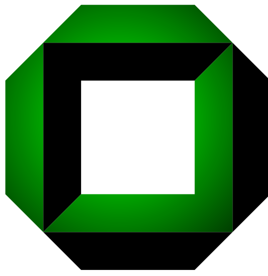


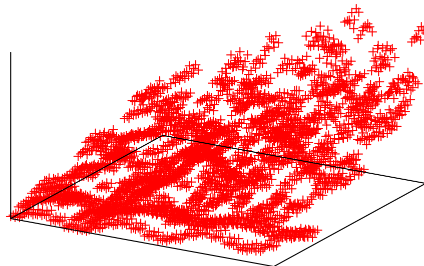
Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Tichy

Evaluation von Algorithmen zur automatischen Performanzoptimierung von parallelen Anwendungen

Studienarbeit von Andreas Zwinkau

15. September 2009



Betreuer:
Dr. Victor Pankratius

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Ort, Datum	Andreas Zwinkau

Auto-Tuning ist das Optimieren von Programmparametern um die Ausführungszeit eines Programms zu beschleunigen. Diese Arbeit implementiert bekannte Optimierungsalgorithmen und überprüft sie experimentell auf ihre Tauglichkeit zum Auto-Tuning von parallelen Anwendungen. Die Arbeit baut auf Autuner auf, einer Erweiterung der Eclipse-Entwicklungsumgebung, die in der Young Investigator Group "Software Engineering für Multicore-Systeme" entwickelt wurde. Zur Evaluation der Optimierungsalgorithmen wird eine Sammlung synthetischer Programme entwickelt; die Ergebnisse der Optimierung werden mit Hilfe bekannter Ergebnisse aus Testprogrammsammlungen überprüft.

Inhaltsverzeichnis

1. Einleitung 9
1.1. Problemstellung 9
1.2. Lösungsansatz 10
1.3. Gliederung 10

2. Grundlagen 11
2.1. Auto-Tuning 11
2.2. Optimierung 11
2.3. Autuner Plug-In 12
2.4. Autuner Rahmenwerk 14

3. Algorithmen 16
3.1. Vergleichsalgorithmen 16
3.1.1. Lineare Suche 16
3.1.2. Zufall 17
3.2. Implementierte Algorithmen 17
3.2.1. Simplex 17
3.2.2. Polytope 20
3.2.3. Evolution 20
3.2.4. Differenzevolution 21
3.2.5. Partikelschwarm 22
3.3. Algorithmenkonfiguration 24
3.3.1. Synthetische Programme 24
3.3.2. Algorithmenparameter 26

4. Evaluation 27
4.1. Bewertung mit synthetischen Programmen 27
4.2. Testprogramme 30
4.2.1. PARSEC 30
4.2.2. SPEC OMP 30
4.2.3. Weitere Sammlungen 31
4.3. Bewertung mit realen Programmen 31
4.4. Bewertung 34

5. Zusammenfassung 35
5.1. Ausblick 35

5.2. Zusammenfassung 36

A. Auswertungsdaten des Algorithmen-Tuning 37

B. Beispielcode 42

Literaturverzeichnis 43

1. Einleitung

Softwaresysteme werden zunehmend komplexer. Während vor 40 Jahren ein Programm aus mehreren tausend Zeilen Code als „groß“ galt, bezeichnet man heutzutage Projekte erst ab mehreren Millionen Zeilen Code als „groß“. Die Schwierigkeit der Entwicklung solcher Systeme besteht zu einem bedeutenden Anteil in der Tatsache, dass ein Mensch nicht in der Lage ist allen Code zu kennen. Um die Komplexität zu verringern, werden Module voneinander abgetrennt und standardisierte Schnittstellen eingeführt. Doch manche Phänomene moderner Software entstehen aus dem Zusammenspiel von Komponenten.

Ein Beispiel für diese Phänomene sind die Laufzeiten nebenläufiger Programme. Bei kurzen Laufzeitmessungen dominieren die Kosten für das Erstellen und Verwalten von Fäden[1], was zu starken, indeterministischen Schwankungen in den Laufzeiten führt. Um dieser Messungenauigkeit entgegenzuwirken sind längere Laufzeiten im Bereich von Minuten oder gar Stunden notwendig. Zusätzlich besitzt ein solches Program oft eine Menge von Einstellungsmöglichkeiten, so dass ein großer Raum von möglichen Konfigurationen besteht. Das automatisierte Optimieren solcher Konfigurationen wird als Auto-Tuning bezeichnet. Das Autuner Werkzeug, das in der Young Investigator Group „Software Engineering für Multicore-Systeme“ entwickelt wird, hilft dem Entwickler bei diesem Prozess.

1.1. Problemstellung

Da es im Allgemeinen nicht wirtschaftlich ist, alle Konfigurationen zu testen um das Optimum zu finden, gilt es einen heuristischen Such- bzw. Optimierungsalgorithmus zu verwenden, so dass in möglichst kurzer Zeit eine möglichst gute Konfiguration gefunden wird. Dabei ist abzuwägen zwischen Suchzeit und Ergebnisqualität. Das Verhalten von Laufzeiten folgt keinem Schema, so dass sehr allgemeine Algorithmen notwendig und bisher keine problem-spezifischen Verbesserungen bekannt sind. Doch welche Algorithmen eignen sich in diesem Gebiet?

[1] engl.: Threads, in dieser Arbeit sind immer Fäden gemeint, die vom Betriebsystemkern verwaltet werden.

1.2. Lösungsansatz

Diese Arbeit stellt drei grundlegende Optimierungsalgorithmen und zwei Varianten vor, die zum Auto-Tuning benutzt werden können. Zwei Qualitätskriterien lassen sich zur Bewertung der Algorithmen anwenden: Die Ergebnisqualität, also wie nahe am Optimum eine Konfiguration ist, und die Geschwindigkeit, also wie schnell das Ergebnis gefunden wurde. Um eine aussagekräftige Bewertung vornehmen zu können, müssen die beiden Kriterien getrennt voneinander betrachtet werden.

Der Anwendungsbereich Auto-Tuning verlangt eine möglichst große Testsammlung wie Parsec als Grundlage. Dagegen abzuwägen ist allerdings die Auswertungsgeschwindigkeit, denn eine Optimierung erfordert mehrere Tage, so dass eine Evaluation an synthetischen Laufzeitdaten attraktiv erscheint. In dieser Arbeit wird eine ausführliche Bewertung anhand eines synthetischen Datensatzes vorgenommen, bei dem auch eine lineare Suche möglich ist. Anschließend soll das Evaluationsergebnis an realistischen Programmen bestätigt werden.

Als Rahmenwerk zur Implementierung und Auswertung der Algorithmen bietet sich das Autuner Eclipse-Plug-In an. Es bietet umfangreiche Einstellmöglichkeiten für verschiedene Auto-Tuning-Bedürfnisse und erlaubt das Integrieren eigener Optimierungsalgorithmen.

1.3. Gliederung

In Kapitel 2 werden zuerst einige Grundlagen eingeführt. Die fünf Optimierungsalgorithmen werden in Kapitel 3 vorgestellt, im Autuner-Rahmenwerk implementiert und konfiguriert. Zur Evaluierung in Kapitel 4 wird anhand einer ausgewählten Anzahl von Funktionen eine Bewertung der Algorithmen für sich und im Vergleich zueinander vorgenommen. Zuletzt wird dieser Vergleich in einer Auswertung mit realistischen Programmen aus Parsec betrachtet. In Kapitel 5 befindet sich abschließend eine Zusammenfassung der Arbeit.

2. Grundlagen

Dieses Kapitel führt in grundlegende Begriffe ein, die notwendig für die weitere Lektüre sind.

2.1. Auto-Tuning

Auto-Tuning ist das automatische Optimieren von Programmparametern zur Verringerung der Laufzeit. Die Zahl der möglichen Konfigurationen für moderne Software ist oft nicht mehr überschaubar und eine analytische Optimierung meist nicht machbar. Es macht daher Sinn die Suche nach den optimalen Parametern zu automatisieren und durch empirische Optimierung zu konfigurieren.

Man unterscheidet zwischen Online- und Offline-Tuning. Offline-Tuning bezeichnet das Suchen statisch-optimaler Parameter, während Online-Tuning eine dynamische Optimierung zur Laufzeit bedeutet. Nehmen wir als Beispiel einen Webserver an der eine Zahl von Arbeiterprozessen n besitzt, die Browser-Anfragen bearbeiten. Offline-Tuning ist, wenn man durch Erfahrung und Testen herausfindet, dass $n = 4$ im Durchschnitt eine bessere Leistung zeigt als andere Werte für n. Online-Tuning ist, wenn der Webserver je nach Prozessorlast Arbeiterprozesse hinzufügt oder entfernt, also n während der Laufzeit angepasst wird.

2.2. Optimierung

Beim Auto-Tuning soll die Laufzeit eines Programms minimiert werden, indem Parameter entsprechend eingestellt werden. Die Laufzeit ist nicht das einzig mögliche Optimierungsziel (Energieverbrauch wäre beispielsweise eine weitere Möglichkeit), aber das Häufigste. Andere Ziele würden für die weitere Betrachtung keinen Unterschied machen.

Zwar sind die Parameter von verschiedenem Typ (int, float, string, ...), allerdings sind alle diskret und endlich. Diskretheit und Endlichkeit folgen aus der Tatsache, dass Autuner das Setzen von Bereichen wie „1 bis 16“ fordert. Wir können also alle

möglichen Parameter nummerieren und deren Index $i \in \{0, \ldots, n\}$ nutzen, womit wir eine Programmkonfiguration bestehend aus k Parametern als Vektor $x \in \mathbb{N}^k$ modellieren können. Die Laufzeitmessung des Programms mit konkreten Parametern lässt sich als Funktion $t : \mathbb{N}^k \to \mathbb{R}$ darstellen. Das Optimierungsproblem stellt sich nun als Minimierungsproblem dar:

$$\min(t(x)) \text{ mit } x \in \mathbb{N}^k$$

Eine Parameterbelegung x wird im weiteren als eine „Konfiguration“ bezeichnen. Da Konfigurationen Elemente des Vektorraums $\mathbb{N}^k$ sind, ist die Minimierung von t äquivalent zu einer Suche nach dem kleinsten Element im Raum $\mathbb{N}^k$, wobei der Vergleich zweier Konfigurationen x und y auf ihre „Größe“ durch die Laufzeit bestimmt sei:

$$x < y \Leftrightarrow t(x) < t(y)$$

Das Problem beim Auto-Tuning ist einerseits, dass die Auswertung von $t(x)$ mit einem Programmdurchlauf verbunden ist, der Minuten wenn nicht Stunden dauern kann. Man möchte also möglichst wenige Auswertungen durchführen, um in akzeptabler Zeit ein Ergebnis zu erhalten. Andererseits hängt die Auswertung stark von der Umgebung ab, was dazu führt, dass die Messungen „verrauscht“ sind und viele lokale Minima existieren, was die Optimierung erschwert. Ein effizienter Algorithmus muss also robust gegenüber lokalen Extrema sein und gleichzeitig sehr schnell zu einem Ergebnis kommen. Schnelligkeit hat allerdings zur Folge, dass kein optimales Ergebnis mehr gefunden werden kann. Das heißt wir haben es in jedem Fall mit einer heuristischen Suche zu tun.

