UNIVERSITY OF DORTMUND

REIHE COMPUTATIONAL INTELLIGENCE

COLLABORATIVE RESEARCH CENTER 531

Design and Management of Complex Technical Processes and Systems by means of Computational Intelligence Methods

Parallele Strategien für schwierige Optimierungsprobleme

Thorsten Bernholt

No. CI-86/00

Technical Report ISSN 1433-3325 Juli 2000

Secretary of the SFB 531 · University of Dortmund · Dept. of Computer Science/XI
44221 Dortmund · Germany

This work is a product of the Collaborative Research Center 531, "Computational Intelligence", at the University of Dortmund and was printed with financial support of the Deutsche Forschungsgemeinschaft.

Parallele Strategien für schwierige Optimierungsprobleme

Thorsten Bernholt

11.Juli.2000

1 Einleitung

In der Informatik sind viele Probleme als NP-vollständig bekannt. Bei diesen Problemen kann man es nicht erwarten, Algorithmen zu finden, die immer in polynomieller Zeit eine optimale Lösung finden. Daher hofft man mit Hilfe von Heuristiken Lösungen zu finden, die in vernünftiger Zeit akzeptable Annäherungen an die optimalen Lösungen des Problems liefern. Für das Erfüllbarkeitsproblem SAT wurde von Cook zuerst die NP-Vollständigkeit nachgewiesen [GJ78][W93]. Die Optimierungsvariante MAXSAT ist folgendermaßen definiert:

Definition MAXSAT Eine Eingabe für das Problem MAXSAT besteht aus einer Klauselmenge mit k Klauseln über einer Menge von v Variablen. Variablen können mit „Und“, „Oder“ bzw. „Nicht“ zu booleschen Formeln kombiniert werden. Eine einfache Variable oder eine negierte Variable wird Literal genannt. Eine Disjunktion von Literalen heißt Klausel. Gesucht ist eine Belegung der Variablen mit 0 oder 1, so dass die Auswertung der Klauselmenge die maximale Anzahl von Klauseln erfüllt.

Es wurden verschiedene Algorithmen implementiert, um MAXSAT exakt zu lösen bzw. zu approximieren. Exemplarisch werden der exakte Algorithmus und der genetische Algorithmus mit Mehrfach-Selektion vorgestellt.

2 Exakter Algorithmus

Um sich dem Problem MAXSAT zu nähern, wurde im ersten Schritt versucht, das Problem durch Aufzählung aller Möglichkeiten exakt zu lösen. Der einfache Ansatz wurde mittels Pruning erweitert und schließlich parallelisiert.

2.1 Effiziente Auswertung einer Klauselmenge

Der naive Ansatz besteht darin, für eine Belegung der Variablen alle Literale zu testen. Betrachtet man jedoch zwei aufeinander folgende Belegungen, so unterscheiden sich diese häufig nur in wenigen Bits. Es reicht damit aus, nur die Literale, die sich geändert haben, neu zu testen und sich die übrigen Zwischenergebnisse zu merken. Welche Zwischenergebnisse müssen gespeichert werden? Dies sind zum einen für jede Klausel m die Anzahl der Literale L_m, die Anzahl

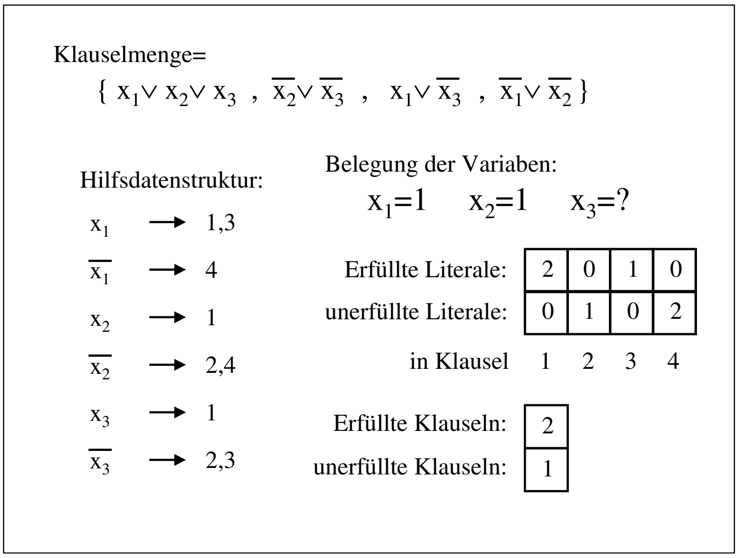

Abbildung 1: Auswertung einer Klauselmenge. In der Hilfsdatenstruktur ist gespeichert, in welcher Klausel ein Literal vorkommt, z.B. kommt x_1 in den Klauseln 1 und 3 vor. Für eine exemplarische Belegung der Variabeln mit $x_1 = 1$ und $x_2 = 1$ sind in den Datenstrukturen entsprechende Werte eingetragen.

der erfüllten Literale L_m^1 und die Anzahl der unerfüllten Literale L_m^0 . Zusätzlich benötigt man die Anzahl erfüllter Klauseln K^1 und unerfüllter Klauseln K^0 insgesamt. Des Weiteren benötigt man eine Hilfsdatenstruktur, in der für jede Variable verzeichnet ist, in welchen Klauseln sie positiv bzw. negativ vorkommt. Die Datenstrukturen sind in Abbildung 1 dargestellt.

