Soft Computing Approaches to DPLL SAT Solver Optimization

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte

Dissertation

von

Dipl.-Ing. Raihan Hassnain Kibria

Geboren am 20.07.1976 in Dacca / Bangladesh

Referent:	Prof. Dr.-Ing. Hans Eveking
Korreferent:	Prof. Dr. Christoph Scholl
Tag der Einreichung:	11. April 2011
Tag der mündlichen Prüfung:	23. September 2011

D 17

Darmstadt 2011

Erklärung laut §9 PromO

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe. Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 11. April 2011

Raihan Hassnain Kibria

Zusammenfassung

Moderne, digitale Elektronik ist so komplex, dass die Sicherstellung ihrer korrekten Funktion zur Zeit der zeitaufwändigste Teil der Entwicklung ist. Formale Verifikation ermöglicht es, funktionale Eigenschaften solcher Systeme automatisch und vollständig zu überprüfen, ohne dass der Entwickler zeitraubende und fehleranfällige Tests für jeden Einzelfall schreiben muss. Die Eigenschaften werden in Instanzen des Erfüllbarkeitsproblems der Aussagenlogik (SAT, von engl. *satisfiability*) übersetzt, die dann mit Erfüllbarkeitsprüfer-Software (auch SAT-Solver genannt) gelöst werden können. Die derzeit besten SAT-Solver für industrielle Anwendungen basieren auf verbesserten Versionen des DPLL-Algorithmus. DPLL ist ein Suchalgorithmus, der von einer Vielzahl von Heuristiken gesteuert wird. Die zur Lösung eines SAT-Problems benötigte Zeit ist stark abhängig von der Wahl der Parameter dieser Heuristiken, und die optimale Einstellung der Parameter ist selbst ein schwieriges Problem. In der vorliegenden Arbeit wird ein neues, vollautomatisches Optimierungsverfahren für die Heuristikparameter von SAT-Solvern präsentiert und getestet. Das Verfahren basiert auf der Benutzung von Optimierungsalgorithmen, mit denen versucht wird, optimale Parameterkonfigurationen für Trainingsmengen von SAT-Problemen anzunähern. Ein Endergebnis wird dann aus allen gesammelten Daten berechnet. Als Optimierungsalgorithmen wurden zwei Unterarten von Evolutionären Algorithmen getestet: Genetische Algorithmen und Evolutionsstrategien. Der zu optimierende SAT-Solver war das bekannte Open Source Programm MINISAT. Es konnte gezeigt werden, dass die Parameterkonfigurationen, die mit dem Verfahren erzeugt wurden, ähnlich gut sind wie die von Experten ermittelten Standardparameter.

Abstract

Digital electronic systems are now so large and complex that ensuring their correct functionality has become the most time-consuming part of their design. Formal verification allows the exhaustive, automatic testing of functional properties of such systems without requiring the designer to create individual test cases manually, which is time-consuming as well as prone to errors and oversights. The properties are first transformed into instances of the Boolean satifiability problem (SAT), which are then solved with SAT solvers. The most efficient SAT solvers for industrial SAT problems are based on enhanced versions of the DPLL algorithm which employs a number of heuristics to guide the search for a solution. Solving times are highly dependent on the choice of the solver's heuristic parameters, and adjusting the heuristics optimally is a complex task in itself. This work presents and tests a new, fully automatic optimization procedure for a SAT solver's heuristic parameters that is based on using local search algorithms which attempt to find optimal parameters for training sets of SAT problems; a result configuration is synthesized from the gathered data. For the optimization two subtypes of Evolutionary Algorithms (local search algorithms that mimic Darwinian evolution), Genetic Algorithms and Evolution Strategies, were tested. The target of optimization was the well known open-source SAT solver MINISAT. It could be shown that the parameter configurations generated by the automatic procedure are competitive with the default parameters set by human experts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