Die Funktion t kann nicht differenziert werden, was bestimmte Verfahren wie das Gradientenverfahren oder das Newton-Verfahren aussschließt. Auch klassische Suchstrategien, wie die binäre Suche, lassen sich nicht anwenden, da notwendige Annahmen für t nicht getroffen werden können. Man beginnt also mit einer mehr oder weniger zufällig gewählten Startkonfiguration. Nun gilt es eine möglichst gute Strategie zu wählen, mit der weitere Konfigurationen so bestimmt werden, dass die Laufzeit immer weiter minimiert wird. Zur Auswahl einer weiteren Konfiguration stehen nur die Auswertungen vorhergegangener Versuche zur Verfügung, sowie Kenntnisse über die Grenzen des Suchraums.

2.3. Autuner Plug-In

Autuner ist ein Eclipse-Plug-In das vor Beginn der Studienarbeit am Institut für Programmstrukturen und Datenorganisation (IPD) von der Young Investigator Group: Software Engineering für Multicore-Systeme entwickelt wurde. Es soll Entwickler bei der Entwicklung von komplexen, parallelen bzw. nebenläufigen Anwendungen durch automatisierte Optimierung unterstützen.

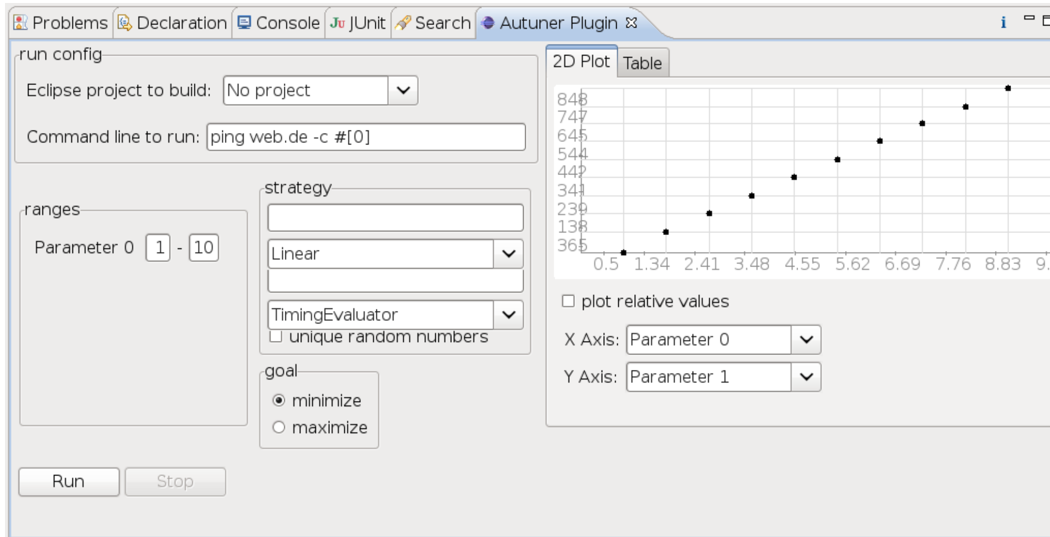


Abbildung 2.1.: Bildschirmfoto des Autuner Plug-In

In Abbildung 2.1 ist die Autuner-Oberfläche in Eclipse zu sehen. Im Bereich „run config“ wird das auszuführende Programm angegeben. In diesem Beispiel wird das Kommandozeilenwerkzeug `ping` aufgerufen eine bestimmte Anzahl an Anfragen an `web.de` abzuschicken. Bei `ping` kann durch den Parameter `-c` die Anzahl an Anfragen angegeben werden. Das Kürzel `#[0]` zeigt Autuner, dass an dieser Stelle der Wert von Parameter 0 einzufügen ist. Autuner kann zusätzlich Eclipse-Projekte bauen, falls diese ausgeführt bzw. optimiert werden sollen, allerdings wird diese Funktionalität in dieser Arbeit nicht benötigt. Die Anzahl der `ping`-Anfragen ist der einzige Parameter in diesem Fall, der wie unter „ranges“ angegeben den Wertebereich 1-10 annehmen kann. Im Bereich „strategy“ ist in diesem Beispiel angegeben, dass eine lineare Suche („Linear“) verwendet und die Ausführungszeit („TimingEvaluator“) gemessen werden soll. Optimiert wird im Hinblick auf ein Minimum, wie unter „goal“ angegeben ist.

Auf der rechten Seite ist ein Diagramm der Auswertung zu sehen. Wie man erwartet verhält sich die Laufzeit von `ping` linear zur Anzahl der Anfragen. Alternativ können die Messdaten auch als Tabelle dargestellt und im CSV-Format exportiert werden.

Autuner kann auch von der Kommandozeile ausgeführt werden. Ein Beispielaufruf äquivalent zum Bildschirmfotobeispiel wäre:

```
java -jar autuner.jar "ping web.de -c #[0]" -p=1-10 -a=Linear
```

Da hierbei die Messdaten direkt ausgegeben werden, eignet sich diese Methode sehr gut, um die Daten anschließend mit anderen Programmen weiterzuverarbeiten.

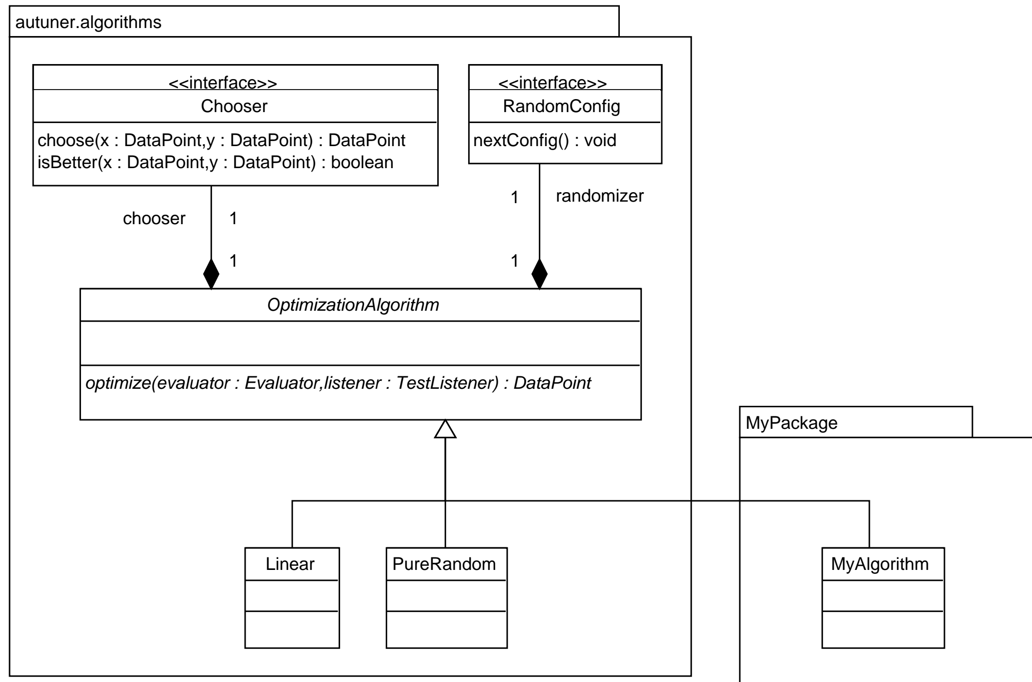


Abbildung 2.2.: Klassendiagramm zu Optimierungsalgorithmen

2.4. Autuner Rahmenwerk

Autuner ist auch ein Rahmenwerk in dem Optimierungsalgorithmen implementiert werden können. Über das Plug-In lassen sich diese dann zum Auto-Tuning verwenden.

Jeder Algorithmus muss von der abstrakten Oberklasse `OptimizationAlgorithm` erben wie in Abbildung 2.2 zu sehen [1]. Die Algorithmen `Linear` und `PureRandom` sind in Autuner integriert, entsprechen den Algorithmen „Lineare Suche“ und „Zufall“ in Abschnitt 3.1 und können als Java-Code in Fragment B.1 und Fragment B.2 betrachtet werden. Außerdem ist beispielhaft der Algorithmus `MyAlgorithm` in `MyPackage` abgebildet, um anzudeuten, dass beliebige Unterklassen verwendet werden können ohne Einschränkungen an Name oder Paket.

Die abstrakte Methode, die von der Unterklasse implementiert werden muss, ist `optimize`. Die `optimize` Methode gibt ein `DataPoint`-Objekt zurück, welches die Konfiguration und die dazu gemessene Ausführungszeit beinhaltet. Zur Auswertung wird ein `Evaluator` übergeben, der das auszuführende Programm kapselt, denn die

[1] Das Modell wurde auf den zum Verständnis essentiellen Teil gekürzt. Beispielsweise sind die Attribute `private` und über Getter/Setter-Methoden zu nutzen.

Ausführung und Messung selbst ist nicht Aufgabe des Algorithmus. Das übergebene `TestListener`-Objekt ist zur Beobachtung das Ablaufs vorhanden, so wird dadurch unter anderem die Plug-In-GUI aktuell gehalten. Das verlangt vom Entwickler eines Algorithmus, dass er darauf achtet diese Listener-Aufrufe zu machen.

Zur Implementierung des Algorithmus sollte das Attribut `chooser` verwendet werden, damit nicht festgelegt ist, ob Minimum oder Maximum gesucht wird. Anhand der `choose`-Methode kann zwischen zwei `DataPoint`-Objekten das „bessere“ ausgewählt werden und mit der `isBetter`-Methode kann geprüft werden, ob ein `DataPoint`-Objekt „besser“ ist als ein Anderes. Ebenso steht `randomizer` zur Verfügung, um durch die `nextConfig`-Methode zufällige Konfiguration zu erzeugen. Die beiden orthogonalen Attribute `chooser` und `randomizer` werden vorinitialisiert bzw. können über Parameter oder GUI konfiguriert werden.

3. Algorithmen

In diesem Kapitel werden die implementierten Suchalgorithmen vorgestellt. Außerdem wurde eine Testumgebung aus synthetischen Programmen entwickelt, um die Parameter der Algorithmen zu konfigurieren.

3.1. Vergleichsalgorithmen

Die zwei folgenden Algorithmen „Lineare Suche“ und „Zufall“ sind in Autuner bereits integriert und dienen vor allem zu Vergleichszwecken in Kapitel 4.

3.1.1. Lineare Suche

Der einfachste Algorithmus ist die Enummeration und Auswertung aller Parameterkonfigurationen, wie in Algorithmus 1 gezeigt. Jede mögliche Konfiguration x wird ausgewertet und mit dem bisherigen Minimum b verglichen.

Algorithmus 1 Lineare Suche

for $x \in \mathbb{N}^n$ **do**
    if $t(x) < t(b)$ **then**
        $b \leftarrow x$
return b

Dieser Algorithmus hat die schlechteste Ausführungszeit von allen betrachteten Algorithmen, garantiert aber, dass die optimale Konfiguration gefunden wird. Ein Suchalgorithmus der im Durchschnitt länger braucht als lineare Suche ist nicht akzeptabel, da kein besseres Ergebnis erreicht werden kann. Hiermit ist also die Obergrenze der Suchzeit festgelegt. Als Variante zu diesem Algorithmus liegt die Möglichkeit nahe eine größere Schrittweite zu verwenden, wodurch das Ergebnis ungenauer wird, aber auch die Laufzeit nur einen Bruchteil beträgt. Dieser Ansatz wird allerdings immer noch langsam sein, da lineare Suche in jedem Fall in der Komplexitätklasse $O(n)$ liegt, während andere Algorithmen sich in $O(\log(n))$ befinden.