Mit diesen Datenstrukturen ist es effizient möglich, Variablen zu ändern. Das Ändern einer Variable z. B. von 0 auf 1 besteht aus 2 Schritten. Zunächst wird die Variable x_i von 0 auf „nicht belegt“ gesetzt. Alle Klauseln, die x_i oder $\overline{x}_i$ enthalten werden „benachrichtigt“ und die Datenstrukturen werden nach folgendem Algorithmus aktualisiert:

Ändere x_i von 0 auf „nicht belegt“:
  $\forall$ Klauseln m, die x_i enthalten:
    Wenn $L_m^0 = L_m$,
      dann erniedrige K^0 um 1
    erniedrige L_m^0 um 1
  $\forall$ Klauseln m, die $\overline{x}_i$ enthalten:
    Wenn $L_m^1 = 1$,
      dann erniedrige K^1 um 1
    erniedrige L_m^1 um 1

Im zweiten Schritt wird x_i von „nicht belegt“ auf 1 gesetzt:

Ändere x_i von „nicht belegt“ auf 1:
  $\forall$ Klauseln m, die x_i enthalten:
    erhöhe L_m^1 um 1
    Wenn $L_m^1 = 1$,
      dann erhöhe K^1 um 1
  $\forall$ Klauseln m, die $\overline{x_i}$ enthalten:
    erhöhe L_m^0 um 1
    Wenn $L_m^0 = L_m$,
      dann erhöhe K^0 um 1

Das Ändern einer Variable von 1 auf 0 geschieht analog. Auch bei genetischen Algorithmen zeigt sich, dass sich häufig in einem Schritt nur wenige Variablenbelegungen ändern, wie z.B. bei der Mutation und somit der Effizienzgewinn auch dort genutzt werden kann.

Binär-Code	Gray-Code
0 0 0	0 0 0
0 0 1	0 0 1
0 1 0	0 1 1
0 1 1	0 1 0
1 0 0	1 1 0
1 0 1	1 1 1
1 1 0	1 0 1
1 1 1	1 0 0

Abbildung 2: Bitänderungen im Binär-Code und im Gray-Code. Die Änderungen sind mit Pfeilen markiert.

Gray-Code Im naiven Ansatz müssen alle 2^v Belegungen der Variablen probiert werden. Zweckmäßigerweise ordnet man die Belegungen in einer festen Reihenfolge an, wie z. B. dem Binär-Code. Wie in Abbildung 2 zu sehen, können beim binären Zählen innerhalb eines Schrittes mehrfache Bitänderungen auftreten. Da Bitänderungen eine teuere Operation darstellen, sollten sie vermieden werden. Eine Möglichkeit ist es, die Reihenfolge umzuordnen und hierfür den Gray-Code zu verwenden [B75]. Wie man in der Abbildung sieht, kommt der Gray-Code mit einer Bitänderung pro Zählschritt aus, so dass sich die Anzahl der Bitänderungen für die Aufzählung aller Belegungen insgesamt von $2^{v+1} - 2 - v$ auf nur noch $2^v - 1$ verringert und rund die Hälfte eingespart werden kann.

In Tabelle 1 sind einige Testläufe mit dem naiven Ansatz aufgeführt, um die Rechenzeit für die Probleminstanz *aim-50-1_6-no-1* abzuschätzen. Die Berechnung wird mit dem naiven Algorithmus wahrscheinlich ungefähr 463 Jahre dauern.

Problem	Variablen	Klauseln	Literale/K.	Zeit [s]
zufällig	21	80	3	28
zufällig	22	80	3	51
zufällig	23	80	3	109
aim-50-1_6-no-1	50	80	3	ca. 463 Jahre

Tabelle 1: Rechenzeiten für den exakten Algorithmus.

2.2 Alpha-Pruning

Da es sehr zeitintensiv ist, alle Belegungen aufzuzählen, ist es ratsam, durch gezielte Maßnahmen Rechenzeit einzusparen. Zu diesem Zweck ist es vorteilhaft, sich den binären Berechnungsbaum wie in Abbildung 3 anzusehen. An der Wurzel wird die erste Variable x_1 mit „0“ oder mit „1“ belegt, wobei die übrigen Variablen $x_1, \ldots, x_v$ noch „nicht belegt“ sind. Auf der nächsten Ebene wird x_2 belegt, auf den weiteren Ebenen die übrigen Variablen bis hinunter zu x_v.

