Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Samo Tuma

Combinatorial Optimization of Boolean Satisfiability Problems

diplomsko delo

univerzitetni študijski program računalništvo in informatika

Mentor: dr. Andrej Brodnik

Ljubljana 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01889/2013
Datum: 07.01.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **SAMO TUMA**

Naslov: **KOMBINATORIČNA OPTIMIZACIJA PROBLEMA BOOLEOVE IZPOLNJIVOSTI**

COMBINATORIAL OPTIMIZATION OF BOOLEAN SATISFIABILITY PROBLEMS

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Problem izpolnjivosti (SAT) je v zgodovini računalniških znanosti eden najpogosteje preučevanih problemov. Srečamo ga v središču pozornosti teoretičnih raziskav (prvi problem, za katerega je bilo pokazano, da je NP-poln), kot tudi praktičnih aplikacij (nanj se prevaja vrsta praktičnih industrijskih problemov). Prav zaradi slednjega kljub svoji NP-polnosti potrebujemo njegove učinkovite rešitve, vendar ne v njegovi odločitveni inačici, ampak v praksi običajno rešujemo inačico s kombinatorično optimizacijo (MAX-3SAT).
Ker je problem MAX-3SAT še vedno NP-poln, potrebujemo učinkovite meta-hevristične metode, od katerih je verjetno ena najpopularnejših metoda genetskih algoritmov (GA). V diplomski nalogi preučite rabo GA pri reševanju problema MAX-3SAT. Pri tem posvetite pozornost rabi hibridizacije GA z lokalno optimizacijo in vpliv slednje na kakovost rešitve. Preučite tudi možnost in vpliv povzporejanja programske kode.

Mentor:

doc. dr. Andrej Brodnik

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01889/2013
Datum: 07.01.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **SAMO TUMA**

Naslov: **COMBINATORIAL OPTIMIZATION OF BOOLEAN SATISFIABILITY PROBLEMS**

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Satisfiability problem (SAT) is in the history Computer Science one of the most frequently studied problems. It can be found in the centre of the theoretical research (it was the first problem proven to be NP-complete), and also in practical applications (a number of practical industrial problems are converted into it). This is also one of the major reasons that we need a reasonable solution to the problem in spite of its NP-completeness. Indeed, in practice we are not solving the decision problem, but its combinatorial optimization version MAX-3SAT.
Because MAX-3SAT problem is still NP-complete we require an efficient meta-heuristic method. Probably the most popular such method are Genetic Algorithms (GA). Research the use of GA for solving of MAX-3SAT problem. Particular attention should be payed to the use of hybridization of GA with a local optimization and its influence on the quality of solution. Study also the possibility and influence of parallelization of the code.

Mentor:
doc. dr. Andrej Brodnik

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Samo Tuma, z vpisno številko **63070275**, sem avtor diplomskega dela z naslovom:

Combinatorial Optimization of Boolean Satisfiability Problems

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom dr. Andreja Brodnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 6. februarja 2013 Podpis avtorja:

Svoji družini in prijateljem..

Contents

Povzetek

Cilj diplomske naloge je bil izdelati *hibridni genetski algoritem* za reševanje MAX-3SAT problema. MAX-3SAT problem sodi med *odločitvene* probleme v poglavju računalništva, ki se imenuje teorija kompleksnosti (*ang. complexity theory*). MAX-3SAT problemi sodijo, podobno kot SAT problemi, med NP polne probleme. Podrobne definicije 3SAT problemov in njihovih različic so podane v poglavjih 1.2-1.4.

Motivacija za reševanje omenjenih problemov je precejšnja, saj je poleg praktične uporabnosti možna preprosta prevedba marsikaterega drugega NP polnega problema v 3SAT problem. Področja uporabnosti algoritmov za reševanje 3SAT problemov se raztezajo od računanja zakasnitev na *integriranih vezjih* preko optimizacije *izhodov funkcij* integriranih vezij pa vse do optimiziranja poti signalov znotraj *FPGA* čipov. Več informacij o praktični uporabnosti 3SAT problemov sledi v poglavju 1.9.

Te probleme v splošnem rešujemo na dva načina: s *hevrističnimi* in z *determinističnimi* metodami. Bistvena razlika med pristopoma je, da s pomočjo determinističnih metod dobimo *natančne* rešitve, pri hevrističnih metodah pa je to nekoliko drugače - tu želimo poiskati in optimizirati (zadosti dobre) *delne* rešitve. Podrobnejši opis in primerjava determinističnih in hevrističnih metod se nahajata v poglavju 2. Omeniti velja še dejstvo, da deterministične metode potrebujejo precej več računske moči in posledično več časa, medtem ko hevristične metode veljajo za bolj varčne in zvečinoma precej hitrejše.

Algoritem, ki je bil razvit v sklopu diplomske naloge, je mešane narave ter izkorišča relativno mlado in zanimivo vejo kombinatoričnih optimizacij

- genetske algoritme. Ti algoritmi se zgledujejo po *Darwinovem* naravnem procesu *evolucije* in osnovnih principih *genetike*.

Genetski algoritem deluje tako, da začne z naključno kombinacijo vrednosti spremenljivk konkretne 3SAT instance problema, nato pa postopoma - v nekaj sto generacijah - razvija delne rešitve problema. To počne z uporabo rekombinacije in mutacij če pa dodamo še *lokalne iskalne procedure* dobimo takoimenovani *hibridni* genetski algoritem. Algoritem je podrobno opisan v poglavju 3.

Pri testiranju algoritma, se pojavi vprašanje, kaj v resnici pomeni težek 3SAT problem? Obstajajo zelo usmerjene študije, ki so uspešno dokazale da med najtežje 3SAT probleme za kakršnekoli metode sodijo instance problema, ki vsebujejo okoli $4,42$ stavkov na eno spremenljivko ([8]). Še več, intance teh problemov so prosto dostopne ter tako omogočajo preprosto primerjavo rezultatov različnih algoritmov. Več informacij o instancah problemov, ki so bili uporabljeni za testiranje našega algoritma sledi v poglavju 1.8.

Zaradi reševanja velikih instanc 3SAT problema in velikim deležem računsko drage lokalne optimizacije, prav tako pa tudi zaradi želje po čim hitrejšem (povprečnem) času izvajanja, smo v poglavju 3.3 predstavili preprost način povzporejanja lokalne optimizacije. Algoritem lokalne iskalne procedure tako teče na t programskih nitih naenkrat. Rezultati povzporejanja so predstavljeni v poglavju 4.3.

Zanimalo nas je, v kakšnem odnosu, če sploh, sta si kakovost rešitve in odstotek generacij, ki so uporabile lokalno optimizacijo. Izkazalo se je, da smo dosegli najboljšo kakovost rešitve pri 20 odstotkih lokalne optimizacije, vendar je bila le malce (manj kot desetino odstotka) višja od kakovosti rešitve pridobljene pri 5 odstotkih lokalne optimizacije. Povprečni čas izvajanja je bil pri 20-ih odstotkih, približno štirikrat višji od časa izvajanja pri 5 odstotkih lokalne optimizacije. Več sledi v poglavju 4.4.

Zanimalo nas je tudi, v kakšnem odnosu so: porazdelitev lokalne optimizacije znotraj generacij in kakovost rešitve ter povprečni čas izvajanja. Namesto da lokalno optimizacijo vključimo le vsakih m generacij, smo poskusili

z lokalno optimizacijo v segmentih in različno postavitvijo le-teh. Pri velikosti segmenta 20 generacij (kar je ravno 5 odstotkov vseh generacij) smo dobili najboljše rezultate pri postavitvi segmenta lokalne optimizacije med začetne generacije. Ti rezultati in njihova analiza sledijo v podpoglavju 4.4, vendar so bili manj uspešni - izkazalo se je da je naš algoritem najbolj uspešen takrat, kadar je lokalna optimizacija prisotna skupaj z genetskim algoritmom in sicer periodično vsakih u generacij.