3.1.2. Zufall

Ein weiterer einfacher Algorithmus ist das Auswerten einer Menge von Zufallskonfigurationen, wie in Algorithmus 2 gezeigt. Es wird eine Reihe von n zufälligen Stichproben x gemacht und die Beste b als Optimum zurückgegeben.

Algorithmus 2 Zufall

```
for n iterations do
    x ← random configuration
    if t(x) < t(b) then
        b ← x
return b
```

Ein Suchalgorithmus der im Durchschnitt unter der Bedingung, dass die gleiche Anzahl an Auswertungen vorgenommen wurde, ein schlechteres Ergebnis als dieser Zufallsalgorithmus liefert ist nicht akzeptabel. Die Aufgabe einer Suchstrategie ist es einen sinnvollen nächsten Schritt auszuwählen und diese Auswahl sollte geschickter sein, als eine zufällige Auswahl. Hiermit ist also die Untergrenze der Ergebnisqualität festgelegt.

3.2. Implementierte Algorithmen

Betrachten wir nun weitere Optimierungsalgorithmen, die im Bereich der Optimierungstheorie bekannt sind. Aus Zeitgründen beschränken wir uns hier auf relativ grundlegende Algorithmen, ohne viele Untervarianten zu berücksichtigen.

3.2.1. Simplex

Das Downhill-Simplex-Verfahren von Nelder und Mead[BI91] basiert auf der Idee, dass ein Körper sich durch den Suchraum bewegt. Ein Simplex s ist der einfachste Körper für eine gegebene Dimensionalität d, also beispielsweise ein Dreieck in einem 2-dimensionalen Raum.

Das Verfahren besitzt drei Parameter α, β und γ und ist in Algorithmus 3 formal beschrieben. In jedem Schritt des Algorithmus wird eine der Ecken des Körpers versetzt. Dazu wird ein neuer Punkt n bestimmt, indem die schlechteste Ecke mit einem Skalierungsfaktor α über den Mittelpunkt der anderen Ecken gespiegelt wird (Reflexion). Als Beispiel sei der erste Schritt in Abbildung 3.2 zu betrachten, wobei Knoten b die schlechteste Ecke ist. Nun können drei Fälle eintreten:

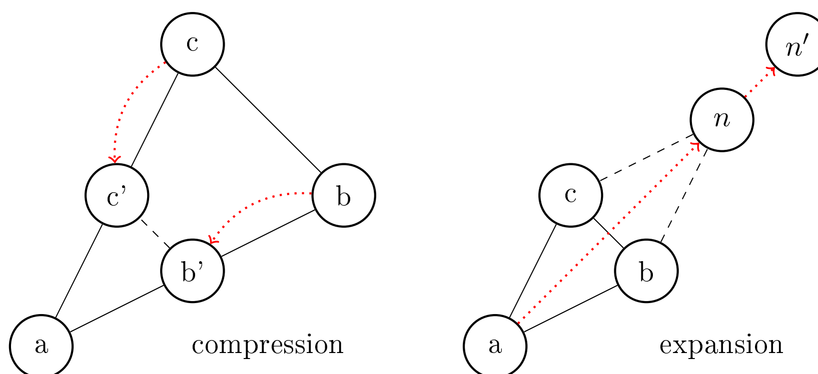


Abbildung 3.1.: Komprimierung und Expansion im Simplexalgorithmus

1. Der neue Punkt n ist schlechter als alle Ecken in s ($t(n) > t(\text{worst}(s))$), dann wird der neue Punkt um den Faktor β in Richtung des besten Werts verschoben (Kontraktion). Falls dieser Wert $t(n)$ wieder schlechter ist als alle Anderen, wird der Simplex um den besten Punkt herum mit den Faktor β geschrumpft (Komprimierung, siehe Abbildung 3.1).

2. Der neue Punkt n ist besser als alle Ecken in s, dann wird der Schritt in dieser Richtung um den Faktor γ vergrößert (Expansion, siehe Abbildung 3.1).

3. Ansonsten ersetzt dieser neue Punkt n den bisher schlechtesten Punkt in s.

Als Indikator für die Abbruchbedingung (auch Konvergenzkriterium genannt) schlagen Nelder und Mead den Standardfehler σ_T vor, wobei $T = t(s)$ die Zufallsvariable der Auswertungen der Konfigurationen der Simplexpunkte sei, also:

$$\sigma_T = \frac{\sigma}{\sqrt{|s|}} = \sqrt{\frac{\text{Var}(T)}{|s|}} = \sqrt{\sum_{i=1}^{|s|} \frac{\left(t(s_i) - \overline{t(s)}\right)^2}{|s|}}$$

Die Abbruchbedingung wäre damit $a \leq \sigma_T$, wobei die Grenze a frei zu wählen ist.

Da der Suchraum in Autuner diskret ist, könnte diese Abbruchbedingung allerdings unerreichbar sein. Auch könnte eine „Ebene“ oder „Stufe“[1] im Suchraum den Algorithmus zu früh beenden. Alternativ eignet sich stattdessen die Summe der Abstände der Simplexpunkte zum Besten, die wir als u bezeichnen wollen. Der Algorithmus wird ein (möglicherweise lokales) Minimum erreichen und dort das Simplex immer weiter komprimieren. Bei diskreter Wertemenge gibt es dabei natürlicherweise eine Grenze, da keine infinitesimalen Seitenlängen möglich sind. Aus diesem Grund wird der Algorithmus irgendwann zu einem Zustand kommen, in dem u kleiner als das Produkt aus Eckenzahl des Simplex und Dimension ist. Also ist $u > |s| \cdot d$ eine

[1] Der Vergleich ist anschaulich, aber nur für eine 2-dimensionalen Suchraum angebracht

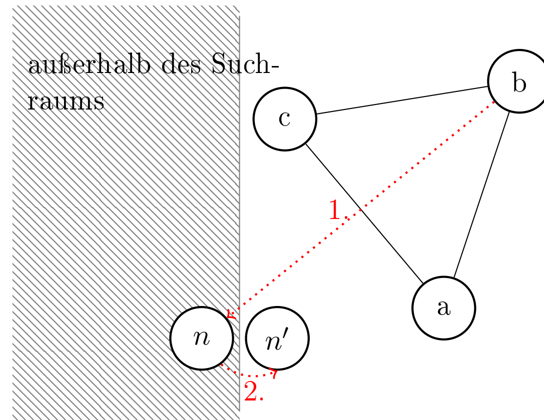


Abbildung 3.2.: Reflexion im Simplexalgorithmus

geeignete Schleifenbedingung.

Algorithmus 3 Simplex

$s \in (\mathbb{N}^n)^{n+1}$
$n \leftarrow \text{reflexion}_\alpha(s)$
while $u > |s| \cdot d$ **do**
  if $t(n) > t(\text{worst}(s))$ **then**
    $n \leftarrow n + \beta \cdot (\text{best}(s) - n)$
    if $t(n) > t(\text{worst}(s))$ **then**
      $\text{compress}_\beta(s)$
      $n \leftarrow \text{reflexion}_\alpha(s)$
  else
    if $t(n) < t(\text{best}(s))$ **then**
      $n \leftarrow n + \gamma \cdot (n - \text{worst}(s))$
    else
      $s \leftarrow (s \cup \{n\}) \setminus \{\text{worst}(s)\}$
      $n \leftarrow \text{reflexion}_\alpha(s)$
return m

Eine Besonderheit in unserem Fall ist die Einschränkung, dass keine Punkte außerhalb des definierten Bereichs gewählt werden dürfen, aber durch Reflexion und Expansion könnte eine Grenze überschritten werden. Die Autoren schlagen vor diesen ungültigen Konfigurationen eine hinreichend große Laufzeit zuzuweisen, so dass ein Übertritt eine Komprimierung zur Folge hat und der Simplex zurück in den definierten Bereich kommt. Da das Autuner-Rahmenwerk dies nicht erlaubt, muss ein Übertritt verhindert werden indem neue Punkte bei Bedarf in den gültigen Bereich „geschoben“ werden bevor sie in s eingehen. Ein Beispiel ist in Abbildung 3.2 zu sehen, wo das neue Eck n zurück in den gültigen Bereich zu n' gesetzt wird. Ein

Folgeproblem dieser Maßnahme ist die Möglichkeit, dass sich Endloszyklen bilden, falls der zurückgeschobene Punkt n' bereits in s vorhanden ist, also beispielsweise $n' = a$. Um diese Schleife zu vermeiden, wird ein zurückgeschobener Punkt leicht in eine zufällige Richtung versetzt.

3.2.2. Polytope

Eine kleine Variation des Algorithmus ist es ein Polytop, statt nur ein Simplex zu benutzen, so dass $|s| > d + 1$ ist. Dies verbessert [TCC+09] das Ergebnis, weshalb in der Evaluation der „Polytope"-Algorithmus mit $|s| = 4d$ betrachtet wird. Weitere Änderungen am Simplexalgorithmus sind nicht notwendig.

3.2.3. Evolution

Evolution bezeichnet ursprünglich eine wissenschaftliche Theorie über die Entstehung der Arten, wie Darwin sie im 19. Jahrhundert beschrieb. Heute lässt sich Evolution beispielsweise definieren als „die Veränderung der vererbbaren Merkmale einer Population von Lebewesen von Generation zu Generation". Der Mechanismus der Evolution besteht aus den drei Teilen Mutation, Selektion und Retention. Dieser Mechanismus lässt sich verallgemeinern und zur Optimierung[Rec73] verwenden. Dazu wird eine Fitnessfunktion aufgestellt, die jedem Individuum in der Population aufgrund seiner Merkmale eine Überlebensstärke zuweist. Durch Mutation entstehen Individuen mit veränderten Merkmalen und durch Selektion verschwinden Individuen mit schlechter Überlebensstärke. Retention bezeichnet das Phänomen, dass Merkmale über die Generationen hinweg trotz Mutation erhalten bleiben.

Eine Konfiguration entspricht einem Individuum der Population und die Menge der gerade betrachteten Konfigurationen ist damit die Population. Die Auswertungsfunktion t übernimmt die Rolle der Fitnessfunktion. Aus einer Anfangspopulation von zufälligen Konfigurationen werden die k Besten ausgewählt (Selektion) und daraus neue Konfigurationen erzeugt durch Mischen und kleine zufällige Änderungen (Mutation). Indem sowohl Eltern- als auch Kindgeneration in der nächsten Epoche zur Auswahl stehen, sterben die besten Konfigurationen nie aus (Retention). In Algorithmus 4 ist das Verfahren für eine feste Populationsgröße k dargestellt.

Die Kernfrage eines Evolutionsalgorithmus ist, wie neue Mutationen bestimmt werden. In unserem Fall wird eine Mischung aus den beiden besten Individuen und einem zufälligen Individuum erstellt. Die Mischung ist dabei durch Parameter α,β und γ gegeben, wobei $\alpha + \beta + \gamma = 1$ gilt. So konzentriert sich die Suche auf das Umfeld der besten Individuen.

Algorithmus 4 Evolution

```
p ← (ℕ^n)^k
for g generations do
    p ← selection_k(p ∪ {MUTATION(p)})
return selection_1(p)

procedure MUTATION(p)
    b = best of p
    s = second best of p
    r = random element of p
    return αb + βs + γr
```

Eine Abbruchbedingung ist schwer zu finden, denn selbst wenn keine Verbesserung stattfindet, könnte Fortschritt erzielt werden. Es kann nicht gefordert werden, dass sich alle Individuen irgendwann in einer bestimmten Umgebung befinden, wie das beim Simplexalgorithmus der Fall ist. Da die anderen Algorithmen ungefähr logarithmischen Aufwand im Bezug auf den Suchraum betreiben, verwendet die Implementierung deswegen eine entsprechende Formel, um die Generationenzahl g zu bestimmen: $g = d \cdot \log(1000n)$ wobei d die Dimension des Suchraums und n die Anzahl der Konfigurationen angibt. Die Populationsgröße ist konstant.