Mit Hilfe des Alpha-Pruning wird nun die Berechnungstiefe begrenzt. Für das Alpha-Pruning ist die Anzahl der unerfüllten Klauseln K^0 wichtig. Dies sind Klauseln, in denen bereits alle Literale zu „0“ ausgewertet wurden. Auch aus einer teilweisen Belegung der Variablen $x_1, \ldots, x_i$ lässt sich die Anzahl der unerfüllbaren Klauseln K^0 berechnen. Angenommen, die Variablen $x_1, \ldots, x_i$ sind belegt, und es ist bereits eine gute Lösung bekannt. Wenn es aufgrund der unerfüllten Klauseln K^0 nicht mehr möglich ist, durch eine Belegung der restlichen Variablen $x_{i+1}, \ldots, x_v$ eine bessere Lösung zu finden, so muss der Teilbaum der Variablen $x_{i+1}, \ldots, x_v$ nicht weiter durchsucht werden. Stattdessen wird die Berechnung mit der Nachfolgebelegung von $x_1, \ldots, x_i$ fortgesetzt. Durch diesen Ansatz verringert sich die Rechenzeit für das Problem *aim-50-1_6-no-1* auf 11,4 Stunden.

Der hell schattierte und dunkel schattierte Bereich in Abbildung 3 enthalten die Belegungen, die vom Alpha-Pruning durchsucht werden. Der Baum wird von oben nach unten durchlaufen, wobei immer das linke Kind zuerst betrachtet wird. Die Belegung „000“ erfüllt 4 der 5 Klauseln und wird zuerst gefunden. Bei der Prüfung der Belegung „01X“ wird festgestellt, dass bereits eine Klausel nicht erfüllt ist und es muss nicht weiter in die Tiefe gesucht werden. Gleiches trifft auch auf die Belegungen „11X“ und „10X“ zu.

2.3 Beta-Pruning

Im Berechnungsbaum muss bei einer Variablen die Wahl getroffen werden, ob sie mit 0 oder 1 zu belegen ist. Man schaut sich nun die Anzahl der erfüllten Klauseln vor und nach der Belegung mit einem Wert an. Werden durch die Belegung keine zusätzlichen Klauseln erfüllt, so ist die Belegung der Variablen nutzlos. In diesem Fall muss nur in eine Richtung verzweigt werden. Dies kann auftreten, wenn eine Variable ausschließlich positiv oder negativ vorkommt oder wenn alle Klauseln, in denen die eine Sorte von Literalen vorhanden ist, bereits erfüllt sind. Die Rechenzeit für *aim-50-1_6-no-1* verringert sich weiter auf nur noch 9 Minuten.

Der dunkel schattierte Bereich in Abbildung 3 enthält die Belegungen, die von Alpha-Beta-Pruning probiert werden. In der Belegung „1XX" kommt das Beta-Pruning zum Zuge, da die Belegung $x_1 = 0$ bereits getestet ist und mit $x_1 = 1$ keine zusätzlichen Klauseln zu erfüllen sind.

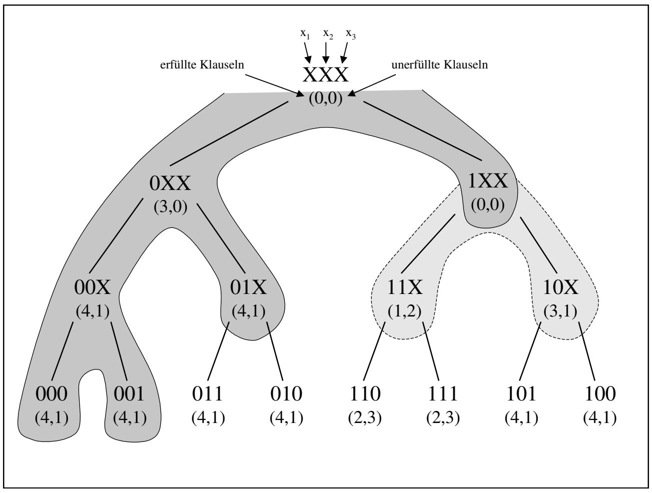

Abbildung 3: Berechnungsbaum für die Klauselmenge $\{\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3, \overline{x}_1 \vee \overline{x}_2, \overline{x}_1 \vee x_3, x_2, \overline{x}_2\}$. Der Baum wird von links nach rechts durchlaufen. Der dunkel hinterlegte Bereich beschreibt die Berechnungsschritte des Alpha-Beta-Prunings. Bei Verwendung des Alpha-Prunings ohne Beta-Pruning werden zusätzlich die Belegungen im hellen Bereich geprüft.

3 Parallelisierung

Der verwendete Parallelrechner, eine SGI Origin 2000, unterstützt das Modell der Shared-Memory-Programmierung. Mit diesem Modell ist es sehr einfach, parallele Programme zu implementieren. Das Modell sieht vor, dass der gesamte Speicher allen Prozessen zur Verfügung steht. Kommunikation erfolgt durch Zugriff auf die gemeinsam genutzten Speicherbereiche. Bei schreibenden Zugriffen auf gemeinsam genutzte Variablen muss explizit synchronisiert werden, um Inkonsistenzen zu vermeiden.

