.dgpf.serach.algorithms.sa.*

Die Klasse *SAEngine* ist von der Klasse *SearchEngine* abgeleitet und stellt die grundlegenden Funktionen zum Initialisieren, Starten und Stoppen der Suche bereit.
Die Klasse *SAParamters* enthält die spezifischen Suchparameter wie die (Start)Temperatur (*m_temperatur* und *m_start_temperatur*) und die Temperaturanpassungsstrategie (*m_temperatur_adaption_strategy*). Die Variable *m_start_temperatur* gibt dabei die zum Startzeitpunkt verwendete Temperatur an, wohingegen die Variable *m_temperatur* die aktuell verwendete Temperatur anzeigt.
Die Temperaturanpassungsstrategie bestimmt dabei, wie bereits erwähnt, wie sich die Temperatur im Laufe der Zeit verändern soll. Alle Temperaturanpassungsstrategien befinden sich im Package *org.dgpf.search.algorithms.sa.ta*.
Im Augenblick stehen die drei bereits oben erläuterten Strategien „konstant“ (*ConstantAdaption*), „arithmetisch“ (*ArithmeticAdaption*) und „geometrisch“ (*GeometricAdapation*) zur Verfügung. Wobei alle drei Klassen von der gemeinsamen Oberklasse *TemperaturAdapationStrategy* abgeleitet sind, die die abstrakte Methode *adapt_temperatur* zur Verfügung stellt. Diese Methode wird nach jedem Update aufgerufen und muss von jeder abgeleiteten Klasse mit dem entsprechenden Code zur Anpassung der Temperatur überschrieben werden.
Neben diesen Variablen enthält die Klasse *SAParameters* noch die üblichen get- und set-Methoden zum Abrufen und Setzen der Temperatur und der Temperaturanpassungsstrategie.
Die Klasse *SAUtil*“ definiert einige Default-Werte wie z.B. einen Default-Temperaturwert (*DEFAULT_TEMPERATUR*) oder einen Default-Wert zur konstanten Temperaturreduktion (*DEFAULT_REDUCE_KONSTANT*).

Im Package *org.dgpf.search.algorithms.sa.adaption* befindet sich nur die von der Klasse *DefaultSearchAdapter* abgeleitete Klasse DefaultSAAdapter, bei der es sich um die Default-Adaptionsstrategie für den Simulated Annealing Algorithmus handelt. Neben der Methode *reset*, welche bei einem Reset die aktuelle Temperatur wieder auf die Starttemperatur zurücksetzt, enthält die Klasse noch die Funktion *adapt* die von der Search Engine zur Anpassung der Suchparameter aufgerufen wird und nach jedem Update die Funktion zur Anpassung des Temperaturwertes aufruft.

Wie bereits in Kapitel 3.1 dargestellt gibt es drei Möglichkeiten, eine Suche mit dem DGPF auf ein Netzwerk zu verteilen. Um diese auch für Simulated Annealing zu übernehmen, wurden von der bereits erwähnten *SAEngine* weitere Klassen abgeleitet.
Im Package *org.dgpf.search.algorthms.sa.cs* befindet sich die von der Klasse *SAEngine* abgeleitete Klasse *CSSAEngine* die eine Client/Server-Implementierung der Search Engine darstellt. Dabei werden die Reproduktion und die Evaluation der Individuen vom Client auf den Server ausgelagert. Die Klasse stellt dazu eine Variable vom Typ *SearchClient* zur Verfügung, die die Verteilung der Rechenlast auf die Server übernimmt. Des Weiteren enthält die

Klasse neben einigen Standardmethoden zum Starten, Stoppen usw. noch die Methode *addServer* zum Hinzufügen von weiteren Servern.

Die Implementierung der Simulated Annealing Engine zur lokalen Ausführung befindet sich im Package org.*dgpf.search.algorithms.sa.local*. Wie die Klasse *CSSAEngine* ist diese Klasse, *LocalSAEngine*, von der Oberklasse *SAEngine* abgeleitet. Neben den üblichen Start- und Stop- Funktionen enthält die Klasse noch ein Array vom Typ *SearchWorkerThread,* welches die Threads enthält die die eigentliche Sucharbeit erledigen. Erzeugt werden die Threads in der *init_threads*-Methode der Klasse.