3.2.4. Differenzevolution

Ein Variante des Evolutionsalgorithmus ist die Differenzevolution (engl.: Differential Evolution) [SP95], die eine veränderte Mutationsmethode verwendet. Neue Individuen werden, wie in Algorithmus 5 gezeigt, mit Hilfe der Differenz zwischen zwei zufällig gewählten Parametervektoren gebildet. Diese Differenz wird mit dem Faktor F skaliert und zu einem dritten zufällig gewählten Vektor addiert, was folgende Formel ergibt:

$$v = p_1 + F \cdot (p_2 - p_3)$$

Um die Vielfältigkeit weiter zu erhöhen, wird ein vierter Vektor p_4 elementweise mit v vermischt, indem mit Wahrscheinlichkeit α eine Element jeweils aus p_4 verwendet wird, sonst aus v.

$$n = \text{mix}_\alpha(v, p_4)$$

Dieser neue Vektor n geht in die Population p ein und das schlechteste Individuum wird entfernt.

Algorithmus 5 Differenzevolution

$p \leftarrow (\mathbb{N}^n)^k$
for g generations **do**
  $p \leftarrow \text{selection}_k(p \cup \{\text{MUTATION}(p)\})$
return $\text{selection}_1(p)$

procedure MUTATION(p)
  select $p_1, p_2, p_3, p_4 \in p$ randomly
  return $\text{MIX}_\alpha(p_1 + F \cdot (p_2 - p_3), p_4)$

procedure $\text{MIX}_\alpha(x, y)$
  for all i **do**
    $$z_i = \begin{cases} x_i & \text{with probability } \alpha \\ y_i & \text{else} \end{cases}$$
  return z

3.2.5. Partikelschwarm

Grundidee des Partikelschwarmalgorithmus sind Partikel, die sich mit einer gewissen Trägheit durch den Suchraum bewegen und dabei dem globalen und dem eigenen Maximum zustreben. Die ursprüngliche Idee[KE95] basiert auf dem Verhalten von Fisch- und Vogelschwärmen.

Zu Beginn wird eine Menge bzw. ein Schwarm S zufälliger Partikel generiert. Jedem Partikel p sei ein Bewegungsvektor d_p zugeordnet. Weiter sei p^* die beste Konfiguration die p bisher angenommen hat und b die global minimale Konfiguration, so dass gilt $t(b) \leq t(p^*)$. In jeder Iteration i wird nun für jedes Partikel p der Bewegungsvektor angepasst:

$$d_{p,i} = d_{p,i-1} + \alpha(b - p) + \beta(p^* - p)$$

Anhand des neuen Bewegungsvektors wird dann die neue Position berechnet:

$$p_i = p_{i-1} + d_{p,i}$$

Für die Abbruchbedingung ist, wie auch im Evolution-Fall, der Schwarmzustand als Indikator nutzlos, so dass Schrittanzahl und Partikelmenge vorbestimmt werden muss. Die Partikelanzahl ist im logarithmischen Verhältnis zum Suchraum gewählt, während die Schrittanzahl linear zur Zahl der Dimensionen gewählt wurde. Diese Verhältnisse führen zu einem mit den anderen Algorithmen vergleichbaren Aufwand.

Wie bereits bei dem Simplexalgorithmus besteht auch hier die Möglichkeit, dass ein Partikel den Suchraum verlässt. In diesem Fall wird es genauso „zurückgeschoben“.

Algorithmus 6 Partikelschwarm

```
initialize particle swarm S
b ∈ S with t(b) minimal
for k steps do
    for p ∈ S do
        p* is the minimum in ps history
        d_p is ps current movement
        d_p ← d_p + α(b − p) + β(p* − p)
        p ← p + d_p
        if t(p) > t(b) then
            b ← p
return b
```

Da sowohl lokales noch globales Minimum den Bewegungsvektor vom Rand „wegziehen“, kann ein Partikel nur durch eine zwischenzeitig zu hohe Geschwindigkeit aus dem Suchraum „herausschießen“. Auf Dauer streben alle Partikel einem Minimum zu und bleiben dadurch innerhalb des definierten Bereichs.

3.3. Algorithmenkonfiguration

Aus den Beschreibungen der Algorithmen ist ersichtlich, dass diese Parameter besitzen (beispielsweise α und β von Partikelschwarm) deren Werte festzulegen sind. Zwar ist klar, dass beispielsweise ein Wert zwischen 1 und 0 zu wählen ist, doch wir wollen uns nun mit der Frage auseinandersetzen, inwiefern die Parameterwahl optimiert werden kann.

3.3.1. Synthetische Programme

Als Bewertungsgrundlage werden Testprogramme benötigt, deren Parameter mit Autuner optimiert werden können. Statt echte Programme ablaufen zu lassen und deren Laufzeit zu messen, kann man Optimierungsalgorithmen auch auf Funktionen anwenden, die zu Konfigurationen eine entsprechende Laufzeit berechnen. Das erlaubt ein schnelleres Testen, da keine wirkliche Programmlaufzeit ablaufen muss. So kann eine lineare Suche verwendet werden und es wird jedes Optimum gefunden. Es lassen sich verschiedene Funktionen verwenden, welche die Verläufe von echten Laufzeitverteilungen modellieren.

Es wurde eine Menge von sechs einfachen Funktionen erstellt, die in Abbildung 3.3 zu sehen sind. Die Auswahl der Funktionen ist einerseits inspiriert von De Jongs[DJ75] fünf Funktionen (Sphäre, Rosenbrocks Sattel, Stufen, Biquadratische Funktion mit Rauschen und Schekels Fuchsbauten), welche die wichtigsten Charakteristika zum Vergleich von Optimierungsalgorithmen beinhalten. Andererseits stützt sich die Auswahl auf Funktionsmuster, wie sie üblicherweise beim Auto-Tuning beobachtet werden können.

Jede Funktion (außer Holes) ist jeweils für alle Dimensionen gleich, so dass sich der mehrdimensionale Fall durch das Produkt der eindimensionalen Funktionen berechnen lässt:

$$f(x_1, \cdots, x_n) = \prod_{i=1}^{n} f(x_i)$$

Die Funktionen beinhalten die bekannten Formen einer Hyperbel, einer Parabel und eines Logarithmus. Dazu kommt eine nicht-stetige Stufenfunktion. Die Funktionen Bump und Spike bestehen aus einer einfachen Grundform in die jedoch an einer bestimmten Stelle ein Einschnitt gemacht wurde, den es für einen Auto-Tuner zu finden gilt. Die Wavy-Funktion besteht im Kern aus einer Sinusfunktion, welche die wellige Form erzeugt. Holes als letztes besteht aus einer geraden Ebene, mit mehreren Löchern unterschiedlicher Tiefe.

Als mathematische Formel können sie wie in Tabelle 3.1 dargestellt werden, wobei

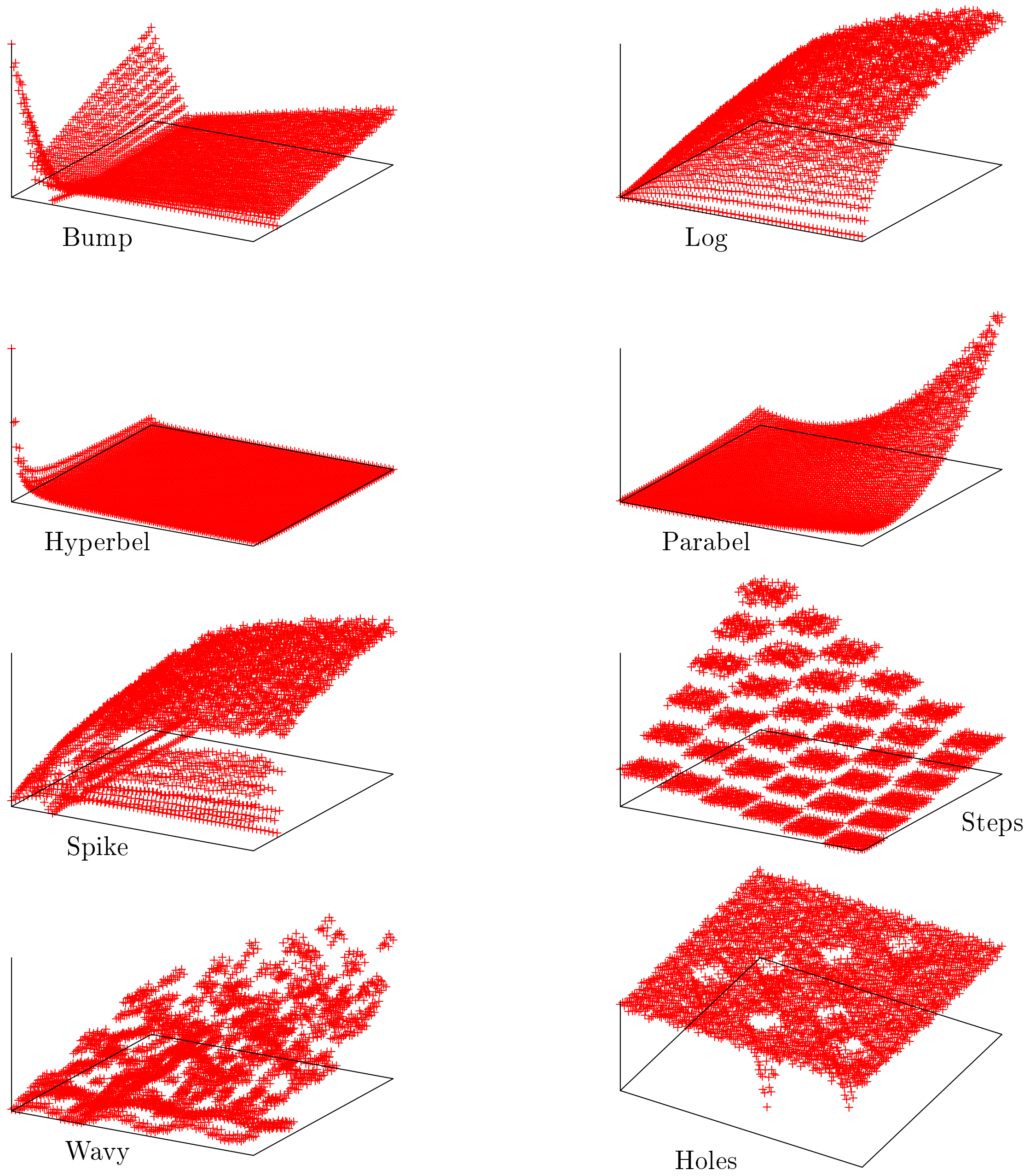


Abbildung 3.3.: Einfache Funktionen für den Algorithmenvergleich

Funktion	Formel	De Jong Äquivalent
Log	$\ln(x+e)$	Sphäre
Hyperbel	$10/(1+x)$	
Steps	$10 - \lfloor x/10 \rfloor$	Stufen
Bump	$\begin{cases} 10 \cdot (10-x) & \text{für } x \leq 10 \\ x & \text{sonst} \end{cases}$	
Spike	$\log_{10}(x) - \max(0, 3 - \text{abs}(x-10))$	
Wavy	$10 \cdot \sin(x/3) + x/2$	Biquad. Funktion
Holes	– zu komplex –	Schekels Fuchsbauten

Tabelle 3.1.: Einfache Funktionen

die implementierten Funktionen noch zusätzlich auf den Bereich $[1, 1000]$ skaliert wurden. Außerdem werden zu jedem Punkt jeder Funktion zufällig 0-10% addiert, um Ungenauigkeiten der Laufzeitmessung zu simulieren. Damit liegt das Minimum jeder Funktion bei 1 und das Maximum zwischen 1000 und 1100.