3.1 Parallelisierung des exakten Algorithmus

Die obigen 3 Algorithmen werden auf eine sehr einfache Art parallelisiert. In einer gemeinsamen Variable wird eine Menge von Aufgaben verwaltet. Sobald ein Prozess frei ist, greift er auf diesen Speicherbereich zu, holt sich eine Aufgabe ab und berechnet die nächste Aufgabe für seinen Nachfolger. Durch Synchronisierung wird sichergestellt, dass nur ein Prozess zur gleichen Zeit Zugriff erlangt und somit jede Aufgabe nur einmal bearbeitet wird. Falls mehrere Prozesse gleichzeitig neue Aufgaben anfordern, werden sie in eine Warteschlange eingereiht und nacheinander bearbeitet, wobei natürlich Rechenzeit vergeudet wird.

Als Aufgabe wird die Belegung eines Teils des Variablenvektors $x_1, \ldots, x_t$ verwendet. Ein Prozess kopiert sich diese Belegung und berechnet die nächste Aufgabe für seinen Nachfolger, indem die Nachfolge-Belegung im Gray-Code generiert wird. Während der Bearbeitung der Aufgabe werden die Variablen $x_1, \ldots, x_t$ konstant gehalten. Die restlichen Variablen sind unbelegt und werden durchprobiert.

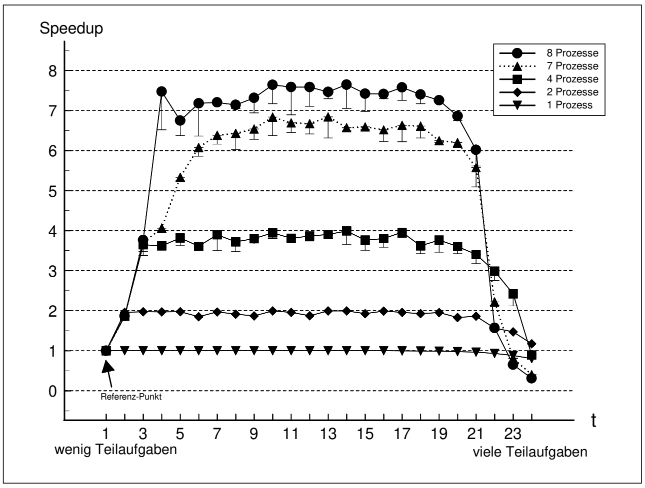

Abbildung 4: Speedup des naiven Algorithmus. Es werden Läufe mit 1, 2, 4, 7 und 8 Prozessen verglichen. Je nach Wahl des Parameter t gibt es 2^t Aufgaben.

In Abbildung 4 ist der Speedup des naiven Algorithmus dargestellt. Der Speedup ist definiert als der Quotient aus der Rechenzeit, die das Programm bei Ausführung auf einem Prozessor braucht, geteilt durch die Rechenzeit mit mehreren Prozessoren. Als Referenz wurde der Lauf mit $t = 1$ auf einem Prozessor gewählt. Bei 8 Prozessoren beträgt der Speedup idealerweise 8, d. h. das Programm läuft achtmal schneller ab. Dieser Wert kann aber normalerweise nur schwer erreicht werden, da es Wartezeiten bei der Kommunikation gibt. Am Ende der Rechnung tritt ein weiteres Problem auf, dass einige Prozesse noch mit ihrer Aufgabe beschäftigt sind, andere ihre Rechnungen aber bereits beendet haben und warten, wodurch der Speedup ebenfalls verschlechtert wird.

In Abbildung 4 ist auf der x-Achse der Parameter t aufgetragen. Mit dem Parameter t wird die Anzahl der verfügbaren Aufgaben auf 2^t festgelegt. Es wurden Läufe auf 1, 2, 4, 7 und 8 Prozessoren durchgeführt. Über einen weiten Wertebereich von t liegt der Speedup der Läufe nahe am idealen Wert. An den Rändern fällt die Kurve jedoch stark ab.

Am linken Rand liegt der Grund darin, dass es zu wenig Aufgaben gibt, um alle Prozesse nutzen zu können. Beim Lauf mit 8 Prozessoren sind erst ab

$t = 3$ genug Aufgaben vorhanden. Beim Lauf mit 7 Prozessoren ist der Anstieg der Kurve recht flach, da sich die Zahl der Aufgaben nicht durch 7 teilen lässt. Dadurch ergeben sich am Ende der Berechnungen Wartezeiten aufgrund der ungleichen Prozessorauslastung.

Am rechten Rand gibt es zwar sehr viele Aufgaben, allerdings sind diese so klein, dass die meiste Zeit mit Warten auf Kommunikation verbracht wird und sich die Rechenzeit entsprechend verschlechtert.

Für die Wahl von t scheint es sinnvoll zu sein, t etwas größer als die Zahl der Prozesse zu wählen.