Das Package *org.dgpf.search.algorithms.sa.p2p* enthält sowohl die Implementierung zur verteilten Berechnung in einem Peer-to-Peer-Netzwerk, als auch eine Implementierung die eine hybride Verteilung erlaubt.
Dazu existiert jeweils sowohl eine spezielle Peer-to-Peer-SearchEngine (*P2PSAEngine*), als auch eine spezielle Engine für die hybride Verteilung (*P2PCSSAEngine*). Des Weiteren existieren speziell angepasste Versionen der Search Parameter (*P2PSAParameters*), des SearchStates (*P2PSAState*) und des SearchStateBag (*P2PSAStateBag*). Neben diesen Klassen enthält das Package noch ein weiteres Unterpackage: *org.dgpf.search.algorithms.sa.p2p.adapation*, welches mit der von der Klasse *DefaultSAAdapter* abgeleiteten Klasse *DefaultP2PSAAdapter* eine spezielle Klasse für die verwendete Adaptionsstrategie im Peer-to-Peer-Betrieb besitzt. Die Klasse besitzt eine Variable vom Typ *P2PadaptionStrategy*, die auf die aktuell verwendete Adaptionsstrategie verweist, wobei als Default der *DefaultP2Padapter* verwendet wird.

Wie schon erwähnt befinden sich im *org.dgpf.search.algorithms.sa.ta* Package die von uns implementierten Temperaturanpassungsstrategien, wobei es durch das Ableiten weitere Klassen von der *TemperaturAdaptionStrategy*-Klasse möglich ist weitere hinzuzufügen.

Um die Implementierung des Simulated Annealing Algorithmus zu testen haben wir mehrere Test-Dateien erstellt. So befinden sich im Package *examples.gp.automaton.gcd* jeweils eine Test-Datei für die lokale, für die Peer-to-Peer und für die hybride Suche nach einem Programm zur Berechnung des größten gemeinsamen Teilers zweier Zahlen.
In den Packages *examples.search.numeric.singleobjective* und *examples.search.numeric.multiobjective* befinden sich ebenfalls jeweils drei Test-Dateien (lokal, P2P und hybrid) zur Berechnung relativ einfacher mathematischer Funktionen.
Des Weiteren befinden sich im Package *examples.search.numeric2.tests* eine Reihe von Tests zur Berechnung von, ebenfalls von uns implementierten, etwas komplizierteren mathematischen Funktionen [4]. Im Package *examples.search.soduko* schließlich befindet sich noch eine Test-Implementierung zur Lösung des beliebten Sudoku-Rätsels [4].

Als Beispiel soll hier auf die „boshafte Schwefelfunktion“ (*BoshafteSchwefelfunktion*) aus dem *examples.search.numeric2* Package eingegangen werden, die sich, wie sich herausgestellt hat, besonders effizient mit dem Simulated Annealing Algorithmus lösen lässt.
Die „boshafte Schwefelfunktion“ ist eine relativ schwierig zu handhabende Funktion, da sie viele lokale Minima mit fast gleichen Funktionswerten besitzt. Die ursprüngliche Funktionsgleichung sieht folgendermaßen aus:

$$f(\vec{x}) = -\sum_{i=1}^{n} x_i \sin(\sqrt{|x|})$$

mit $-500 \le x_i \le 500$

Zur eigentlichen Berechnung muss die Funktion noch etwas umgeformt werden, was allerdings hier nicht weiter von Bedeutung ist. Für eine detaillierte Erklärung siehe [4].