Zgolj kot zanimivost navajamo dejstvo, ki je v nasprotju z rezultati disertacije M. Đorđevića [5], v kateri najdemo podobno implementacijo hibridnega genetskega algoritma za TSP. Dejstvo je, da se je naša lokalna optimizacija v *segmentih* najbolje odrezala na začetku generacij in ne na koncu kot je bilo pričakovano iz omenjene disertacije.

Za splošne rezultate z najboljšimi parameteri lahko rečemo, da smo dobili dokaj solidne številke. Naša kakovost rešitve zaostaja za manj kot en odstotek za podobno implementacijo hibridnega algoritma podrobno opisanega v članku [14]. Prav tako je naš povprečni čas izvajanja višji le za približno 30 odstotkov. Bralec si lahko podrobno ogleda rezultate v podpoglavju 4.6.

Ključne besede: MAX-3SAT, hibridni genetski algoritem, lokalna optimizacija, paralelizacija, odločitveni problemi

Abstract

The purpose of this thesis is to design, implement and analyse a *hybrid genetic algorithm* for solving the MAX-3SAT problem. The problem is a well known NP-complete decision problem. Solving this type of problems is highly motivated by its practical use in industry. The main field of 3SAT solver application ranges from *integrated circuit delay optimization* to *FPGA routing*. See section 1.9 for details on practical applications of the MAX-3SAT solver.

The idea behind this hybrid genetic algorithm, is to see just when and how much of local search can still prove beneficial to the overall solution quality. In the results section 4.4, we have shown that this algorithm performs best when the local search is applied periodically[1] and with our best run results (presented in section 4.6) we came very close to a similar implementation described in [14]. Our best average run time on the biggest test instances was trailing behind aforementioned article by about 30 percent. The reason, why our average running time is fairly close to the aforementioned implementation, lies in exploiting parallelism. We used a simple parallelization technique of local searching, which is described in section 4.2. Moreover, our solution quality - that is the number of unsatisfied clauses, was pretty much as good as the one reported in [14]. Ours was worse by only 0.6 percent. It is important to point out that we used *exactly* the same test instances as used in [14].

[1]The number of generations where we applied local search optimization, was best set at 20 percent of total generations. See section 4.4.

Keywords: MAX-3SAT, hybrid genetic algorithm, local search optimization, parallelization, decision problems

List of Algorithms

List of Tables

List of Figures

Chapter 1

Introduction and Background

1.1 NP-Complete Problems

There are numerous problems, computer scientists are faced with nowadays, yet speaking in terms of complexity, some of the hardest problems were discovered many years ago. Probably the most infamous among them is the *Travelling Salesman Problem* (TSP), referenced for the first time, as Knight's tour problem in the 9th century AD. In 1800s, it was recognized and studied by the Irish mathematician W. R. Hamilton and by the British mathematician Thomas Kirkman in the form of a recreational puzzle based on finding the *Hamiltonian cycle*.

In this thesis we are dealing with a special family of problems - NP problems. These problems can be solved in polynomial time on a non-deterministic Turing machine. Indeed, we are interested in the hardest among NP problems and they are called NP-complete problems.

There are more than 3000 known NP-complete problems including those that everyone likes solving like Tetris or Sudoku ([4, 10]). For the purpose of this thesis, the following definition of NP-completeness is well put.

Definition 1.1 (NP-Completeness) *Let L be a language over a finite alphabet $\sum$. L is NP-complete if, and only if the following two conditions are satisfied:*

1. $L \in NP$

2. *any* $L' \in NP$ *is polynomial-time-reducible to* L *(written as* $L' \leq_p L$*, where* $L' \leq_p L$ *if and only if, the following two conditions are satisfied:*

   (a) *There exists* $f : \sum^* \to \sum^*$ *such that* $\forall w \in \sum^* (w \in L' \leftrightarrow f(w) \in L)$ *and*

   (b) *There exists a polynomial-time Turing machine that halts with* $f(w)$ *on its tape on any input* w*.*

Basically the definition 1.1 says that a decision problem C is NP-complete if:

1. C is in NP, and

2. Every problem in NP is reducible to C in polynomial time.

C can be shown to be in NP by demonstrating that a candidate solution to C can be verified in polynomial time. Note that the problem C satisfying *condition 2* is said to be NP-hard, whether or not it satisfies *condition 1.*

A consequence of this definition is as follows: if we had a polynomial time algorithm (on any Turing or Turing-equivalent abstract machine), we could solve all problems in NP in polynomial time ([21]).

1.2 Satisfiability

To satisfy an equation means simply to make the left and right sides of an equation equal.

A comprehensive explanation can be found in the book called Computability and Logic [2]. However, for the purpose of this thesis an abstract from Wikipedia page on Satisfiability will be sufficient [22]:

> In mathematical logic, *satisfiability* and *validity* are elementary concepts of semantics. A formula is *satisfiable* if it is possible to find an interpretation (model) that makes the formula true. A formula is valid if all interpretations make the formula true. The opposites of these concepts are *un-satisfiability* and *invalidity*, that is, a formula is *un-satisfiable* if none of the interpretations make the formula true, and invalid if some such interpretation makes the formula false.

Having briefly defined *NP-completeness* and *satisfiability*, we can now proceed by defining the SAT problem.

1.3 The Boolean Satisfiability Problem

Satisfiability problem (SAT) is a decision problem within the general scope of complexity theory. An instance of SAT is a Boolean expression consisting only of *and*, *or*, *not* operators and logical *variables*. The question we want to answer is whether it is possible to assign *true* and *false* values, to logical variables in such a way, that the entire expression evaluates to *true*.

A formula is said to be **satisfiable** if logical values can be assigned to variables in such a way to make the entire formula *true*.

To formally define the SAT problem we first define the *boolean variables*, *literals* and *clauses*.

Definition 1.2 (Variables, Literals and Clauses) *The Boolean variable is a variable of data type Boolean, capable of representing exactly two values, usually denoted as* $true$ *and* $false$.

A literal is either a variable or the negation of a variable. x_1 *is a positive literal.* $\overline{x_1}$ *is a negative literal.*

A clause is a disjunction of literals.

For example $(x_1 \wedge \overline{x_2} \wedge x_3)$ *is a clause.*

A SAT formula consists of n variables, whose literals are in m clauses, which are connected in a logical conjunction. The entire formula must be in Conjunctive Normal Form (CNF).

Having briefly defined *literals, variables* and *clauses* we can now give a definition of the **boolean satisfiability problem**:

Definition 1.3 (Boolean satisfiability problem) *Given a CNF formula* α *consisting of* m *clauses, find a variable assignment, which satisfies all of the clauses, therefore making the entire formula true.*

For example:
Assign true or false values to variables $x_1, x_2, ..., x_i$ in such a way that, the following equation will be satisfied,

$$(x_1 \wedge \overline{x_2} \wedge x_3) \wedge (x_4 \wedge x_5 \wedge \overline{x_6}) \wedge .. \wedge (x_{i-2} \wedge x_{i-1} \wedge x_i) = true$$

This is a SAT problem.

Complexity SAT was the first NP-complete problem discovered and proved by Stephen Cook in 1971 with the *Cook's* or sometimes called *Cook-Levin theorem*, which (in short) states [19]:

Theorem 1.1 (Cook-Levin) *The Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.*

Solving SAT
In general, there are two branches of algorithms for solving SAT instances. One branch is more ambitious and is attempting to find an *exact* solution to the problem. The other one is less ambitious and therefore finds only partial solutions called *approximations*.

1.4 The MAX-3SAT Problem

The MAX-3SAT problem is a generalization of the *boolean satisfiability problem*. To define the MAX-3SAT problem we first define the 3SAT problem:

Definition 1.4 (3SAT Problem) *Given a CNF formula α containing at most 3 variables per clause, find a variable assignment in such a way that it makes the entire formula true.*

The difference between 3-SAT problem and MAX-3SAT problem is that the 3SAT problem requires finding an exact solution, which MAX-3SAT does not. The MAX-3SAT problem is defined as follows.