3.3.2. Algorithmenparameter

Anhand der synthetischen Programme lassen sich die Optimerungsalgorithmen mit verschiedenen Parametern ausführen. Autuner erlaubt die Parameter für jede Optimierung individuell einzustellen, aber es soll eine gute Standardeinstellung zur Verfügung stehen. Zielgröße dieser Parameteroptimierung soll die Ergebnisqualität sein, also die Minimierung der besten gefundenen Laufzeit. Die Zahl der Auswertungen wird vernachlässigt.

Die Messungen ergeben, dass keine Konfiguration signifikant besser als Andere ist, sich also keine global-optimale Konfiguration ermitteln lässt. In Anhang A finden sich beispielhafte Messdaten. Die Abweichung ist sehr groß und wiederholte Messungen zeigen, dass die Schwankungen durch die Zufallszahlengeneratoren der Optimierungsalgorithmen überwiegen. Das schließt nicht aus, dass es für spezielle Funktionen jeweils eine optimale Konfiguration gibt, allerdings ist die Funktion einer realen Anwendung unbekannt und eine entsprechende Parameteroptimierung daher nicht möglich. Die Algorithmen sind daher nur „sinnvoll“, aber nicht optimal eingestellt.

4. Evaluation

In diesem Kapitel werden die Optimierungsalgorithmen bewertet und miteinander verglichen.

4.1. Bewertung mit synthetischen Programmen

Als erster Bewertungsschritt lassen sich die Algorithmen anhand der synthetischen Programmen (siehe Abschnitt 3.3.1) vergleichen. Direkt lassen sich die Algorithmen nicht vergleichen, da die zwei Faktoren Qualität (beste gefundene Konfiguration) und Schnelligkeit (wenige Auswertungen) zu bewerten sind. Getrennt lassen sich diese allerdings nicht betrachten, da beispielsweise Simplex und Polytope nicht so eingestellt werden können, dass sie gleich viele Auswertungen benötigen, was einerseits am Zufallsfaktor und andererseits an der Abbruchbedingung (siehe Abschnitt 3.2.1) liegt. Der Zufallsalgorithmus dagegen kann auf eine genaue Schrittanzahl eingestellt werden. Es ist also möglich zu jedem Lauf eine Zufallsauswertung mit derselben Auswertungsanzahl durchzuführen. Der Unterschied zwischen den gefundenen Optima zeigt, wieviel besser ein Algorithmus verglichen mit dem Zufall ist. Als Testfeld benutzen wir synthetische Programme wobei ein Durchschnittswert aus allen Optimierungen zur Bewertung genutzt wird. In Abbildung 4.1 sind die Algorithmen jeweils verglichen mit einem anschließenden Lauf des Zufallsalgorithmus mit der gleichen Anzahl von Auswertungen. Der dargestellte Wert ist die Differenz des durchschnittlichen Ergebnisses von Algorithmus zu Zufallsalgorithmus.

Jeder Algorithmus ist besser als der Zufallsalgorithmus bei gleicher Anzahl von Auswertungen, da alle Werte positiv sind. Wie man sieht schneidet BasicEvolution in diesem Vergleich am besten ab, da der Abstand zum Zufallsalgorithmus am größten ist. Simplex und DifferentialEvolution haben das schlechteste Ergebnis.

Zusätzlich zum Vergleich mit dem Zufallsalgorithmus sollen die Algorithmen gegeneinander abschätzen lassen. Dazu lässt sich die Menge an Auswertungen im direkten Vergleich darstellen. Dabei ist zu beachten, dass bei dieser Übersicht Qualität und Schnelligkeit gleichzeitig bei Standardkonfiguration der Algorithmen dargestellt sind und die Darstellung dadurch sehr vereinfacht ist. In Abbildung 4.2 auf der X-Achse ist das beste Ergebnis, das im Schnitt jeweils gefunden wurde, aufgetragen. Je weiter

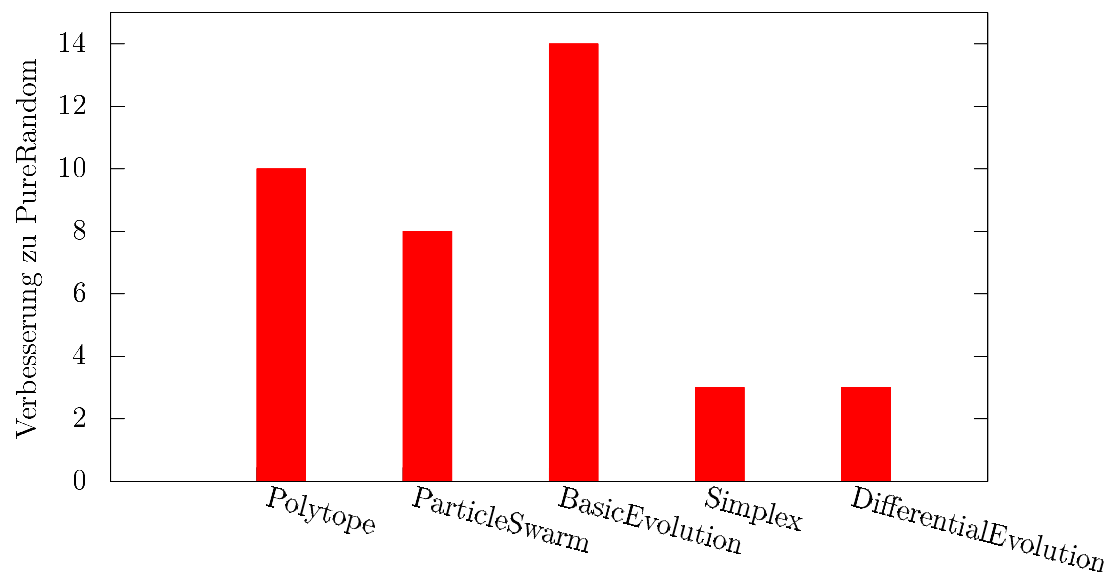


Abbildung 4.1.: Vergleich der Algorithmen zu PureRandom

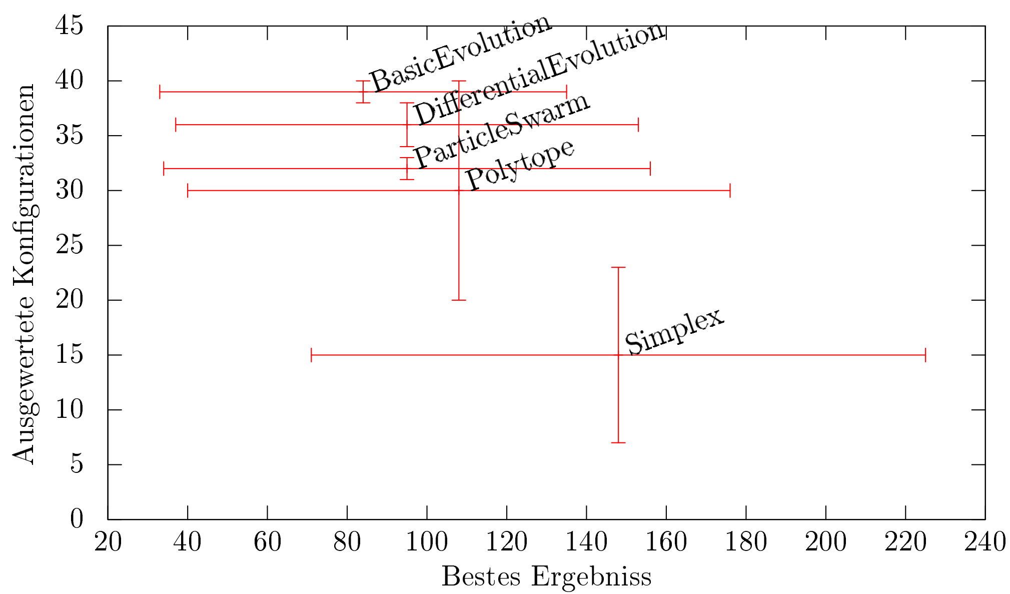


Abbildung 4.2.: Algorithmen im Vergleich für einfache Funktionen

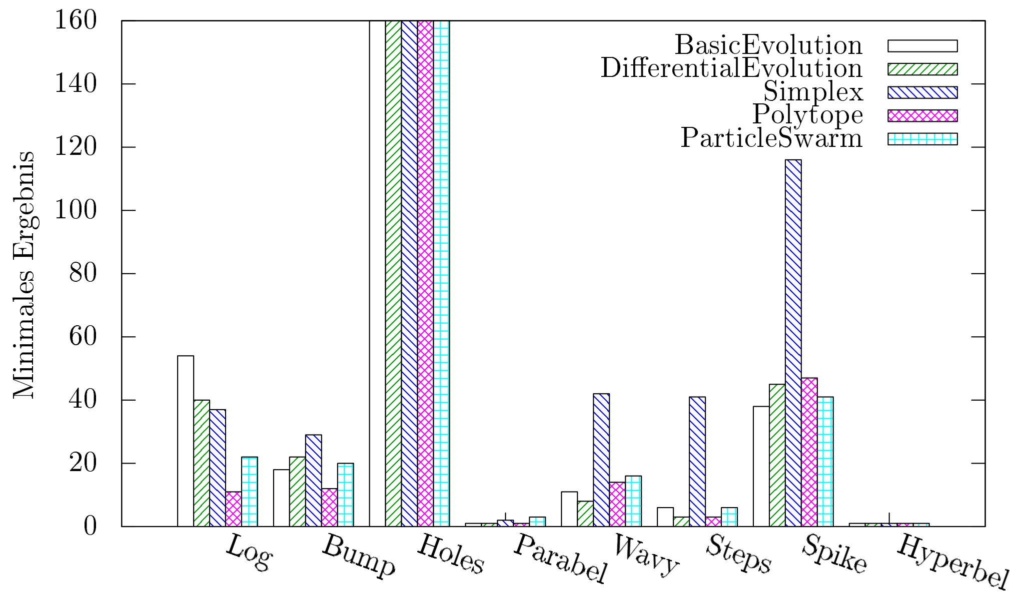


Abbildung 4.3.: Abstand zum Optimum pro Funktion und Algorithmus

links ein Algorithmus also steht, um so besser das Ergebnis, das er gefunden hat. Auf der Y-Achse ist die Anzahl der ausgewerteten Konfigurationen gezeigt. Je tiefer ein Algorithmus eingezeichnet ist, um so schneller hat er sein Ergebnis gefunden. Um die Zuverlässigkeit der Algorithmen darzustellen, ist zusätzlich die Standardabweichung[1] eingezeichnet.