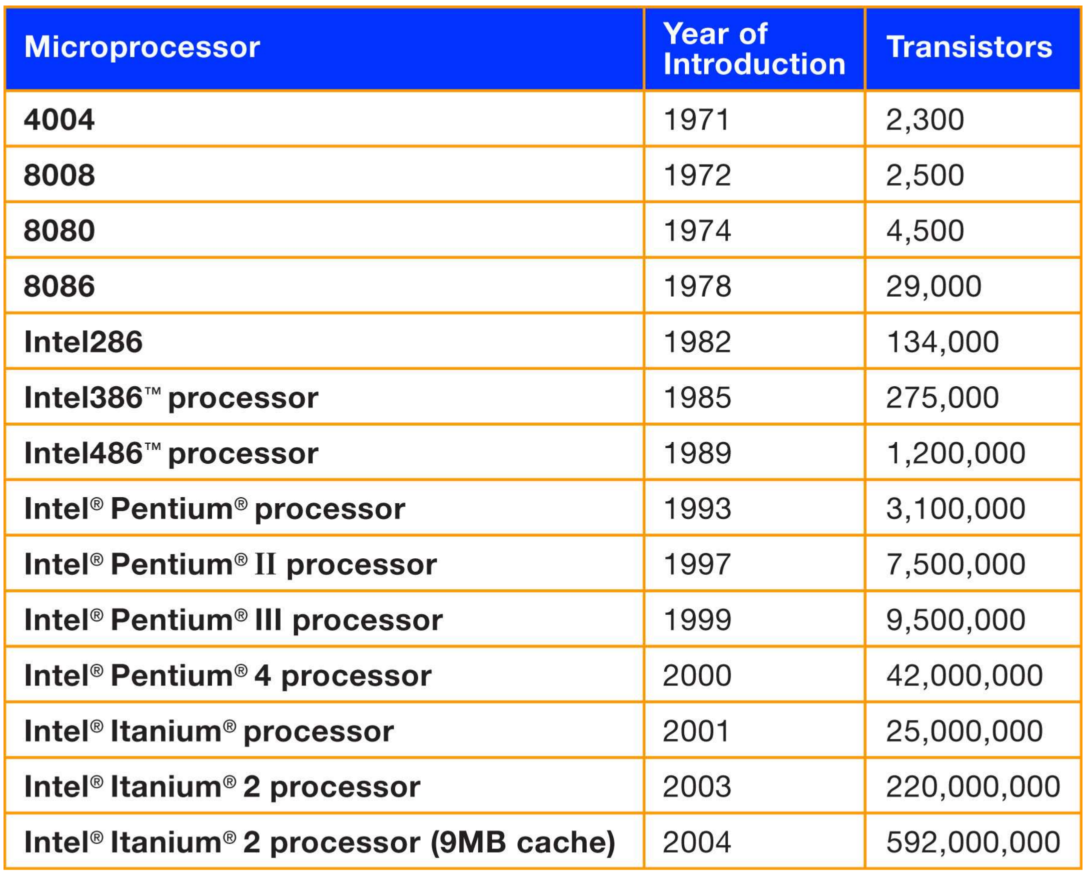
1.1 Integrated circuits

The first *integrated circuit* (IC or “chip”) was invented in 1958 by the American electrical engineer Jack Kilby while working at Texas Instruments. It was a simple oscillator circuit which consisted of a single transistor, some resistors and a capacitor on a small piece of germanium. This electronic circuit was “integrated” because instead of being made of many discrete components like resistors etc. wired together, every component was made in place from the same piece of semiconductor. Kilby wrote in a 1976 article [Kilby76]:

> “In my discouraged mood, I began to feel that the only thing a semiconductor house could make in a cost-effective way was a semiconductor. Further thought led me to the conclusion that semiconductors were all that were really required—that resistors and capacitors, in particular, could be made from the same material as the active devices. I also realized that, since all of the components could be made of a single material, they could also be made *in situ*, interconnected to form a complete circuit.”

Up to this point, large electronic circuits had to be soldered together out of thousands of components, which was a time-consuming and error-prone process. On top of that, the resulting circuits were often unreliable as well as large and heavy, precluding their use in many applications. Integrated circuits on the contrary are easy to mass-produce (and therefore cheaper), small, light, and comparatively reliable. Jack Kilby was one of the laureates for the Nobel Prize in physics in 2000 for his work on integrated circuits.