Definition 1.5 (MAX-3SAT Problem) *Given a CNF formula β containing at most 3 variables per clause, find a variable assignment that maximizes the number of clauses satisfied in β.*

Complexity of the MAX-3SAT algorithm is still NP-complete, and no polynomial-time solution can be found unless $P = NP$.

1.5 Evolutionary Approach Basics

The goal of this thesis is to compare and make use of a custom MAX-3-SAT solver, that makes use of both the local search space - using local optimization techniques, and global space using the evolutionary algorithm approach.

Evolutionary or *genetic* algorithms are inspired by Darwin's theory on evolution. The model of the algorithm had been introduced by John Holland in 1975. A great definition can be found in Darrell Whitley's paper *A genetic algorithm tutorial* ([18]):

> These algorithms encode a potential solution to a problem on a simple chromosome-like data structure and apply recombination operators to these structures so as to preserve critical information. Genetic algorithm are often viewed as function optimizers, although the range of problems to which genetic algorithms have been applied is quite broad.

The final solution is obtained by evolving the initial solutions. The whole algorithm starts by initializing a set of individual *chromosomes* that form the initial *population*. These individuals get rated according to a *fitness function*. The function calculates, which solutions are better, and which are not as good. The chromosomes with greater fitness value will get a better chance of reproducing in the recombination process, and the ones with poor fitness values will be eliminated. The fitness evaluation and recombination cycle mimics the process of *natural selection* found in nature. The fitness evaluation also ensures the survival of the fittest individuals in the population, which in this case are the promising *partial solutions* of the satisfiability problem. Following are the definitions of those terms for the satisfiability problem.

Definition 1.6 (Chromosome, Population and Fitness) *Chromosome is a bit-string of zeroes and ones, whereas n-th bit represents the* ***value*** *of n-th variable in the problem.*

Population, at any point in time consists of ***unique*** *chromosomes.*

Fitness of a chromosome c is determined by percentage of clauses it satisfies

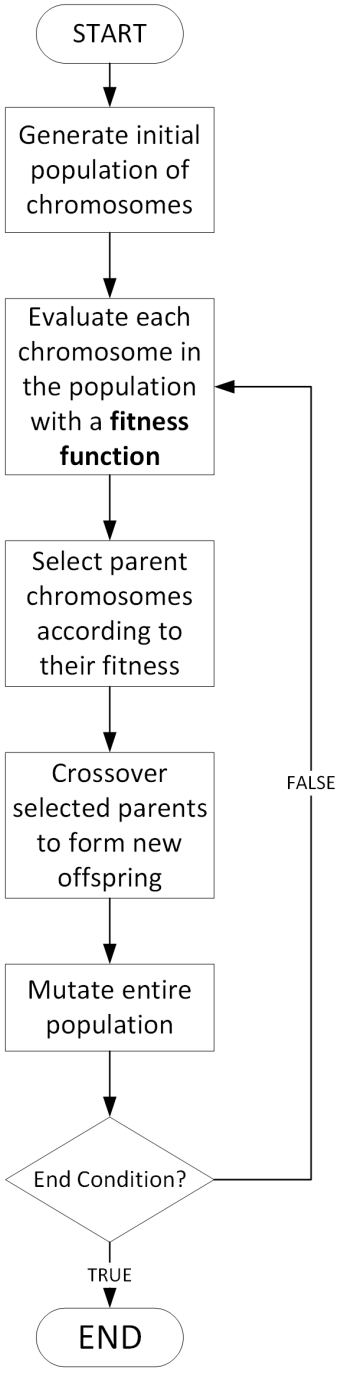

Figure 1.1: Evolutionary algorithm flow chart.

in the entire problem domain P:

$$F(c) = \frac{number\ of\ satisfied\ caluses\ in\ P}{number\ of\ all\ clauses\ in\ P}$$

Fitness function has to be as simple as possible. Because it is evaluated on every iteration of the algorithm for each chromosome, it should take as little computational effort as possible.

1.6 Evolving The Chromosomes

As stated before, after the chromosomes get evaluated they have to be changed in order to yield better offspring - the kind that leads to good solutions. This can be achieved in different ways. For example one could mutate only the reproduced children instead of the entire population, but the change of genes is usually a three step process. It can be seen as the lower part of the flow chart in Figure 1.1.

1. Each chromosome has to either be **selected** as parent or it has to be deleted in order to make space for the offspring.
2. The parents are recombined, generally using a specific **crossover operator**.
3. Each individual in the population gets mutated with probability p.

Note that a few of the fittest parents are never deleted, this process is called *elitism* and is commonly used in many implementations.

1.7 Hybrid Genetic Algorithm Using Local Search Optimization

Genetic algorithms are a great way of exploring vast solution spaces. When searching a solution space we mostly care about the global maxima, except when a less optimal but quickly calculable partial solution is required.

Genetic algorithms tend to get easily stuck in local maximum. If the algorithm got stuck in a local maximum, one could increase the probability of mutating the chromosome, thus randomly forcing the algorithm to redirect the search in another part of the search space. A downside of this technique is, that rather than exploring the local area of the search space, radical movement of the search area is performed and some candidate solutions might not be explored thoroughly enough.

The bottom line is: these algorithms have great properties when it comes to exploring the breadth of the search space, but are at the same time not as good in checking, whether a maximum is global or not.

Local Search Optimization on the other hand it is quite good at quickly exploring the local search area, where a potential solution maximum might be.

So the idea is to combine searching the solution space in breadth - using the *genetic algorithm*, and searching the solution space in depth - using a *local search* technique to see if there might be a solution there. The idea is not new and has been around for quite some time, proving very useful in this area ([3, 12]).

1.8 Test Instances

While looking for different benchmark instances to test the algorithm, a project called *SATLIB* proved convenient ([8]) and was also used for testing the algorithm in this thesis.

It is a collection of benchmark instances, that are clearly defined and on which a number of algorithms were used, so the results are easily *comparable*. They are defined as ([8]):

> Uniform Random-3-SAT is a family of SAT instance distributions obtained by randomly generating 3-CNF formulae in the following way: For an instance with n variables and m clauses,

each of the m clauses is constructed from 3 literals, which are randomly drawn from the $2n$ possible literals (the n variables and their negations) such that each possible literal is selected with the same probability of $\frac{1}{2}n$. Clauses are not accepted for the construction of the problem instance if they contain multiple copies of the same literal or if they are tautological (i.e., they contain a variable and its negation). Each choice of n and m thus induces a distribution of Random-3-SAT instances. Uniform Random-3-SAT is the union of these distributions over all n and m. One particularly interesting property of uniform *Random-3-SAT* is the occurrence of a phase transition phenomenon, i.e., a rapid change in solubility, which can be observed when systematically increasing (or decreasing) the number m of clauses.

The results shown in Figure 1.2 were obtained from a comprehensive study on how to generate hard SAT problems ([16]).

The ratio $\frac{m}{n}$ between clauses and variables used in this thesis is equal to 4.42 *clauses per variable*. Here the average instance hardness is the greatest for both stochastic and deterministic algorithms ([8]).

1.9 Practical Applications of SAT

Regarding the topic of this thesis, one might say: "Who cares if there exists a satisfying assignment for the formula or not." Well one could not be more wrong, for there are numerous practical applications that have been driving the scientific community to further investigate this area, and as a result it has seen some remarkable improvements since the mid 90s.