3.2 Parallele Auswertung der Klauselmenge

Es wurden weitere Algorithmen implementiert, die sequentiell arbeiten und sich nicht mit obigem Ansatz parallelisieren lassen. Um diese Algorithmen dennoch zu parallelisieren, wurde ein feinkörniger Ansatz gewählt. Die Änderung einer Variablen wird nun parallelisiert, indem vor Beginn der Berechnung die Klauseln auf die Prozessoren verteilt werden. Bei einer Variablenänderung berechnet jeder Prozessor für die ihm zugeteilten Klauseln die Anzahl der erfüllten bzw. unerfüllten Literale jeder Klausel. Erst nachdem alle Klauseln bearbeitet wurden, wird von jedem Prozessor die Gesamtanzahl der erfüllten bzw. unerfüllten Klauseln aktualisiert, so dass nur am Ende der Berechnung einer Variablenänderung Kommunikation notwendig ist.

4 Genetischer Algorithmus mit Mehrfach-Selektion

Bei genetischen Algorithmen wird häufig nur ein Fitnesskriterium verwendet, wie z. B. bei SAT die Anzahl der erfüllten Klauseln. In der Natur existieren jedoch verschiedenartige Umweltbedingungen, denen sich Individuen anpassen müssen, wie z. B. unterschiedliche Temperaturen, viel oder wenig Niederschlag oder das Vorhandensein von Räubern. Nicht immer werden alle Eigenschaften eines Individuums zu seiner Beurteilung herangezogen; so ist es unerheblich, ob es schnell laufen kann, wenn keine Feinde vorhanden sind.

Diese Idee wird auf MAXSAT übertragen, indem zur Fitnessbewertung nur eine Teilmenge der Klauseln berücksichtigt wird. Ein Selektor betrachtet nur eine Teilmenge der Klauseln und wählt Individuen für die nächste Generation aus. Damit alle Klauseln in den genetischen Prozess eingehen, sind mehrere Selektoren mit jeweils verschiedenen Teilmengen vorhanden. Ein Individuum des genetischen Algorithmus stellt eine Belegung der gegebenen Variablen dar, besteht also aus einem Bitstring der Länge v. Der Algorithmus gliedert sich in folgende Schritte:

Initialisierung. Es werden n Individuen und s Selektoren erzeugt. Jeder Selektor besitzt eine Maske, die die Klauseln bestimmt, welche in die Fitnessberechnung eingehen. Am Anfang werden die Klauseln zufällig auf die Selektoren verteilt, so dass jede Klauseln genau in einem Selektor enthalten ist. Jeder Selektor erhält also eine disjunkte Teilmenge mit $1/s$ aller Klauseln und zudem gilt, dass die Klauselmenge überdeckt ist.

Die Schritte Rekombination, Mutation, Selektion und Selektoranpassung werden nun g-mal iteriert:

Rekombination. Die n Individuen werden nach ihrer Fitness sortiert. Die Eltern werden zufällig bestimmt, wobei ein Individuum, welches in der Sortierung an der Position i $(0 \leq i < n)$ steht, mit Wahrscheinlichkeit $\frac{n-i}{n(n-1)/2}$ als Elter gewählt wird.

Mit 90% Wahrscheinlichkeit werden 2 Eltern ausgewählt. Aus diesen wird mittels uniformen Crossovers ein Kind erzeugt. Der Bitstring des Kindes wird gebildet, indem für jedes Bit getrennt mit einem Münzwurf ausgewürfelt wird, von welchem Elter das Bit zu nehmen ist.

Mit 10 % Wahrscheinlichkeit wird der ausgesuchte Elter geklont.

Mutation. Alle im vorherigen Schritt erzeugten Kinder werden mutiert, indem jedes Bit mit Wahrscheinlichkeit $1/v$ gekippt wird.

Selektion. Jeder Selektor wertet eine Teilmenge der Klauseln aus und berechnet hieraus die Anzahl der erfüllten Klauseln. Aufgrund dieses Fitnesswertes wird allen Individuen ein Rang zugeordnet, das beste erhält den Rang 0. Da die Fitnessbewertungen der Selektoren sich unterscheiden, werden den Individuen unter Umständen verschiedene Ränge zugeordnet. Ein Indiviuum merkt sich davon den kleinsten Rang. Anhand des Ranges werden die besten Individuen selektiert, die die Eltern der nächsten Generation bilden.

Selektoranpassung. Ein Selektor besitzt am Anfang $1/s$ aller Klauseln. Die Selektionsbedingungen werden nun verschärft, indem nach jeder Iteration Klauseln hinzugefügt werden, so dass am Ende alle Klauseln in die Fitnessbewertung eines Selektors eingehen. Die Auswahl erfolgt zufällig aus der Menge der restlichen Klauseln. Um den Zeitpunkt und die Anzahl der Klauseln zu bestimmen, wurden 2 Strategien probiert:

Linearer Zuwachs. Pro Iteration werden $\frac{k-k/s}{g}$ Klauseln hinzugefügt, so dass die Größe der Teilmenge linear mit der Anzahl der Generationen wächst und die Teilmenge am Ende alle Klauseln enthält.