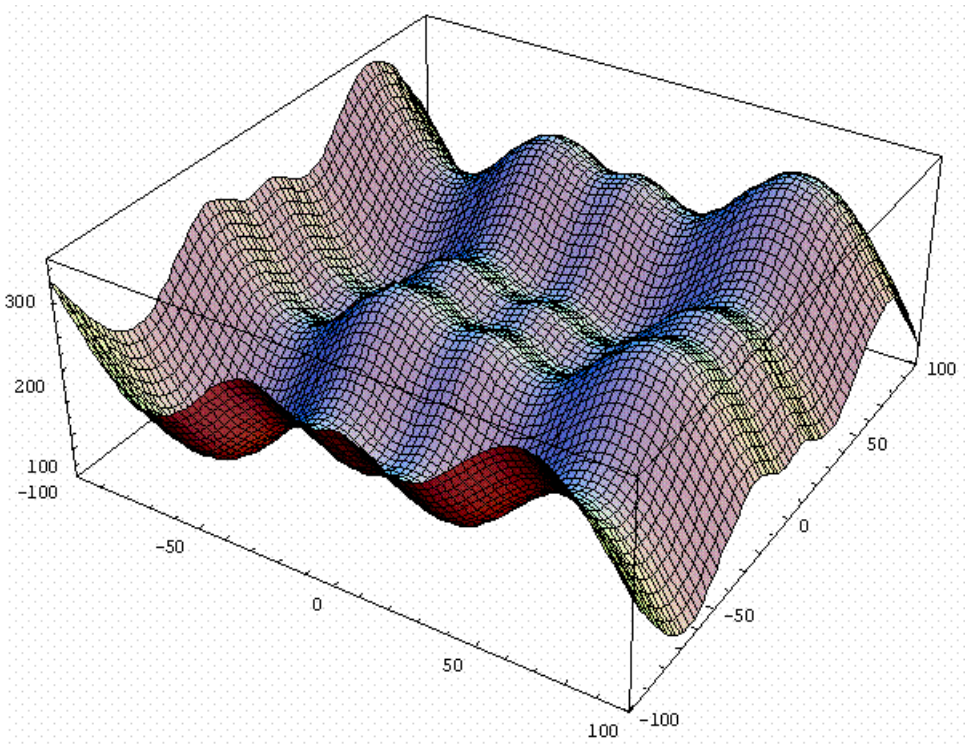

Abbildung 5: Abgeänderte boshafte Schwefelfunktion

Führt man die Berechnung einige Male jeweils mit dem Genetischen Algorithmus, mit dem Hill Climbing Algorithmus und dem Simulated Annealing Algorithmus aus und berechnet den Durchschnitt, so ergibt sich mit 10 Veränderlichen auf einem 1,5 GHz Athlon folgendes Ergebnis [4]:

Genetischer Algorithmus:	1,083 s
Simulated Annealing:	0,318 s

Hill Climbing: 0,389 s

Als Komperator wurde der Sum-Komperator aus dem Package *org.dgpf.search.api.comperator* verwendet. Der Sum-Komperator addiert alle Fitnesswerte eines Individuums und vergleicht diesen Wert mit den, auf die gleiche Art und Weise, berechneten Fitnesswerten der anderen Individuen.
Anhand der Ergebnisse der drei Algorithmen ist ersichtlich, dass je nach zu lösendem Problem der Simulated Annealing Algorithmus unter Umständen sehr effizient sein kann.
Natürlich kann aus diesem einem Test nicht geschlossen werden dass der Simulated Annealing Algorithmus generell effizienter ist als Genetische Algorithmen es sind. Er zeigt aber das es durchaus Probleme gibt die mittels des Simulated Annealing Algorithmus effizient lösbar sind.

Zusammenfassung

Genetische Algorithmen basieren auf den grundlegendsten Merkmalen der Evolution (Anpassung und natürliche Auslese) und orientieren sich an dessen Ablauf. Das Distributed Genetic Programming Framework (DGPF) ist ein auf diesen Genetischen Algorithmen basierendes Framework zur automatischen Erzeugung von Programmcode für bestimmte Problemstellung.
Neben den Genetischen Algorithmen verwendet das DGPF aber auch andere heuristische Suchverfahren wie z.B. den Hill Climbing Algorithmus oder den von mir implementierten Simulated Annealing Algorithmus.
Der Simulated Annealing Algorithmus ist eine Verbesserung des Hill Climbing Algorithmus, da er nicht notwendigerweise in lokalen Minima „hängen“ bleibt.
Verschiedene Tests mit dem DGPF haben bewiesen, dass der Simulated Annealing Algorithmus die Genetischen Algorithmen und den Hill Climbing Algorithmus für bestimmte Problemstellung an Effizienz übertrifft.

Abbildungsverzeichnis

Literaturverzeichnis

[1] http://dgpf.sourceforge.net/
[2] http://sourceforge.net/projects/dgpf/
[3] Thomas Weise. Genetic Programming for Sensor Networks
[4] Stefan Niemczyk. Lösen mathematischer Funktionen und Sudokus mit Hilfe des DGPF
[5] Ingrid Gerdes, Frank Klawonn, Rudolf Kruse. Evolutionäre Algorithmen
[6] Thomas Weise. Distributed Genetic Programming Framework. The DGPF-Project. A presentation for interested students.
[7] http://www.eberl.net/chaos/Eberl/node50.html