First of all, there are many problems that can be *encoded* as SAT problems. For instance *graph colouring*, *planning* and *scheduling* problems can

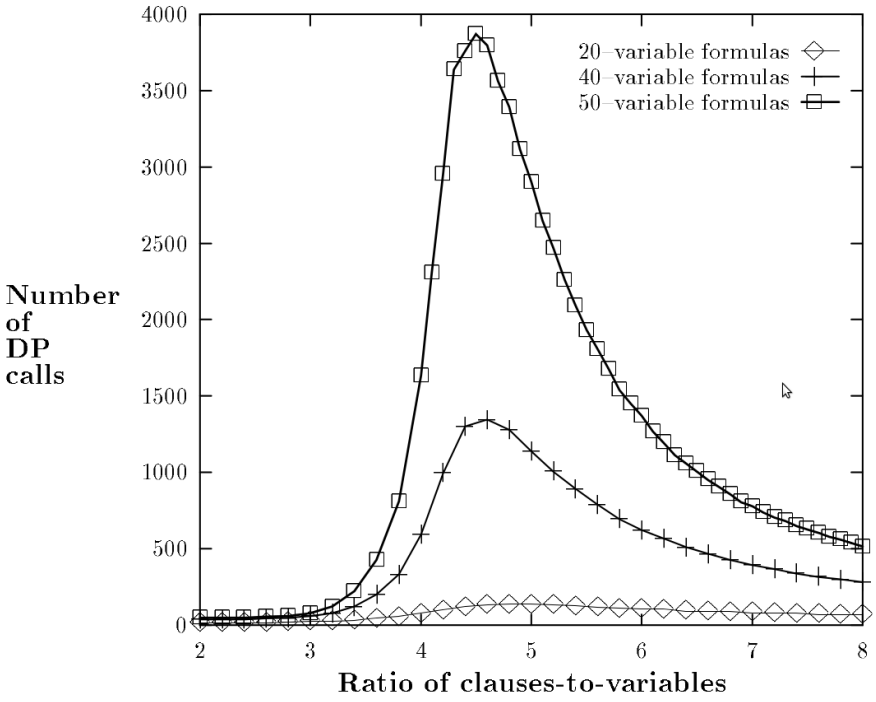

Figure 1.2: Hardness of $\frac{m}{n}$ ratio (denoted as clauses to variable ratio on the x axis). The variable on the y axis is the number of calls to the Davis-Putnam algorithm (Section 2.1).

easily be encoded as SAT problems ([23]), but those were predominantly introduced by the research community. As the main real-world practical uses, where satisfiability solvers are playing and will play a key role in the future of research and development, are:

- **Combinational Equivalence Checking** An essential circuit design task is to check the functional equivalence of two circuits ([13]).
- **Automatic Test Pattern Generation (ATPG)** Fabricated integrated circuits may be subject to defects, which in turn may cause circuit failure. The most popular approach for identifying fabrication defects is automatic test pattern generation (ATPG) ([13]).
- **Field Programmable Gate Array (FPGA) Routing** It is used in many different areas of the FPGA design: architecture evaluation, technology mapping and resynthesis ([9]). These FPGA areas exceed the scope of this thesis, but what is common for SAT solvers in these areas is:

  Problem 1.1 (SAT problem in FPGA design) *Given an n-variable Boolean function, $F(x_1, x_2, ..., x_n)$, does there exists a programmable configuration to a circuit such that the output of the circuit will equal $F(x_1, x_2, ..., x_n)$ for all inputs?*

Other uses include *logic synthesis*, *circuit delay computation* and in crosstalk noise analysis. SAT models have also been used for *functional vector generation*. Note that getting into the specifics of the above applications exceeds the scope of this thesis, mainly because the applied area is complicated and the reader can further investigate it in the literature ([9, 11, 13]).

Chapter 2

State of the Art

2.1 Deterministic Methods

The general motivation to use these algorithms is that they will eventually always give us an exact solution, hence they are called *deterministic* methods. Their downside however, is that they take a lot of time, and even on relatively small problems, they still require a lot of computational effort to calculate the exact solution.

Let's look at some of the best in the field.

Davis–Putnam–Logemann–Loveland (DPLL) Algorithm is a backtracking based search algorithm for solving the satisfiability problem. It was introduced in 1962 and is an improved version of the original *Davis-Putnam algorithm* (1960). DPLL is a highly efficient foundation for most state of the art complete SAT and MAX-SAT solvers.

The algorithm starts by choosing a *literal* and assigning a *truth* value to it. Next, it simplifies the formula, and then recursively checks whether the simplified formula is *satisfiable*. During each step it uses two rules ([20]):

Unit propagation

If a clause is a *unit clause*, i.e. it contains only a single unassigned literal, this clause can only be satisfied by assigning

the necessary value to make this literal true.

Pure literal elimination
If a propositional variable occurs with only one polarity in the formula, it is called pure. Pure literals can always be assigned in a way that makes all clauses containing them true. Thus, these clauses do not constrain the search anymore and can be deleted.

MiniMaxSAT makes use of the best current SAT and MAX-SAT techniques. It can handle pseudo-boolean objective functions and constraints. Following are its main features ([7]).

> ...learning and back-jumping on hard clauses; resolution-based and substraction-based lower bounding; and lazy propagation with the two-watched literal scheme.

It is currently the best exact solver, and has won the *The Seventh Evaluation of Max-SAT Solvers (Max-SAT-2012)* competition ([15]), where the goal is assessing the *state of the art* in the field of MAX-SAT solvers ([7]).

2.2 Stochastic Methods

Stochastic means *randomly* determined. These methods are very different from deterministic methods mentioned in section 2.1. They give no guarantee that an optimal solution will be found, because the entire search space is never traversed. However, they are very good at converging to a local or global maximum and are therefore much faster. They have developed greatly in the last years and have the ability of solving hard combinatorial problems like SAT or Travelling Salesman Problem.

Greedy SAT (GSAT) is a greedy local search procedure for solving propositional satisfiability problems. It can be used to solve hard SAT problems ([1]). Its basic idea can be best seen as pseudocode:

1. Assign a random value for each variable
2. If all clauses are satisfied, **return** Solution
3. Pick a variable with maximal *gain*[1]
4. If maximum number of iterations has not been reached, go to step 2.

Depending on the implementation, the algorithm may start from a new randomly generated variable assignment if some number of flips has been reached. This algorithm is very effective on problems that are too hard for deterministic methods such as DPLL ([1]).

WalkSAT is an improved version of GSAT and it has an added mechanism to avoid local maxima called *random walk*. The only difference is its way of choosing a variable to flip.

1. Assign a random value for each variable
2. If all clauses are satisfied, **return** Solution
3. Pick a random unsatisfied clause.
4. With probability p pick a random variable (amongst variables in previously selected *clause*), and with probability $1 - p$ pick **greedily**[2].
5. go to step 2.

Instead of every time greedily picking a variable with the most gain, the greediness is reduced to confining the variable pick set to a subset of variables found in a randomly picked unsatisfied clause. So the search is not entirely random. This algorithm is currently amongst the best *stochastic* algorithms for the SAT problem.

[1]The *gain* of a variable x is equal to the difference between the number of satisfied clauses, if we flip the value of x.

[2]Actually the algorithm in this case performs exactly like GSAT and with probability $1 - p$ deterministically picking a variable with the largest *gain*.

Chapter 3

Architecture and Design

This chapter intends to clarify all of the implementation phases, from *design* to *architecture* decisions and finally to the actual runs and their parameters.

In the thesis we used *modular* approach of designing software, following *object-oriented-programming (OOP)* directives. The reason was to try out different approaches during the algorithm implementation. For example, the program had to implement three different genetic operators to see, which would best fit the particular problem. Also different *local search techniques* were tested.

The reader can learn more about object oriented programming online ([24]). In the following section basic understanding of this concept is assumed.

3.1 Algorithm design

In the following section we present an overview of both algorithms used together to form a *hybrid* genetic algorithm. Further description of methods, data structures and parameters, used in Algorithms 3.1 and 3.2, are explained in sections 3.2 to 3.2.2.

Input: I - 3SAT problem instance
R - local search range
$MAXGEN$ - maximal number of generations
Output: Chromosome C

```
1  randomly generate initial population P;
2  evaluate fitness of P;
3  i = 0;
4  while i < MAXGEN do
5      recombine P using Uniform Crossover Operator;
6      mutate P;
7      if i in range R then
8          local search procedure (P,I);
9      end
10     evaluate fitness (P,I);
11     increment i;
12 end
13 return Best chromosome C from P
```

Algorithm 3.1: Genetic algorithm

Local search procedure is the algorithm that was used as a local search heuristic in genetic algorithm (line 8) described above (Algorithm 3.1). This algorithm keeps flipping a variable, chosen at random, amongst unsatisfied clauses, until a solution is found or the maximal number of iterations was reached.