Since their inception the number of components that can be put on one integrated circuit has been steadily rising. The size of the structures on the chips became smaller and smaller, and have now reached deep-submicron (DSM) sizes (smaller than one millionth of a meter). Intel corporation’s co-founder Gordon Moore made a prediction in a 1965 magazine article [Moore65] that has become known as *Moore’s law*: the transistor density on integrated circuits doubles about every two years. This



Microprocessor	Year of Introduction	Transistors
4004	1971	2,300
8008	1972	2,500
8080	1974	4,500
8086	1978	29,000
Intel286	1982	134,000
Intel386™ processor	1985	275,000
Intel486™ processor	1989	1,200,000
Intel® Pentium® processor	1993	3,100,000
Intel® Pentium® II processor	1997	7,500,000
Intel® Pentium® III processor	1999	9,500,000
Intel® Pentium® 4 processor	2000	42,000,000
Intel® Itanium® processor	2001	25,000,000
Intel® Itanium® 2 processor	2003	220,000,000
Intel® Itanium® 2 processor (9MB cache)	2004	592,000,000

Figure 1.1: Moore's Law Transistor Count Chart, Copyright ©2005 Intel Corporation.

"law" has been accurate for a long time as can be seen in the transistor counts of Intel's own microprocessors (Fig. 1.1), and only now seems to be slowing down. The latest benchmark was passed in 2010 when Intel began production of a processor codenamed "Tukwila" which was the first to contain over 2 billion transistors.

Making use of this vast number of transistors requires increasingly more efficient design strategies and tools. Manually placing the millions of transistors on an IC (and managing their interconnections) is unfeasible, hence today's chips are designed using *Hardware Description Languages* (HDLs) like VHDL and Verilog, which describe the behavior of the circuit in text form. *Synthesis tools* then translate the HDL source code into a form from which eventually chips can be manufactured. *Simulator* software can be used to test designs without needing to build prototypes of the circuits, which would be prohibitively expensive and time-consuming. *Electronic Design Automation* (EDA) is the umbrella term for software tools that deal with designing ICs.

1.2 Verification

While the increase in transistor density has led to extremely large and powerful designs being possible in smaller and smaller spaces, it has also become much harder to detect design errors. Apart from the unavoidable possibility of human error, synthesis tools can not be guaranteed to always generate correct circuits from HDL sources. Making sure that a design works as intended is called *verification* [Bergeron00, pg. 1]:

> "Verification is a *process* used to demonstrate that the intent of a design is preserved in its implementation."

Verification happens *before* the design is manufactured, determining the correctness of a finished chip is called *testing* [Bergeron00, pg. 16]:

> "Testing is often confused with verification. The purpose of the former is to verify that the design was manufactured correctly. The purpose of the latter is to ensure that a design meets its functional intent."

IC designs are so complex now that the major portion of the entire design effort is in fact verification [Bergeron00, pg. 2]:

> "Today, in the era of multi-million gate ASICs, reusable intellectual property (IP), and system-on-a-chip (SoC) designs, verification consumes about 70% of the design effort. Design teams, properly staffed to address the verification challenge, include engineers dedicated to verification. The number of verification engineers can be up to twice the number of RTL designers."

One approach to verification is the use of simulation on *testbenches*. A testbench models the environment of the chip and provides inputs to it. When simulating the testbench together with the *Design Under Verification* (DUV), the outputs can be observed and compared to expected values. Writing testbenches is a complex problem in itself: the verification engineer has to ensure that every possible combination of inputs and every possible state of the DUV is reached and the correct outputs are generated. This is practically impossible even with smaller designs, since the number of input and state combinations reaches astronomical numbers very quickly.

An alternative to simulation is *formal verification*, which generally takes the form of *equivalence checking* or *model checking*. Equivalence checking deals with comparing circuits (for example "do the synthesized circuit and its HDL description implement the same Boolean function?") and model checking with properties of *state machines* (for example "will this counter count up correctly?"). In both cases a mathematical model of the DUV's property to be checked is generated, which can then be automatically proven to be correct or incorrect using *theorem prover* software (a program that can automatically prove mathematical theorems). If the property does not hold, the verification software can even provide a *counterexample* that shows a situation where the property fails. The advantage of formal verification over simulation is that formal verification is inherently *exhaustive*: it checks every possible situation at once without the verification engineer having to create the test input sequences (and a testbench) first. Unfortunately formal verification can not be used for designs of arbitrarily large size, therefore it does not replace simulation but rather complements it to increase the chance of finding errors.