Wie bereits beim Vergleich mit dem Zufalls-Algorithmus, zeigt BasicEvolution die beste Ergebnisqualität, allerdings sieht man hier, dass das mit der längsten Laufzeit verbunden ist. Insgesamt sieht man, dass eine kürzere Laufzeit mit einem schlechteren Ergebnis erkauft wird. Simplex ist hier das untere Extrem als schnellster Algorithmus mit der schlechtesten Qualität. Während die Suchdauer relativ wenig schwankt ist die Varianz der Ergebnisqualität auffallend[2]. Allerdings zeigen Simplex und Polytope doch eine erkennbar stärkere Abweichung in der Auswertungszahl als die anderen Algorithmen.

Um zu untersuchen, ob bestimmte Funktionen für bestimmte Algorithmen problematisch sind, ist in Abbildung 4.3 die Ergebnisqualität nach Funktion aufgeschlüsselt. Für die Hyperbelfunktion ist jeder Algorithmus optimal, was daran liegen dürfte, dass ein großer Bereich des Suchraums optimal ist. Genauso ist bei der Parabelfunktion fast überall das Optimum zumindest sehr nahe erreicht worden. Simplex zeigt gerade

[1]Der Übersichtlichkeit wegen wurde die Standardabweichung des Ergebnisses geviertelt und sollte nur relativ bewertet werden

[2]Wie bereits erwähnt, müssten die eingezeichneten Linien eigentlich 4-mal so lang sein

bei den Funktionen Wavy, Steps und Spike ein relativ schlechtes Ergebnis. Die Log-Funktion liefert bemerkenswert andere Ergebnisse. So ist Polytope Spitzenreiter und BasicEvolution am schlechtesten. Bei Holes versagen alle Algorithmen mit Ergebnissen zwischen 400 und 700. Man kann beobachten, dass die Algorithmen je nach Funktion stärker oder schwächer sind. Dieses Wissen ließe sich beim Auto-Tuning nutzen, falls man etwas über die ungefähre Struktur der Laufzeiten wüsste.

4.2. Testprogramme

Durch die synthetischen Programme in Abschnitt 3.3.1 ist bereits eine Bewertungsgrundlage geschaffen. Die Beobachtungen sollen aber zusätzlich anhand von echten Programmen bestätigt werden.

4.2.1. PARSEC

Eine Sammlung von Testprogrammen ist die Princeton Application Repository for Shared-Memory Computers (PARSEC) [BKSL08, BKL08]. PARSEC beinhaltet unterschiedliche Programme und zielt auf Multicoresysteme ab, wie sie heutzutage bereits auf vielen Schreibtischen stehen. Die Sammlung besteht aus den Programmen blackscholes (Finanzanalyse anhand partieller Differentialgleichungen), bodytrack (Bilderkennung), canneal (Optimierung von Chipdesigns), dedup (Komprimierung), facesim (Gesichtsanimation), ferret (Ähnlichkeitssuche), fluidanimate (Flüssigkeitsanimation), freqmine (Frequent Itemset Mining), streamcluster (Online Clustering Problem), swaptions (Finanzanalyse), vips (Bildtransformationen) und x264 (Videokomprimierung).

4.2.2. SPEC OMP

Als Variante der bekannten SPEC Sammlung besteht OMP2001 aus Programmen die durch OpenMP nebenläufig arbeiten. Es gibt OMPM2001, die für Systeme mit 4 bis 32 Prozessoren ausgelegt ist und OMPL2001, welche auch für bis zu 512 Prozessoren bzw. Cluster genutzt werden kann. SPEC OMPM2001 besteht aus den Programmen 310.wupwise (quantum chromodynamics), 312.swim (Wassersimulation), 314.mgrid (multi-grid solver in 3D potential field), 316.applu (parabolic/elliptic partial differential equations), 318.galgel (fluid dynamics analysis of oscillatory instability), 330.art (neural network simulation of adaptive resonance theory), 320.equake (Erdbebensimulation), 324.apsi (Wettervorhersage), 326.gafort (genetic algorithm code), 328.fma3d (Crash Simulation) und 332.ammp (Chemiesimulation). SPEC OMP stand für diese Arbeit leider nicht zur Verfügung.

4.2.3. Weitere Sammlungen

Eine Sammlung von nebenläufigen Testprogrammen ist Stanford Parallel Applications for Shared Memory (SPLASH) [WOT+95, SWG92]. Die Sammlung besteht aus mehreren nebenläufigen Programmen und wurde zusammengestellt mit dem Fokus auf Multiprozessorsysteme mit verteiltem Speicher in einem gemeinsamen Adressraum, wie sie im Bereich des High-Performance bzw. Cluster Computing üblich sind. Da SPLASH keinerlei Vorteile gegenüber der PARSEC Sammlung hat und seit 15 Jahren nicht mehr gepflegt wird werden diese Testprogramme nicht in die Evaluation miteinbezogen.

Es gibt weitere Sammlungen von Testprogrammen, die auf spezielle Branchen ausgerichtet sind. Die All Levels of Parallelism Benchmarksuite (ALPBench) besteht beispielsweise aus typischen Multimediaanwendungen. Genau handelt sich um SpeechRec (Spracherkennung), FaceRec (Gesichtserkennung), RayTrace (Raytracing), MPGenc (MPEG-2 Kodierung) und MPGdec (MPEG-2 Dekodierung). BioParallel ist eine Sammlung von Testprogrammen aus dem Bereich der Bioinformatik und NUMineBench aus dem Bereich des Data Mining. Da der Fokus dieser Arbeit auf allgemeingültigen Optimierungsalgorithmen liegt werden diese Programmsammlungen nicht weiter berücksichtigt.

4.3. Bewertung mit realen Programmen

Anhand der Testprogrammsammlungen können wir nun die fünf Optimierungsalgorithmen in einem realistischeren Szenario betrachten. Als Testsammlung wird Parsec 2.0 genutzt. Der einzige Optimierungsparameter bei allen Programmen ist die Anzahl an Fäden die benutzt werden. Für jedes einzelne Testprogramm wurde in mehreren Durchläufen von Autuner mit einer jeweiligen Suchstrategie die Thread-Anzahl optimiert. Es ist notwendig jedes Testprogramm einzeln zu betrachten, da (anders als bei synthetischen Programmen) das Optimum stark variiert. Für jedes Testprogramm wurde das durchschnittliche Ergebnis, die Schrittzahl und die jeweilige Standardabweichung gemessen. Zusammenfassend wird für jeden Algorithmus der Durchschnitt von Ergebnis, Schrittzahl und Abweichungen über alle Testprogramme betrachtet.

Die erste Messung in Abbildung 4.4 wurde auf einer Intel Core2 Quad CPU mit 2.4 GHz Taktung und 3GB RAM unter Ubuntu 9.04 mit Linux 2.6.28 ausgeführt. Als Datensatz für Parsec wurde „simlarge“ ausgewählt.

Auch in diesem Diagramm sieht man den Trend, dass weniger Auswertungen schlechtere Ergebnisse liefern. Man kann grob zwei Gruppen ausmachen. Die beiden evo-

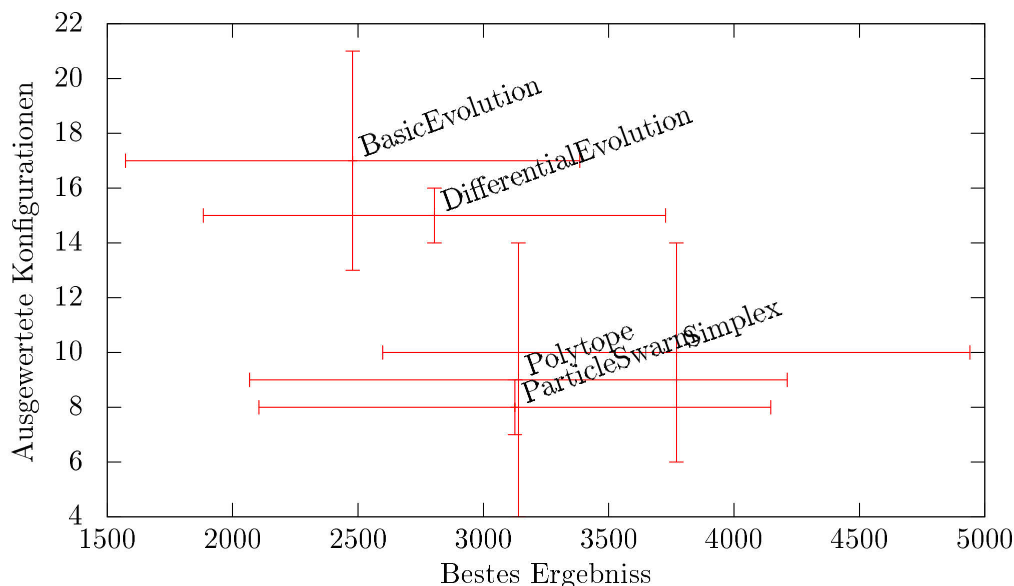


Abbildung 4.4.: PC115 simlarge

lutionären Algorithmen bilden die Gruppe langsam-aber-gut, während die restlichen drei Algorithmen eine Gruppe schnell-aber-schlechter bilden. Beim Testlauf mit synthetischen Programmen liegen Polytope und ParticleSwarm sehr viel näher an den evolutionären Algorithmen. BasicEvolution liefert durchschnittlich die besten Ergebnisse, braucht mit 15 Schritten allerdings auch am längsten. Im Gegensatz dazu liefert ParticleSwarm ein schlechteres Ergebnis, braucht dazu aber nur acht Auswertungen. Polytope ist ähnlich gut mit einer zusätzlichen Auswertung. Simplex hat mit Abstand die schlechteste Ergebnisqualität und braucht zehn Schritte.

Parsec bietet einen noch größeren Datensatz, „native“ genannt. Während simlarge 12-20 Sekunden pro Lauf benötigt, braucht native 10-30 Minuten, was Messverfälschungen verkleinern sollte. Bei ansonsten konstanten Testparametern ergibt sich ein leicht anderes Bild wie in Abbildung 4.5 zu sehen.

Die beiden Gruppen sind ebenso zu beobachten, allerdings ist die Verteilung innerhalb der Gruppen anders. DifferentialEvolution liefert nun die besten Ergebnisse trotz weniger Auswertungen im Vergleich zu BasicEvolution. ParticleSwarm ist immer noch der schnellste Algorithmus, zeigt hier aber die schlechtesten Ergebnisse. Polytope ist im Vergleich zu Simplex etwas schneller und besser.

Als weiteren Vergleich zum Testlauf wurde nun nochmals mit dem simlarge Datensatz aber auf einem anderen Rechner gemessen, einem Dell PowerEdge 2900 mit zwei 64bit Intel Xeon Quadcore CPUs mit 1.85 GHz Taktung und 8GB RAM unter Ubuntu 8.10 mit Linux 2.6.27. Das Ergebnis ist in Abbildung 4.6 zu sehen.