Adaptiver Zuwachs. Es werden die Individuen betrachtet, denen der Selektor die Ränge $0, 1, ..., n/s$ zugeordnet hat. Sind in diesen Individuen alle Klauseln, die der Selektor betrachtet, erfüllt, so werden neue hinzugefügt, und zwar solange, bis eine unerfüllte Klausel im Individuum mit Rang 0 gefunden wird.

Ein genetischer Algorithmus ohne Selektoren lässt sich simulieren, indem nur ein Selektor verwendet wird. Dadurch werden am Anfang alle Klauseln in die Teilmenge aufgenommen.

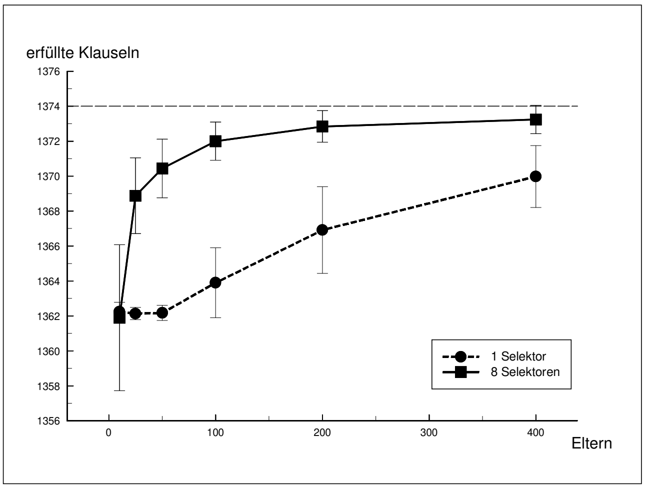

Abbildung 5: Vergleich eines genetischen Algorithmus ohne Selektoren und eines mit 8 Selektoren. Dargestellt ist der Mittelwert der erfüllten Klauseln über 50 Läufe und die Standardabweichung. 1374 Klauseln können maximal erfüllt werden. Die Anzahl der Generationen ist 800.

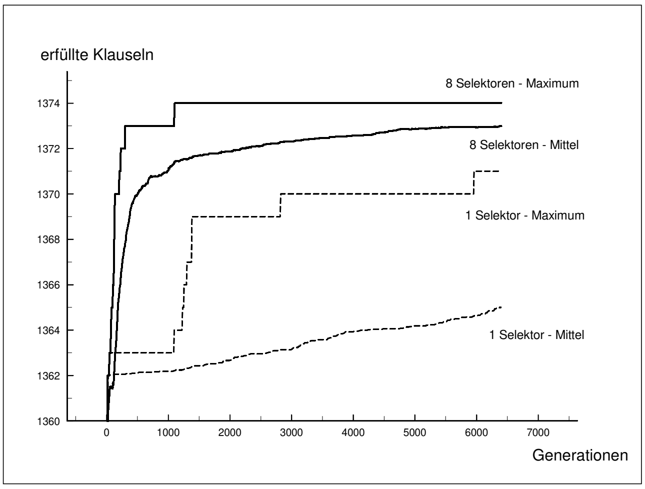

Abbildung 6: Entwicklung der besten Individuen für Läufe ohne Selektor und mit 8 Selektoren. Dargestellt ist der Mittelwert über 50 Läufe und der beste Lauf. Die Population umfasst 50 Eltern und 100 Kinder.

4.1 Probleminstanz *ii32b1*

Im Rahmen eines Diamcs-Workshops wurden schwere Probleminstanzen gesammelt. Die Probleminstanz *ii32b1* ist ein konstruiertes Problem, weitere Erläuterungen finden sich in [KR92]. Die Klauselmenge ist derartig konstruiert, dass sie Algorithmen dazu verführt, Variablen falsch zu belegen, so dass von den 1374 Klauseln 12 unerfüllt bleiben. Um diese Grenze von 1362 Klauseln zu überschreiten, muß eine Vielzahl von Variablen umbelegt werden.

4.2 Vergleich mit Algorithmus ohne Selektoren

Als erstes Experiment wurden ein genetischer Algorithmus mit einem Selektor und einer mit 8 Selektoren verglichen. Um ein Gefühl für die Schwierigkeit zu bekommen, wurde mit verschieden großen Populationen gearbeitet. Die Anzahl der Eltern ist auf der x-Achse aufgetragen. Die Rechnungen liefen jeweils 800 Generationen, in jeder Generation wurden doppelt so viele Kinder wie Eltern erzeugt. Es ist der Durchschnitt über 50 Läufe dargestellt.

In Abbildung 5 ist deutlich zu erkennen, dass der Algorithmus mit einem Selektor Probleme hat, die Grenze von 1362 Klauseln zu überwinden, und ihm dies erst bei größeren Populationen gelingt.