It is best shown as pseudo-code. (Algorithm 3.2)

3.2 The architecture

The basic architecture of the developed software is shown in Figure 3.1. The program consists of a *mini-framework* where all the shared methods and

Input: Chromosome,3SAT problem instance, maximal number of iterations
Output: Exact solution, if found.

```
1  while maximal number of iterations hasn't been reached do
2      randomly pick a unsatisfied clause M;
3      randomly select a variable N from clause M;
4      Flip the value of N;
5      Evaluate the chromosome C;
6      if Chromosome C is a solution then
7          return;
8      end
9  end
10 put the chromosome C back to population;
```

Algorithm 3.2: Local search procedure

data structures reside. A separate *control class* makes use of the frameworks functionalities. So, whenever the control class wants to perform a basic task, like loading a new test instance from a file or evaluating a specific chromosomes or the entire populations fitness, it simply calls those methods from the framework.

3.2.1 The Mini-Framework

Let's look at the class structure of the *mini-framework*:

Clause contains a simple boolean array with three integers, that together form a *clause*. An array of

$$[1, -2, 3]$$

represents a clause

$$(x_1 \wedge \overline{x_2} \wedge x_1)$$

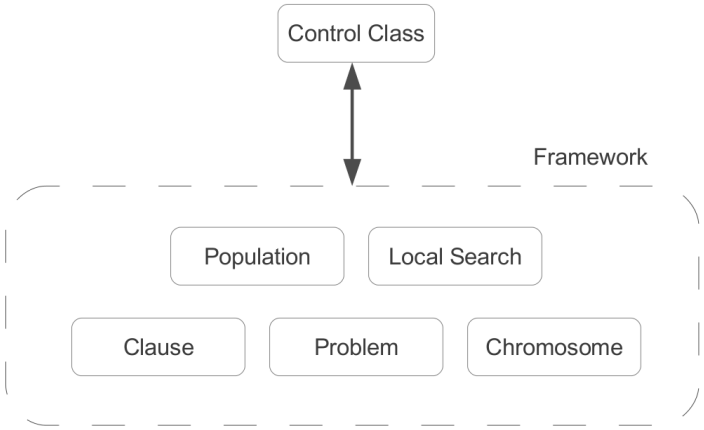

Figure 3.1: Basic Software Architecture

This class has only one *method*, which checks whether a particular chromosome satisfies this specific clause.

Problem is responsible for loading and keeping the main data structure intact. The main *data structure* is a $C++$ vector of clauses. The file with an instance is loaded into this data structure once this class has been instantiated.

Chromosome is a *representation* of a *partial solution* to the problem. When instantiated it generates a partial solution, which is a vector of boolean variables. Each gets assigned a random value. It is able to rate itself against a problem, by calculating the fitness function, with a

$$bool\ rateChromosome(Problem)$$

method, which returns *true* only in case this chromosome satisfies all the clauses in the problem. However, if complete satisfaction cannot be achieved, this class keeps score on how many clauses it was able to satisfy - the fitness of a chromosome.

The chromosome class can also **mutate** itself by:

$$void\ Mutate(double)$$

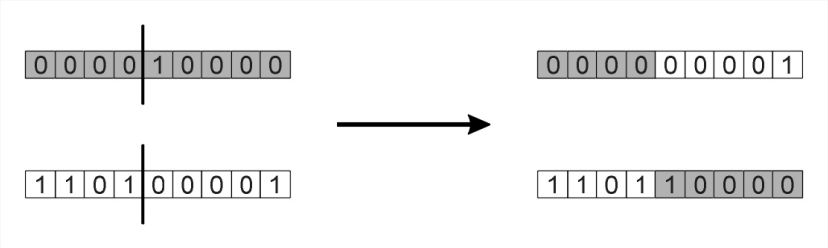

Figure 3.2: One-point crossover operator ([6]).

where the number passed to method tells how strong these mutation ought to be.

Population class contains a vector of chromosomes. During the reproduction process it is able to use *uniform* and *masked* crossover operators. It can evaluate the fitness of the population and trigger local search procedure for each chromosome in the population.

After every *reproduction-mutation-evaluation* cycle it sorts the entire population of chromosomes according to their fitness evaluations.

Uniform Crossover is the type of crossover operator used during the *recombination* process. This is not the most commonly used crossover operator out there, but it has several advantages over the commonly used *one* or *two point* crossovers, which generally work by randomly selecting one or a two points and swapping entire strings of genes between those points. This can be seen in Figures 3.2 and 3.3. However, their behaviour gives us no control over the *percentage* of interchanged *genes*.

Rather than the number of segments exchanged between parent chromosomes, **uniform crossover** exchanges the same amount of genes each time. That proved to be very beneficial ([17]). It works by exchanging a fixed *portion* of genes instead of a fixed segment. This can be seen in Figure 3.4.

Local-Search is a class that is an implementation of the local search heuristic, which runs for the given number of iterations. The procedure used is

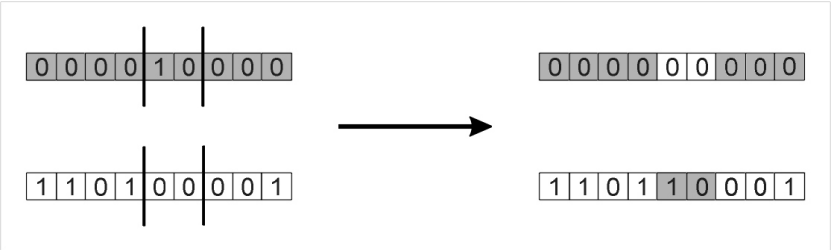

Figure 3.3: Two-point crossover operator (Modified Figure 3.2 from [6]).

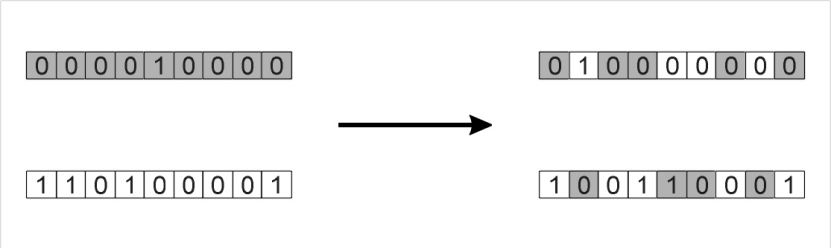

Figure 3.4: Uniform crossover operator ([6])

in Algorithm 3.2. Unlike genetic algorithm this procedure *deterministically* flips a variable in one of the *unsatisfied* clauses. Defining the framework and its basic tasks we can proceed by defining the *control class* and its responsibilities.

3.2.2 The Control Class and Run Parameters

The control classes main method gets called with a few parameters and then it calls the appropriate methods from the *framework*. Some of these parameters have already been mentioned, but in order to represent the results in the next chapter, they will be defined more systematically:

1. $ga\_max\_generations = 300$ - the maximum number of generations that evolutionary algorithm is allowed to run for. This, in most cases, will be an *end condition* for the algorithm. The other type of end condition is a *complete solution*, which in larger test instances rarely happens.

2. $ga\_population\_size = 120$ - the number of *unique chromosomes* representing a *population*.

3. $uc\_percentage\_pop = 25$ - the percentage of population that is to be recombined using **uniform cross-over** recombination technique described in the previous section.

4. $uc\_common\_genes = 50$ is the percentage of genes that (on average) will be exchanged between *parent* chromosome and *offspring*.

5. $ga\_percentage\_of\_genes\_mutated = 10$ - the percentage of genes in a chromosome that will get their value flipped during the *mutation process* in evolutionary algorithm.

6. $number\_of\_iter\_ls = 8000$ - the maximum number of iterations that local search is allowed to run for.