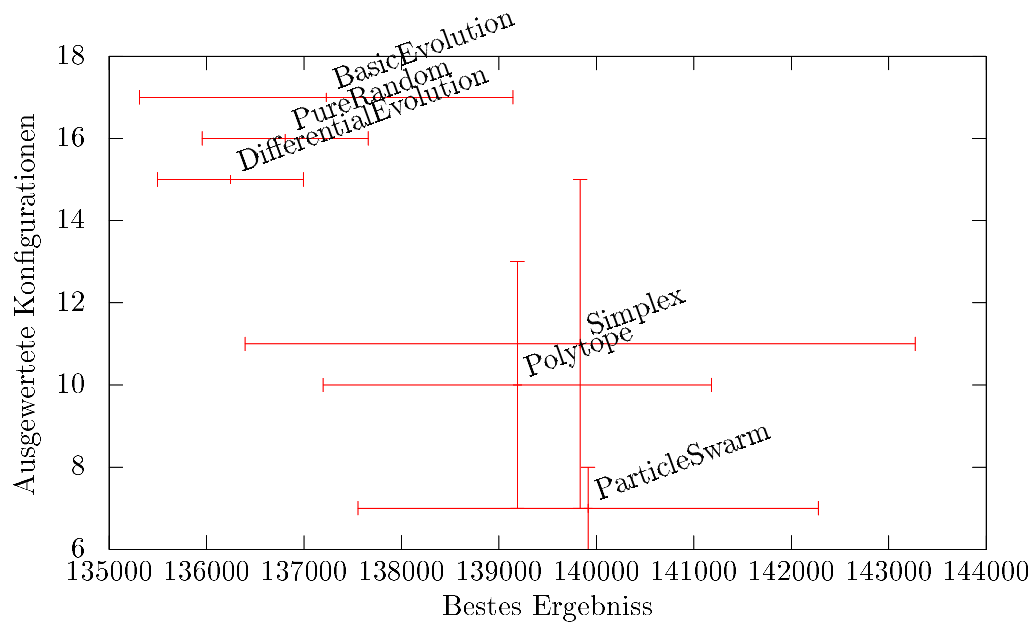


Abbildung 4.5.: PC115 native

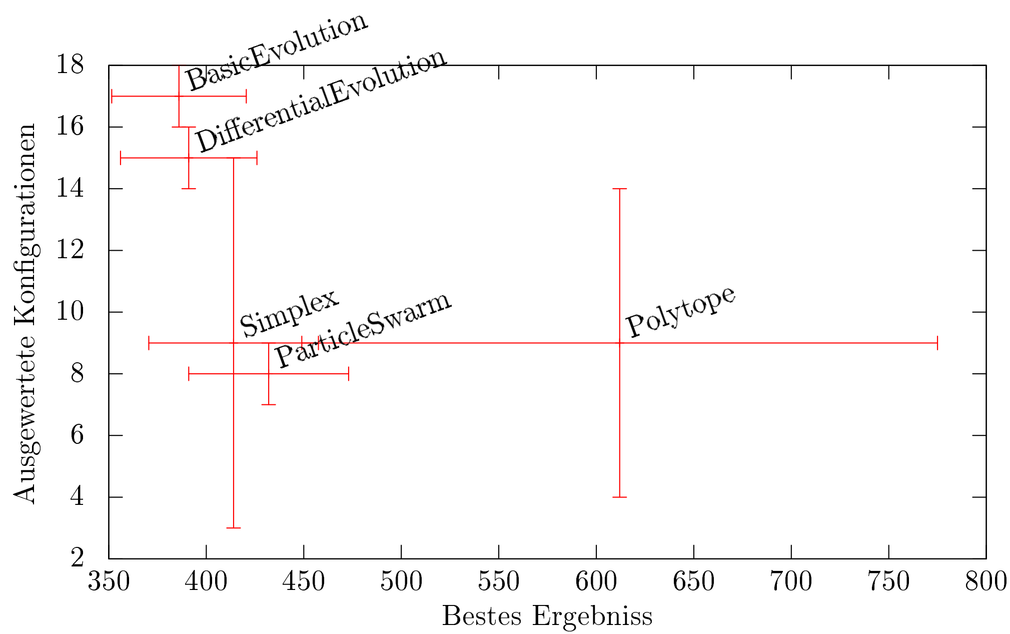


Abbildung 4.6.: DS1 simlarge

Die Gruppe der evolutionären Algorithmen sieht gleich aus: BasicEvolution mit den besten Ergebnissen und DifferentialEvolution etwas schneller und schlechter. ParticleSwarm ist wieder der schnellste Algorithmus, allerdings liefert Simplex etwas bessere Ergebnisse. Auffallend ist Polytope, welcher mit großem Abstand schlechtere Ergebnisse liefert.

4.4. Bewertung

Insgesamt ist es schwer eine allgemeine Bewertung anzugeben, da die Messungen stark von augenscheinlich nebensächlichen Details abhängen. Kleine Änderungen am Experiment, wie einem anderen Datensatz oder Rechner, führen zu starken Änderungen in der Messung. Jedoch lässt sich insbesondere im Hinblick auf die Verbesserung zu PureRandom in Abbildung 4.1, sagen das BasicEvolution die beste Ergebnisqualität zeigt. DifferentialEvolution liegt in der Mehrzahl der Experimente hinter BasicEvolution zurück. Falls die Schnelligkeit der Auswertung eine wichtigere Rolle spielt, ist ParticleSwarm der beste Kandidat, da Ergebnisqualität und Auswertungszahl stabil und gut sind. Simplex und Polytope zeigen sich als relativ instabil, sowohl bei der Ergebnisqualität als auch bei der Schrittzahl.

5. Zusammenfassung

Nun folgen einige Ideen, die im Rahmen dieser Arbeit nicht weiterverfolgt werden konnten und eine Zusammenfasssung.

5.1. Ausblick

Für weitere Vergleiche sollte die SPEC OMP Programmsammlung ebenfalls genutzt werden. PARSEC 2.0 ist zwar durchaus ausreichend, da genügend Vielfalt in der Programmauswahl steckt, aber SPEC OMP enthält aktuellere Programme bzw aktuellere Versionen und liegt dadurch vermutlich näher an tatsächlichen Tuning-Anforderungen.

Ein weiterer Aspekt im Auto-Tuning ist das Auswerten zusätzlicher Informationen. Diese können beispielsweise durch genauere Messungen im Testprogramm erlangt werden. Autuner unterstützt eine derartige Technik in Ansätzen. Inwiefern dadurch die Suchalgorithmen verbessert werden könnten ist unbekannt.

Diese Arbeit beschäftigt sich mit der Optimierung des Suchalgorithmus zur Minimierung der Auswertungsanzahl und damit zur Beschleunigung des Optimierungsprozesses. Eine zweite Möglichkeit den Prozess zu beschleunigen ist es, die einzelne Auswertung schneller durchzuführen. Eine Realisierungsidee ist das Finden von Phasen[FCOT05]. Eine Phase ist ein iterativer Programmteil mit konstanter Performanz über mehrere Iterationen hinweg. Statt die Gesamtlaufzeit eines Programms zu Messen wird die Laufzeit einer Iteration mit verschiedenen Parametern gemessen. Da die Performanz gleich ist, sind Messunterschiede dann den geänderten Parametern zuzuschreiben. Die Schwierigkeit dieses Verfahrens liegt in der Erkennung von Phasen, was gerade im Bezug auf nebenläufige Berechnungen unmöglich erscheint. Eine solche Messung bietet sich für Übersetzeroptimierungen an, da Programmteile (Funktionen) relativ einfach mit verschiedenen Optimierungsparametern erzeugt und ausgetauscht werden können. Für unsere Zwecke wäre es notwendig, dass der Entwickler extra Code einfügt, der dynamische Parametrisierung erlaubt. Trotz dieser Widrigkeiten hat die Grundidee der Beschleunigung der Auswertung seinen Reiz und weitere Gedanken in dieser Richtung im Bereich Auto-Tuning wären sinnvoll.

5.2. Zusammenfassung

In dieser Arbeit wurden fünf Optimierungsalgorithmen im Bereich Auto-Tuning untersucht und verglichen. Eine gute Konfiguration für die Parameter der Algorithmen selbst lässt sich im allgemeinen Fall nicht finden. Es wurden sowohl eine synthetische Testsammlung entwickelt, als auch anhand der Parsec 2.0 Sammlung mit verschiedenen Rechnern und Datensätzen evaluiert. Insgesamt zeigen die beiden evolutionären Algorithmen die beste Ergebnisqualität, während der Partikelschwarmalgorithmus sehr schnell ein relativ gutes Ergebnis liefert. Die beiden Simplex-basierten Algorithmen schwanken stark in ihren Laufzeiten und sind weder an Ergebnisqualität noch an Schnelligkeit herausragend.

A. Auswertungsdaten des Algorithmen-Tuning

Dieser Anhang enthält die Auswertungen der einzelnen Algorithmen mit verschiedenen Konfigurationen. Die vorletzte Spalte ist jeweils das durchschnittliche Ergebnis und die letzte Spalte die Standardabweichung. Alle Spalten davor sind Algorithmenparameter, deren Bedeutung man in der Algorithmenbeschreibung findet. Wie man an der hohen Standardabweichung sieht, ist das Ergebnis sehr variabel. Die Ergebnisse sind in dieser Reihenfolge nicht reproduzierbar und dienen nur als Beispiel, dass sich keine allgemein gute Konfiguration der Algorithmen angeben lässt.

0.40	0.20	4	10.639549
0.10	0.70	4	11.918473
0.50	0.30	5	13.136101
0.20	0.40	5	13.250337
0.20	0.60	5	13.253840
0.20	0.30	5	13.370276
0.30	0.20	5	13.969610
0.30	0.50	6	13.710788
0.850	0.10	6	13.721464
0.10	0.60	6	13.790784
0.60	0.10	6	13.875724
0.20	0.10	6	14.331534
0.60	0.30	6	14.339257
0.60	0.20	6	14.372841
0.50	0.40	6	14.448183
0.40	0.50	6	14.532230
0.30	0.30	6	14.545250
0.20	0.20	6	14.568312
0.20	0.70	6	14.646550
0.20	0.50	6	14.764339
0.40	0.10	6	14.859821
0.30	0.10	6	14.867270
0.30	0.40	6	14.867270
0.70	0.20	6	14.938445
0.10	0.30	7	14.943465
0.10	0.50	7	15.336930
0.10	0.10	7	15.358106
0.50	0.20	7	15.456621
0.40	0.30	7	15.472095
0.30	0.60	7	15.591206
0.50	0.10	7	15.620728
0.40	0.40	7	15.702366
0.10	0.850	7	15.728727
0.10	0.20	7	16.350841
0.70	0.10	8	16.762415
0.10	0.40	8	17.288518

Tabelle A.1.: Auswertung der Konfigurationen von Evolution

1.20	0.60	2.0	25	42.196320
0.90	0.30	2.0	25	42.569942
1.50	0.30	2.0	28	46.943435
1.10	0.30	2.50	28	48.213778
1.10	0.80	1.50	30	53.655823
1.20	0.10	2.50	30	55.059915
1.50	0.80	3.50	31	51.225496
1.20	0.60	3.50	31	53.542423
0.90	0.80	2.0	31	55.840124
1.10	0.10	1.50	31	68.663575
1.10	0.60	1.50	31	72.121736
0.90	0.10	3.50	32	51.061630
1.20	0.30	3.50	32	56.115822
1.50	0.80	2.50	32	56.768919
1.10	0.10	2.0	33	57.891542
1.20	0.80	2.50	33	61.186637
1.20	0.10	3.50	33	75.13130
0.90	0.80	3.50	34	76.417727
0.90	0.60	3.50	34	79.089687
1.20	0.10	1.50	34	80.809738
0.90	0.60	1.50	34	93.768156
0.90	0.10	1.50	35	55.970808
1.10	0.30	1.50	35	73.210203
1.50	0.80	2.0	35	74.799977
1.10	0.80	2.50	35	75.799610
0.90	0.80	1.50	35	78.481198
1.20	0.30	2.0	35	78.821606
1.10	0.30	3.50	36	56.346753
1.10	0.10	2.50	36	78.095861
1.50	0.60	3.50	36	79.019128
1.50	0.10	2.0	37	79.101201
1.10	0.60	2.0	37	91.501386
1.50	0.80	1.50	37	96.817212
1.50	0.30	2.50	37	97.172488
0.90	0.10	2.0	38	105.158892
1.10	0.30	2.0	38	75.734624
1.50	0.30	1.50	39	102.245961
0.90	0.30	2.50	39	77.204018
1.20	0.30	2.50	39	95.876734
1.10	0.60	3.50	40	111.081119
1.50	0.60	2.0	40	63.309294
1.20	0.80	2.0	40	82.260794
1.10	0.10	3.50	40	84.171272
1.10	0.80	3.50	40	85.150107
1.20	0.60	1.50	41	113.924476
0.90	0.30	1.50	41	85.645552
1.50	0.60	1.50	42	95.825033
0.90	0.10	2.50	42	96.056263
1.20	0.80	1.50	43	101.907779
0.90	0.30	3.50	43	102.443295
0.90	0.80	2.50	43	113.831263
0.90	0.60	2.50	45	98.773478
1.20	0.30	1.50	46	108.371368
1.50	0.60	2.50	46	116.163678
1.10	0.80	2.0	46	120.548405
1.50	0.30	3.50	47	110.697653
1.20	0.10	2.0	47	110.819239
1.20	0.60	2.50	47	115.947662
1.10	0.60	2.50	48	113.439979
1.50	0.10	1.50	49	121.486655
0.90	0.60	2.0	51	118.841002
1.50	0.10	2.50	52	126.192935
1.20	0.80	3.50	52	126.672604
1.50	0.10	3.50	57	125.755802