Mit 8 Selektoren wird schon bei einer Populationsgröße von 25 Eltern eine deutliche Verbesserung erzielt und die Grenze von 1362 Klauseln wird deutlich überschritten.

In Abbildung 5 war die Anzahl der Generationen auf 800 beschränkt. Es stellt sich die Frage, wie schnell die Algorithmen der Lösung entgegenstreben. Zu diesem Zweck wurden 50 Läufe über 6400 Generationen durchgeführt, die in Abbildung 6 zu sehen sind. Es ist deutlich zu erkennen, dass im Mittel der Algorithmus mit 8 Selektoren wesentlich besser ist und schneller gute Individuen produziert.

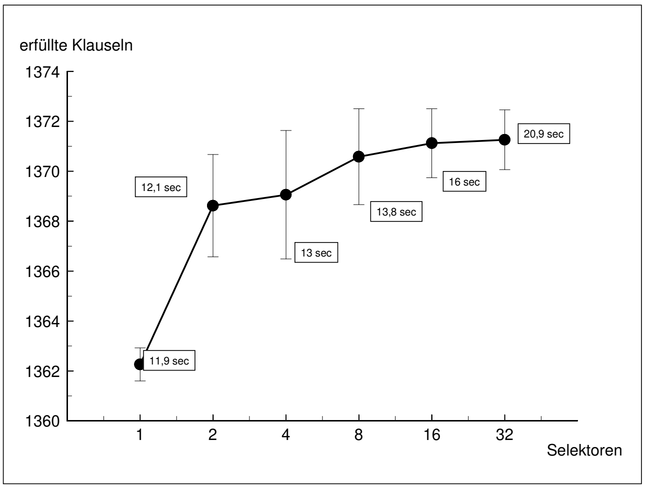

Abbildung 7: Abhängigkeit der Güte von der Anzahl der Selektoren. Dargestellt ist der Mittelwert über 50 Läufe und die Standardabweichung. Die Population umfasst 50 Eltern und 100 Kinder.

4.3 Auswirkung der Anzahl der Selektoren auf die Güte von Lösungen

In Abbildung 7 ist die Güte der Lösungen in Abhängigkeit von der Zahl der Selektoren dargestellt. Der Lauf wurde mit 50 Eltern und 100 Kindern durchgeführt. Auffallend ist der Sprung um 6 Klauseln zwischen einem und zwei Selektoren. Mit steigender Zahl der Selektoren kann die Güte der Lösung nur noch marginal um 1 bis 2 Klauseln verbessert werden. Eine Erhöhung der Selektoren über 2 hat jedoch einen geringen Einfluss auf die Güte als eine Vergrößerung der Population.

An den Rechenzeiten, die ebenfalls in Abbildung 7 zu sehen sind, ist zu erkennen, dass mehrere Selektoren wenig zusätzlichen Aufwand erfordern. Mit wenig zusätzlichem Aufwand lassen sich wesentliche Verbesserungen erreichen.

4.4 Weitere Probleme

Zusätzlich zum obigen Problem wurden weitere Instanzen getestet. Die Instanzen besteht aus 800 Klauseln und 100 Variablen. Die Instanzen entstammen einem Zufallsgenerator, der schwere Instanzen erzeugt. Alle Klauseln sind erfüllbar. In Tabelle 2 sind die im Mittel erfüllten Klauseln von 50 Läufen aufgelistet.

Problem	1 Selektor	16 Selektoren
jnh201	798,92	799,34
jnh204	798,22	798,84
jnh207	797	798,06
jnh210	798,84	799,28

Tabelle 2: Im Mittel erfüllte Klauseln für einen Selektor und 16 Selektoren im Vergleich.

Beide Algorithmen kommen sehr nah an die optimale Lösung von 800 Klauslen heran. Allerdings kann der genetische Algorithmus mit 16 Selektoren einen leichten Vorteil für sich verbuchen, der sich im Bereich von 0.4 bis einer Klausel bewegt.

4.5 Diversität

Da ein Algorithmus mit mehreren Selektoren bessere Resultate liefert, besteht die Vermutung, dass die Diversität in der Population größer ist. Die Diversität einer Population wird definiert als die Anzahl der Variablen, die nicht in allen Individuen mit dem gleichen Wert belegt sind. In Abbildung 8 ist zu sehen, dass die zufällig erzeugte Population am Anfang eine Diversität von 100 hat, da jede Variable mit 50% Wahrscheinlichkeit mit 1 oder 0 belegt wird. Im Laufe der Rechnung mit einem Selektor sinkt die Diversität sehr schnell innerhalb von 50 Generationen unter 25. Bei 16 Selektoren jedoch kann sich die Diversität auf einem Niveau von 40 bis 50 halten.