7. $start\_local\_search\_gen$ & $end\_local\_search\_gen$ - two numbers representing a valid range of generations inside a run. For example, if the number $ga\_max\_generation$ is set to 100 then one could set these values to 60 & 70, which would mean that the local search would only be run from generation number 60 through 70. This is necessary, because we were studying the difference in results, if we change this local search range.

8. $local\_search\_frequency = 1$ - the number is used when in *local search mode*, the algorithm will perform local search method exactly *reciprocal* number of times per genetic algorithm *iteration*(generation). This is useful if we want to have different local search window sizes.

9. $number\_of\_threads\_ls = 8$ - the number of concurrent threads, that will be used during the local search procedure. More about parallelization used in this program can be found in section 3.3.

In the following chapter, the results based on these *parameters* will be presented.

3.3 Parallelization

Most implementations of genetic algorithms are pruned to parallelization ([14],[3]). The reason for that is that it is always possible to divide the *population* of chromosomes into *sub-populations* prior to evaluating their costly fitness functions. Then, each *sub-population* gets evaluated in parallel. In our case it turned out that *most* of the time was spent calculating the local search heuristic. Also, the genetic algorithm used little computational effort as it was dealing with very small populations, consisting of only 120 individuals. Therefore, the *parallelization* has been applied to the local search, instead of the genetic algorithm. This was also achieved by dividing the population into t sub-populations prior to running the local search heuristic.

The algorithm was parallelized using $number\_of\_threads\_ls$ separate program threads to better manage the workload. At each iteration, it is able to run $number\_of\_threads\_ls$ concurrent instances of the local optimization class, see Algorithm 3.2. The best performance is achieved by setting $number\_of\_threads\_ls$ to the number of CPU cores.

Chapter 4

Results

In the following chapter, results of the *test runs* using *parallelization* and *hybridization*, will be presented, therefore it is reasonable to divide this chapter into five sections: description of *testing environment*, performance of a *genetic algorithm run* without hybridization, influence of *parallelization*, influence of *hybridization* and the best *quality of solution* obtained.

4.1 Testing environment

Before presenting the test results, we describe the testing *environment* and test instances used for testing. Testing environment is defined by the *hardware* of the target computer, its operating system and different software packages it provides. The computer used for testing was a Ubuntu Linux (3.2.0-34-generic kernel) machine with an Intel©Xeon(R) W3565 Quad-Core CPU with *Hyper-Threading* Technology enabled. We compiled the program with g++ (version 4.6.3) compiler using the command:

```
user@ubuntu# g++ -I ./HybridGA/framework -I \
/Documents/apps/boost_1_52_0 -O3 -Wall -c -fmessage-length=0
```

Note, the *-O3* flag, which tells the g++ compiler to optimize the resulting binaries for maximum performance.

For implementation of the algorithm, we used C++ programming language. As described before in Chapter 3, both genetic an local search algorithm make use of *pseudo-random* number generators, which are not a part of a standard C++ library. That is the reason we also used *Boost* C++ Libraries (Stable version 1.52.0) . The *Boost Random Library* provides a variety of random generators and distributions to produce random numbers having properties such as *uniform distribution*, which is the one used throughout the implementation.

Uniform random-3-SAT Instances All testing was done using SATLIB test instances ([8]). The test-sets used in this thesis, are sampled from the phase transition region described in Chapter 1.8. The test instances are divided into benchmark test-sets, according to their number of variables n and clauses m. The characteristics of these test-sets are shown in Table 4.1 ([8]). The highlighted rows in the table indicate the test-set instances used for testing of our algorithm and note that due to time limitations we could not test the entire test-sets. Instead we tested our algorithm on a subset of test-sets. For example, we took the first 50 test instances from test-set *uf20-91*, which contains 1000 test instances, and ran all subsequent tests on this sub-set. We used the same approach for all the highlighted rows in the Table 4.1, thus obtaining smaller sub-sets.

In the following sections 4.2 to 4.6 we present results obtained by using the above test environment and the sub-sets of SATLIB test instances.

4.2 Typical genetic algorithm performance

Before we introduce hybridization into genetic algorithm, we present a typical genetic algorithm run results (Figure 4.2) and run parameters (Table 4.2). Observe the *improvements* to the quality of the solution evolved in the *early*[1] generations. However, later little improvement is seen. The reason for this

[1]With this algorithm, early means about a third of the generations.

test-set	# of instances	# of instances tested	n	m	ratio $\frac{m}{n}$
uf20-91	1000	50	20	91	4.55
uf50-218	1000	25	50	218	4.36
uf75-325	100	0	75	325	4.33
uf100-430	1000	10	100	430	4.30
uf125-538	100	0	125	538	4.29
uf150-645	100	0	150	645	4.30
uf175-753	100	0	175	753	4.30
uf200-860	100	0	200	860	4.30
uf225-960	100	0	225	960	4.26
uf250-1065	100	10	250	1065	4.26

Table 4.1: SATLIB test-sets, gray rows indicate the test-sets used for testing.

behaviour lies in the nature of *genetic algorithm*, and for the represented chart one could often say that it is typical for the genetic algorithms – at least the shape of the graph is characteristic. Genetic algorithms, however reach the maximum quality of the solution in different points in time for different problems and with respect to parameters they were given.

parameter	**value**
ga_max_generations	300
ga_population_size	120
uc_percentage_pop	25
uc_common_genes	50
ga_percentage_of_genes_mutated	10
number_of_iter_ls	0
start_local_search_gen	0
stop_local_search_gen	0
local_search_frequency	0
number_of_threads	1

Table 4.2: Parameters used in a typical genetic algorithm run, without the use of hybridization (see Section 3.2.2 for full parameter description).

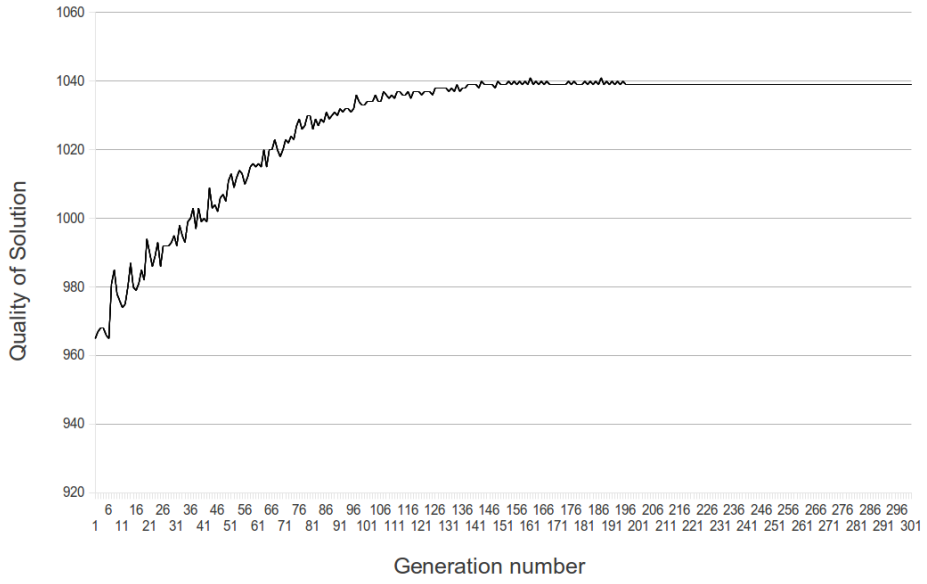

Figure 4.1: Number of satisfied clauses in a typical GA run with 250 variables and 1065 clauses test instance.

4.3 Influence of parallelization

From Figure 4.2 we see the improvement in average execution time when using the parallelization technique, used on local optimizer and described in section 3.3. There is a reason why we can not see any significant improvements after the *fourth* CPU core. This is because of the fact that the actual testing computer only had 4 *cores* available and another 4 simulated cores (Section 4.1). The simulated CPU cores have limited effect in comparison to the real CPU cores, yet they still bring the running time further down.