Tabelle A.2.: Auswertung der Konfigurationen von Simplex

1.20	0.80	2.50	8	13.503796
0.70	0.60	2.50	8	15.608377
1.20	0.10	2.50	9	15.587765
1.20	0.10	2.0	9	16.324609
0.90	0.10	2.50	9	18.722408
0.70	0.30	1.50	9	19.711490
1.20	0.60	2.0	9	20.300070
1.50	0.60	1.50	9	20.149973
0.70	0.30	3.50	10	15.304294
0.70	0.10	3.50	10	16.059710
1.50	0.80	2.0	10	16.930741
0.90	0.30	2.50	10	17.425146
1.10	0.10	2.50	10	17.367252
1.20	0.30	2.50	10	17.991268
0.90	0.10	3.50	10	18.676188
1.50	0.10	3.50	10	18.602235
0.70	0.60	3.50	10	19.685383
1.10	0.30	3.50	10	19.169257
1.20	0.60	1.50	10	19.324938
1.20	0.80	1.50	10	19.429359
0.70	0.60	2.0	10	20.789249
1.10	0.80	1.50	10	21.748398
1.20	0.10	1.50	10	21.942783
1.50	0.10	2.0	10	21.948599
0.70	0.10	2.50	10	22.382214
1.10	0.60	3.50	11	18.905781
0.90	0.30	1.50	11	20.082508
1.10	0.60	2.50	11	21.161454
0.90	0.10	1.50	11	22.764354
1.20	0.80	2.0	11	22.137569
1.50	0.30	2.0	11	22.768556
0.70	0.80	2.50	11	23.719679
0.70	0.30	2.0	11	24.680530
1.50	0.30	3.50	11	24.075180
1.50	0.80	1.50	11	24.273589
0.90	0.60	2.0	11	25.878011
0.70	0.10	1.50	11	26.587725
1.50	0.30	2.50	11	26.338456
1.50	0.60	2.50	11	26.155305
1.20	0.60	2.50	11	28.313298
0.90	0.30	2.0	12	18.335173
1.10	0.30	2.50	12	21.618114
1.10	0.80	2.0	12	22.530773
0.90	0.30	3.50	12	23.445530
0.90	0.60	1.50	12	23.031965
0.70	0.30	2.50	12	25.509802
1.50	0.80	3.50	12	25.207113
0.70	0.80	1.50	12	26.516318
0.90	0.60	3.50	12	26.394534
0.90	0.80	3.50	12	26.005082
1.20	0.30	2.0	12	26.262140
1.20	0.30	1.50	12	27.576905
1.10	0.60	2.0	13	22.787214
1.10	0.80	3.50	13	25.385316
1.10	0.60	1.50	13	27.357292
1.10	0.80	2.50	13	27.060514
1.20	0.60	3.50	13	27.683930
1.20	0.30	3.50	13	28.567351
1.10	0.30	1.50	13	29.295198
0.70	0.10	2.0	13	30.499766
1.50	0.60	2.0	13	30.600466
1.50	0.60	3.50	13	30.693066
0.70	0.60	1.50	13	31.518475
1.10	0.10	3.50	13	52.834341
0.90	0.60	2.50	14	30.287115
0.90	0.80	2.50	14	31.309058
0.70	0.80	3.50	14	33.346128
1.50	0.30	1.50	14	33.382997
1.20	0.80	3.50	14	34.492753
0.90	0.80	1.50	15	30.541892
1.10	0.10	2.0	15	30.279767
1.10	0.30	2.0	15	32.871562
1.50	0.80	2.50	15	33.777528
0.90	0.10	2.0	15	54.816512
1.20	0.10	3.50	15	61.101321
1.50	0.10	2.50	17	53.955405
1.10	0.10	1.50	17	57.100064
0.90	0.80	2.0	17	61.389838
0.70	0.80	2.0	17	62.156884
1.50	0.10	1.50	19	60.326271

Tabelle A.3.: Auswertung der Konfigurationen von Polytope

0.80	0.60	2.0	11	12.738580
0.80	0.90	1.30	12	17.654016
0.80	0.30	2.0	13	16.912802
0.80	0.60	2.70	13	17.202159
1.40	0.90	4.0	13	17.813117
0.80	0.60	0.50	13	19.050309
1.10	0.90	1.30	13	19.295077
0.50	0.90	2.70	13	19.400663
0.50	0.30	2.70	13	20.197419
1.40	0.90	1.30	13	21.420284
1.40	0.60	0.50	14	19.366392
1.40	0.60	1.30	14	19.500733
1.10	0.30	0.50	14	20.989453
0.50	0.90	2.0	14	21.584386
0.50	0.60	4.0	14	21.743965
0.80	0.30	0.50	14	22.062411
1.10	0.90	2.0	14	22.602149
0.80	0.60	1.30	14	24.934486
1.40	0.30	1.30	15	20.331889
0.80	0.30	2.70	15	21.820698
0.50	0.30	1.30	15	22.653130
1.40	0.90	2.0	15	23.090969
1.10	0.30	2.70	15	23.666130
0.50	0.90	0.50	15	23.895606
0.80	0.60	4.0	15	23.974391
1.10	0.30	2.0	15	24.571470
1.10	0.60	1.30	15	26.170048
0.50	0.60	0.50	15	26.882947
0.80	0.90	2.70	15	27.186788
0.50	0.60	2.70	15	27.188233
1.40	0.30	2.70	15	28.052374
0.50	0.60	1.30	15	28.302448
0.50	0.30	2.0	16	24.455207
0.50	0.90	4.0	16	25.416249
1.10	0.60	2.70	16	25.857853
1.40	0.60	2.70	16	26.540266
1.40	0.60	2.0	16	27.198214
1.40	0.90	2.70	16	28.037730
1.40	0.60	4.0	16	29.318327
1.40	0.30	0.50	16	29.679117
1.10	0.90	2.70	16	30.149035
1.10	0.90	0.50	16	30.402420
1.40	0.30	2.0	16	38.306191
1.10	0.30	4.0	16	47.66640
1.10	0.30	1.30	17	22.335430
0.80	0.30	1.30	17	24.885021
1.40	0.30	4.0	17	28.885241
0.80	0.90	2.0	17	29.649741
1.10	0.60	0.50	17	35.428499
1.10	0.60	4.0	17	39.336824
0.50	0.30	4.0	18	29.651307
0.80	0.90	0.50	18	30.691204
0.80	0.90	4.0	18	33.477391
1.10	0.60	2.0	18	51.483284
0.50	0.60	2.0	18	60.446789
0.50	0.90	1.30	18	63.699294
0.80	0.30	4.0	20	34.675434
0.50	0.30	0.50	21	64.594007
1.10	0.90	4.0	23	53.557846
1.40	0.90	0.50	24	52.654399

Tabelle A.4.: Auswertung der Konfigurationen von ParticleSwarm

B. Beispielcode

Listing B.1: Lineare Suche: Java-Code

```
public class Linear extends OptimizationAlgorithm {
  @Override
  public DataPoint optimize(Evaluator e, TestListener l) {
    DataPoint best = null;
    do {
      l.beforeTest(best);
      DataPoint test = e.eval();
      l.afterTest(test);

      best = getChooser().choose(test, best);
    } while (nextConfig()); // linear progression
    return best;
  }
}
```

Listing B.2: Zufall: Java-Code

```
public class PureRandom extends OptimizationAlgorithm {
  @Override
  public DataPoint optimize(Evaluator e, TestListener l) {
    DataPoint best = null;
    for (int i = 0; i < 5; i++) {

      l.beforeTest(best);
      getRandomizer().nextConfig(); // random progression
      DataPoint test = e.eval();
      l.afterTest(test);

      best = getChooser().choose(best, test);

    }
    return best;
  }
}
```

Literaturverzeichnis

[BI91] Barton, Russell R. ; Ivey, John S. Jr.: Modifications of the Nelder-Mead simplex method for stochastic simulation response optimization. In: *WSC '91: Proceedings of the 23rd conference on Winter simulation.* Washington, DC, USA : IEEE Computer Society, 1991. – ISBN 0–7803–0181–1, S. 945–953

[BKL08] Bienia, Christian ; Kumar, Sanjeev ; Li, Kai: PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In: *Proceedings of the 2008 International Symposium on Workload Characterization*, 2008

[BKSL08] Bienia, Christian ; Kumar, Sanjeev ; Singh, Jaswinder P. ; Li, Kai: The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008

[DJ75] De Jong, Kenneth A.: *An analysis of the behavior of a class of genetic adaptive systems.* Ann Arbor, MI, USA, Diss., 1975

[FCOT05] Fursin, Grigori ; Cohen, Albert ; O'Boyle, Michael ; Temam, Olivier: A Practical Method for Quickly Evaluating Program Optimizations. In: *In Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005*, Springer Verlag, 2005, S. 29–46

[KE95] Kennedy, J. ; Eberhart, R.: Particle swarm optimization. In: *in Proceedings of the IEEE Int. Conf. on Neural Networks.* Piscataway, NJ, 1995

[Rec73] Rechenberg, I.: *Evolutionsstrategie: optimierung technischer systeme nach prinzipien der biologischen evolution.* Frommann-Holzboog, 1973

[SP95] Storn, R. ; Price, K.: Differential Evolution- A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces. Version: 1995. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.9696`. 1995. – Forschungsbericht

[SWG92] Singh, Jaswinder P. ; Weber, Wolf ; Gupta, Anoop: SPLASH: Stanford parallel applications for shared-memory*. Stanford, CA, USA : Stanford University, 1992. – Forschungsbericht

[TCC+09] Tiwari, Ananta ; Chen, Chun ; Chame, Jacqueline ; Hall, Mary ; Hollingsworth, Jeffrey: Scalable Autotuning Framework for Compiler Optimization. (2009), May

[WOT+95] Woo, Steven C. ; Ohara, Moriyoshi ; Torrie, Evan ; Singh, Jaswinder P. ; Gupta, Anoop: The SPLASH-2 programs: Characterization and methodological considerations. In: *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, S. 24–36