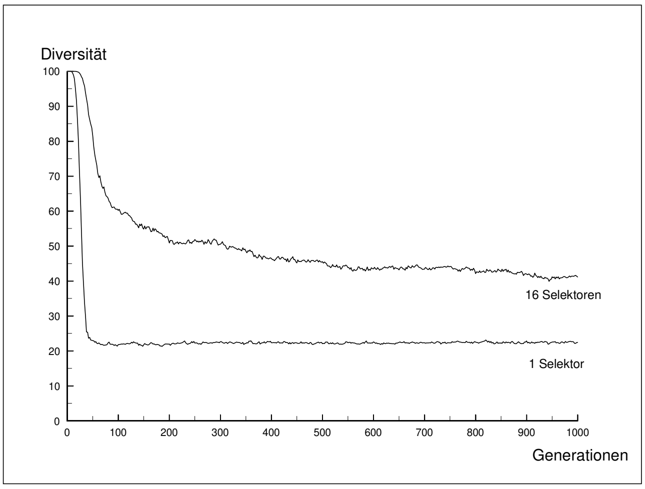

Abbildung 8: Diversität der Population für das Problem *jnh207*. Dargestellt sind der Mittelwert über 50 Läufe mit einem und mit 16 Selektoren.

5 Fazit und Ausblick

Zusammenfassend kann man sagen, dass durch den Einsatz von mehreren Selektoren die Individuen vielfältigen Bedingungen ausgesetzt werden und die Diversität dadurch erhöht wird. Eine andere Sichtweise ist es, dass die zu optimierende Funktion zerlegt wird, und dass jeder Selektor eine Teilfunktion optimiert, die unter Umständen einfacher ist.

In der momentanen Implementierung werden Klauseln zur Teilmenge eines Selektors ausschließlich hinzugefügt. Beim adaptiven Zuwachs kann es passieren, dass die Berechnung stagniert. In diesem Fall könnte man versuchen, die Berechnung wieder in Gang zu bekommen, indem die Teilmenge des Selektors mit neuen Klauseln, z. B. mit den bisher unerfüllten Klauseln, gefüllt wird.

Bei der parallelen Auswertung der Klauselmenge ist noch offen, wie die Klauseln auf die Prozesse verteilt werden müssen, um einen guten Speedup zu bekommen. In dem verwendeten Ansatz werden die Klauseln einfach blockweise aufgeteilt. Kommt es zu einem Ungleichgewicht, so dass z. B. die Klauseln, in denen Variable x_1 vorkommt, alle einem Prozess zugeordnet werden, so muss dieser Prozess die gesamten Berechnungen bei Änderung von x_1 durchführen. Es ist also gefordert, dass die Klauseln so in Teilmengen aufgeteilt werden, dass jede Variable gleich häufig in jeder Teilmenge vorhanden ist.

Der genetische Algorithmus wurde bisher noch nicht parallelisiert. Hierzu böte sich das Inselmodell an, in dem es mehrere Populationen gibt, die gelegentlich Individuen austauschen. Hier könnte es sinnvoll sein, bei einem Selektor mit einer Teilmenge der Klauseln zu starten. Interessant ist die Frage, ob ein Selektor pro Inselpopulation schon eine Verbesserung gewährleistet oder nicht.

Literatur

[B75] K.G. Beauchhamp. **Walsh Functions and their Applications.** Academic Press 1975.

[B96] T. Bäck. **Evolutionary Algorithms in Theory and Practice.** Oxford University Press, 1996.

[BP96] R. Battiti, M. Protasi. **Solving MAXSAT with non-oblivious functions and history-based heuristics**, DIMACS Workshop on Satisfiability Problem, Rutgers University, 1996.

[BP98] R. Battiti, M. Protasi. **Approximate algorithms and heuristics for MAXSAT.** in Handbook of Combinatorial Optimization, D.-P. Du, P.M. Pardalos.(Eds.), Kluwer Academic Publishers, pages 77-148, 1998.

[FPS99] G. Folino, C. Pizzuti, G. Spezzano. **A parallel hybrid method for SAT by coupling genetic algorithms and local search**. Technical Report

[GJ78] M.R. Garey, D.S. Johnson. **Computers and Interactibility. A Guide to the Theory of NP-Completeness.** W.H.Freeman and Company, New York, 1979.

[KR92] A.P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, M.G.C. Resende. **A continous approach to inductive inference.** Mathematical Programming 57, pages 215-238, 1992.

[LGS98] M. Laumanns, G. Rudolph, H.-P. Schwefel. **A spatial predator-prey approach to multi-objective optimization: A preliminary study.** In Agoston E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, (Eds), Fifth International Conference on Parallel Problem Solving from Nature (PPSN-V), pages 241-249, Berlin, Germany, 1998. Springer.

[M96] Z. Michalewicz. **Genetic Algorithms + Data Structures.** Springer-Verlag Berlin Heidelberg, 1996.

[S95] H.S. Schwefel. **Evolution and Optimum Seeking.** John Wiley & Sons, Inc., 1995.

[W93] I. Wegener. **Theoretische Informatik**. Teubner Stuttgart, 1993.