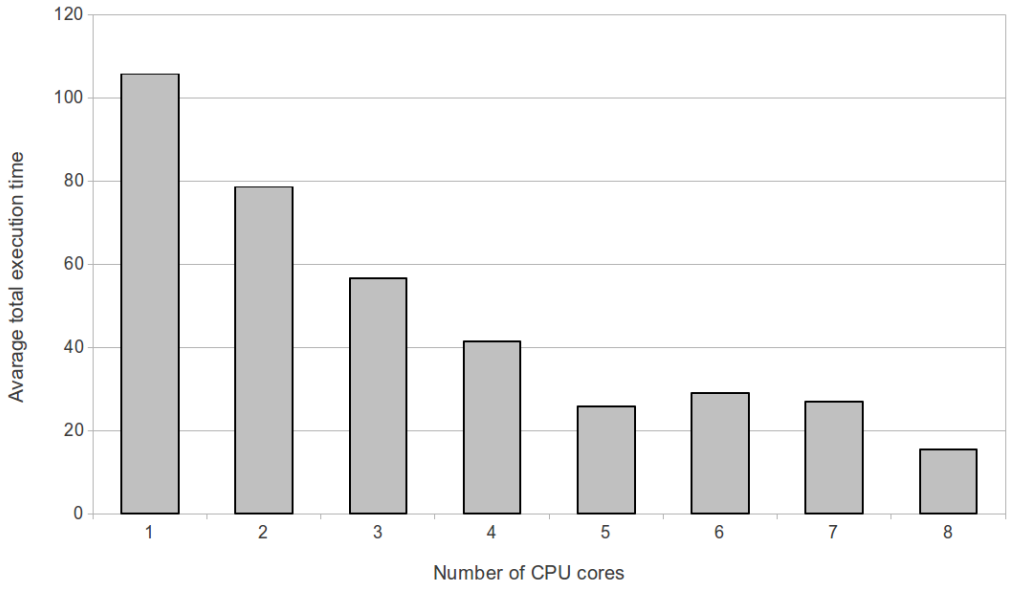

Figure 4.2: Influence of parallelism used in local search procedure.

4.4 Influence of Hybridization

After performing a run without the use of *hybridization*, we gradually rose the percentage of hybridization applied by 5 percent (Figure 4.3). Each time hybridization was applied it either found a complete solution, or it was stopped after 8000 iterations. If hybridization did not return a complete solution and was stopped, the resulting chromosome was put back into genetic algorithms population.

We see a reasonable *compromise* between *average running time* and *percentage* of *un-satisfied* clauses, at around 5 percent of hybridization. Indeed we achieved lower percentage of un-satisfied clauses at 20 percents of hybridization, although the difference was only about 0.1 percent. The average running time, however, was about 4 times longer than the running time achieved with 5 percents of hybridization. Similar testing was also performed in a paper on a hybrid genetic algorithm, which makes use of local search methods for solving the TSP (Travelling salesman problem) ([5]). There the

best level of hybridization applied, was also at around 5 percent of the total number of generations.

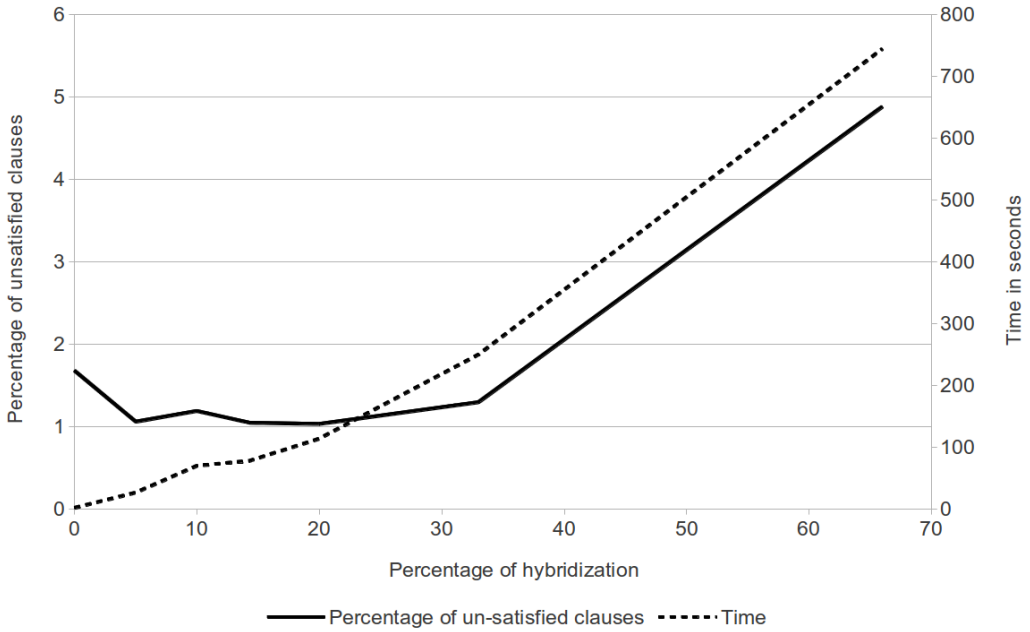

Figure 4.3: Running time and percentage of un-satisfied clauses with respect to percentage of hybridization applied within a run.

After settling for 5 percent of hybridization, we wanted to further improve the results. The idea, whether different hybridization placements would be beneficial, was based on a similar research [5]. We placed hybridization in a single section within the generations, then we moved this section around. To put it simply, we were running these local searches only in the early, middle and late generations - after, during or before the genetic algorithm has done some work on its own.

4.5 Sectional Hybridization

Sectional *hybridization* run results were obtained by running four types of runs, with parameters formally defined in section 3.2.2. These four types

have one thing in common, they all use exactly 5 percent of hybridization within a run. The difference between them is, where this hybridization is placed. We placed the hybridization:

- in the beginning of a run,
- in the middle of a run,
- at the end of a run,
- periodically every 20 generations.

The results of *sectional* hybridization run, are presented in Figure 4.4. However, the results we got, were the opposite of those reported in the aforementioned research ([5]). They reported the best results with hybridization placed in the final generations of a run. In our case it turned out that, if the local search procedure was used in sections, instead of periodically, we got the best results with hybridization used in early generations.

Still, we achieved our best results with 5 percent of hybridization applied periodically.

4.6 Quality of solution

Our hybrid genetic algorithm achieved the best quality of solution with parameters presented in Table 4.3.

The first column in Table 4.4, indicates the size of the test instance set, used to obtain the results. When looking at the results from Table 4.4, we observe a nice average satisfied number of clauses – the *solution quality*. The average number of satisfied clauses is quite close to a solver described in [14]. The aforementioned implementation is similar to ours, but makes use of a different parallelization technique, by utilizing a *graphical processor unit* (GPU) instead of a CPU, they achieved a much higher degree of parallelism. The article also reports using the same test instances found in the SATLIB library ([8]), which makes the article very good for comparison. The article

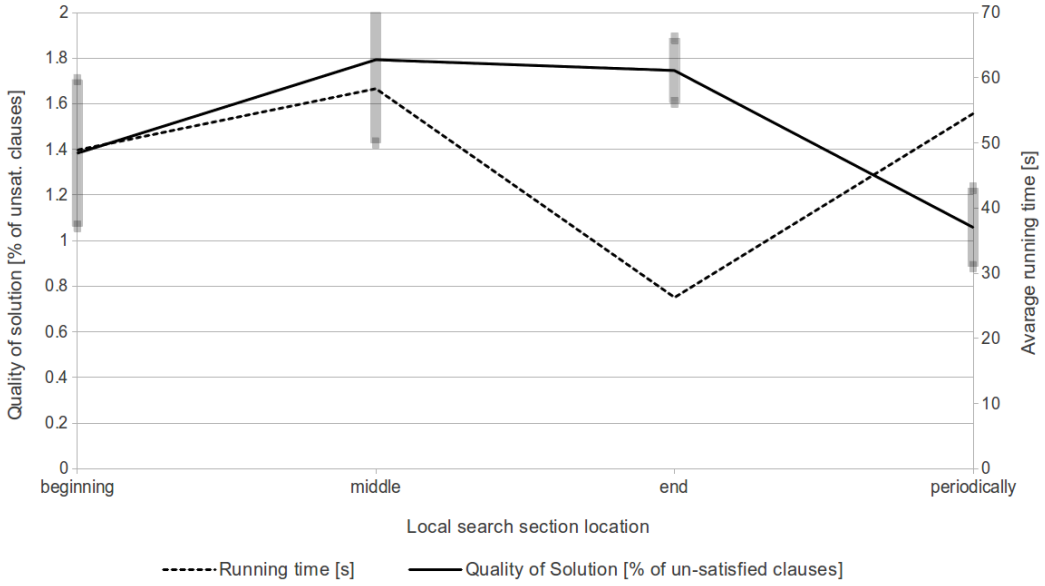

Figure 4.4: Local search placement with respect to average running time and average quality of the solution. Note, the gray lines represent cumulative standard deviation of the acquired data.

reports 1060 out of 1065 as the best average number of satisfied clauses in the biggest test instances, whereas with 1053 out of 1065 (Table 4.4) our *hybrid MAX-3SAT solver* is not far behind. Our *quality of solution*, that is the number of clauses satisfied, was trailing behind by approximately 0.6%.

The average running times also compare nicely to the ones found in [14]. They used slightly different parameters, and a different local search technique called *hill-climbing*. Because they used a much higher degree of parallelism, they achieved a better running time of around $37s$ for the 250 variable test instance. Our hybrid genetic solver achieved the best (average) running time at around $53s$ for the 250 variable problem size.

parameter	**value**
ga_max_generations	300
ga_population_size	120
uc_percentage_pop	25
uc_common_genes	50
ga_percentage_of_genes_mutated	10
number_of_iter_ls	8000
start_local_search_gen	0
stop_local_search_gen	300
local_search_frequency	20
number_of_threads	8

Table 4.3: Parameters used to obtain the best quality of solution and average running time (see Section 3.2.2 for full parameter description).

# of tested instances	variables n	clauses m	ratio $\frac{m}{n}$	Avg. # of satisfied clauses ± std. dev.	Average run-time ± std. dev.[s]
50	20	91	4.55	90.4583 ± 0.7096	4.1170 ± 4.7347
25	50	218	4.36	215.6938 ± 1.8072	17.2952 ± 10.7056
10	100	430	4.30	424.3667 ± 3.6434	32.7512 ± 20.0031
10	250	1065	4.26	1053.2727 ± 11.7140	53.6883 ± 50.9915

Table 4.4: Best run results with local search applied periodically, 5 percent of the generations.

Chapter 5

Conclusions and future work

The purpose of this thesis was to study the influence of *hybridization* on a genetic algorithm when solving the MAX-3SAT problem.

It would be very useful to have some kind of rule or definition on when it is good to use local optimization with genetic algorithms, but to the best of my knowledge there is not. For each problem there are different genetic parameters and/or local optimizations that work best and it is up to the algorithm developer to tweak those parameters in such a way, that they give good results.

In the results section 4.4, we have shown that at 5 percent of local search within a run, considering both the running times and the solution quality, our algorithm is the most efficient. With denser application of local search the running times get longer, especially from 30 percent up, we observed a gradual rise in the average running time. This is due to the computationally expensive local search procedure.

Also from 30 percent up, we see a rise in the average number of *unsatisfied* clauses. This came as a surprise to us, as we expected, that the time consuming local search would prove to be much more effective on maximizing solution quality.

For the purpose of comparison to other similar implementations, we decided on a compromise between running time and solution quality. After

settling for a *solution quality* versus *running time* compromise at about 5 percent of generations running local searches, we have experimented with different placements of local optimization within the evolutionary run. See Figure 4.4.

In our case it seems to be more beneficial to use local search procedure periodically every few generations of genetic algorithm, rather than having local search occurrences applied one after another together in bigger *sections* of generations.

In section 4.6 we presented the best quality of solution achieved using our *hybrid genetic solver*. The results are fairly close to a similar implementation described in [14]. They report a better average running time by around 30 percent and also a better quality of solution by 0.6 percent. For more details see section 4.6.

For future work in this field of combinatorial optimization for hard 3SAT problems, there are numerous open questions and challenges:

- The behaviour of our algorithm depends on a number of parameters. One could use some other optimization technique (perhaps another GA) to optimize them.

- The genetic algorithm could have a mechanism to detect, if it got stuck in a local maximum and react to this information. For example, by amplifying mutations or momentarily switching to a more stochastic local search procedure, or by having some random individuals added to the population on each generation.

- We could replace the local search procedure from section 3.2.1, with a form of a more guided search, commonly referred to as *hill climbing*. Essentially, a less *stochastic* local search procedure could be used to force the algorithm to spend more time using deterministic methods, but starting them where the genetic algorithm has left off (hopefully near a global maximum). We believe we might see an improvement of

the quality of solution, however, a rise in the average running time is also expected.

Bibliography

[1] Hector Levesque Bart Selman and David Mitchell. A new method for solving hard satisfiability problems. Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA, July 1992.

[2] Jeffrey Boolos. *Computability and Logic*. Cambridge University Press, 1974.

[3] Rok Cvahte. Povzporejanje metahevristik za np-polne probleme. Master's thesis, University of Ljubljana, 2010.

[4] Erik D. Demaine, Susan Hohenberger, and David Liben-Nowell. Tetris is hard, even to approximate. *CoRR*, cs.CC/0210020, 2002.

[5] M. Djordjević. *Grafted genetic algorithm and the travelling visitor problem*. PhD thesis, University of Primorska, 2012.

[6] A.E. Eiben and J.E. Smith. *Introduction to evolutionary computing*. springer, 2008.

[7] Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat: An efficient weighted max-sat solver. *J. Artif. Intell. Res. (JAIR)*, 31:1–32, 2008.

[8] Holger H. Hoos and Thomas Stützle. Satlib: An online resource for research on sat, 2013. I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000, pp.283-292, IOS Press, 2000.

[9] Andrew C. Ling. *Field-Programmable Gate Array Logic Synthesis Using Boolean Satisfiability*. PhD thesis, Graduate Department of Electrical and Computer Engineering, University of Toronto, 2005.

[10] I. Lynce and J. Ouaknine. Sudoku as a sat problem.

[11] V. Manquinho and J. Marques-Silva. On using satisfiability-based pruning techniques in covering algorithms,, March 2000. in Proceedings of the Design, Automation and Test in Europe Conference.

[12] E. Marchiori and C. Rossi. A flipping genetic algorithm for hard 3-sat problems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 393–400, 1999.

[13] João Marquez-Silva. Practical applications of boolean satisfiability, May 2008. Workshop on Discrete Event Systems (WODES), Göteborg, Sweden.

[14] Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama. Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework. *Genetic Programming and Evolvable Machines*, 10(4):391–415, December 2009.

[15] Fifthteen International Conference on Theory and Applications of Satisfiability Testing. Seventh max-sat evaluation. `http://maxsat.ia.udl.cat`, 2012. [Online; accessed January-2013].

[16] Mitchell D. Selman B., Levesque H.J. Generating hard satisfiability problems, 1995. Elsevier Science.

[17] W.M. Spears and K.D. De Jong. On the virtues of parameterized uniform crossover. Technical report, DTIC Document, 1995.

[18] Darrell Whitley. A genetic algorithm tutorial, 1994. Computer Science Department, Colorado State University.

[19] Wikipedia. Cook–levin theorem — wikipedia, the free encyclopedia, 2012. [Online; accessed January-2013].

[20] Wikipedia. Dpll algorithm — wikipedia, the free encyclopedia, 2012. [Online; accessed January-2013].

[21] Wikipedia. Np-complete, 2012. [Online; accessed October-2012].

[22] Wikipedia. Satisfiability, 2012. [Online; accessed October-2012].

[23] Wikipedia. Boolean satisfiability problem — wikipedia, the free encyclopedia, 2013. [Online; accessed February-2013].

[24] Wikipedia. Object-oriented programming — wikipedia, the free encyclopedia, 2013. [Online; accessed January-2013].