Both equivalence checking and model checking first generate an "intermediary problem" out of the actual design property to be verified (details on how this works will be presented in later chapters). The form of the intermediary problem for both types of formal verification is the *Boolean satisfiability problem*.

1.3 The Boolean satisfiability problem

The *Boolean satisfiability problem* (SAT) is a fundamental problem of computer science with many important practical applications. The problem is finding *truth values* (*true* or *false*) for the variables of a Boolean formula so that the formula evaluates to *true*. This work uses the notation "+" for Boolean OR, juxtaposition of expressions to signify AND, and a bar over expressions for negation. The following Boolean formula is *satisfiable* because assigning $\{v_1 = 1, v_2 = 1, v_3 = 0, v_4 = 1, v_5 = 1\}$ makes it evaluate to 1 (*true*):

$$(v_1 + v_2)(v_2 + \bar{v_3} + v_4)(v_5) \tag{1.1}$$

In contrast, the next Boolean formula is *unsatisfiable* because there exists no combination of values can be assigned to the variables v_1 and v_2 so that the result is 1:

$$(v_1 + v_2)(v_1)(\bar{v_1}) \tag{1.2}$$

While SAT is easy to solve for a small number of variables simply by trying out all possible combinations, this quickly becomes unfeasible when the number of variables rises. For a SAT problem with 1000 variables (which is commonplace in industrial applications), the number of value combinations is $2^{1000} \approx 10^{301}$ which exceeds the total number of atoms in the observable universe (ca. 10^{80}) by far. Even the fastest computers today would take millenia to try out all possible combinations, until one is found that *satisfies* the formula or all combinations have been tried unsuccessfully and the formula is found to be unsatisfiable. There is currently no known way to solve SAT that does not require an exponentially rising amount of resources (time and computer memory) depending on the number of variables, so that the more complex instances of the problem are essentially unsolvable. SAT shares this property with a whole class of other, hard to solve problems: the *complexity class NP*.

The complexity class NP is a set of problems for which it is easy to check if a given solution is correct, but that are (as far as we know) computationally hard to solve. "Easy to check" means that for any given solution of a problem from the set NP it takes an amount of time that is computed only by a *polynomial* of the number of its variables to confirm if that solution is indeed correct. On the other hand, so far it appears that solving a problem from NP always requires a time that in the worst case rises exponentially depending on the number of variables. This means, in essence, that no algorithm can exist that solves every possible problem in NP in practically feasible time, unless it is true that the "easier" *complexity class P* is equal to NP. The complexity class P is the set of problems which can both be solved and its solutions confirmed in a time that rises as a polynomial of the number of variables of the problem. The question if P=NP, known as the *P versus NP Problem*, is one of the most important open questions of computer science. P versus NP is one of the "Millenium Prize problems": the *Clay Mathematics Institute of Cambridge, Massachusetts* (CMI) has posted a reward of 1 million US$ for solving any of 7 (of which 6 now remain) "important classic questions that have resisted solution for many years". Their website[1] states about P versus NP:

> "In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971."

SAT was the first problem that was proven to be *NP-complete* [Cook71]: an NP-complete problem is in the set NP, and every other problem in NP can be translated into it efficiently (meaning in a time that rises as a polynomial of the number of variables). This means that any other problem in NP can be translated into an equivalent SAT problem.

There is great interest in building efficient *SAT solvers* (programs that solve SAT) because a large number of problems of industrial or scientific interest are in NP, or are not in NP but can still be translated into SAT (for a list of applications of SAT itself see for example [Marques-Silva08]). One of the most well-known NP-complete problems other than SAT is the *Travelling Salesman Problem* (TSP). Given a list of cities and the traveling cost between every pair of cities, TSP asks what the cheapest possible tour is that visits all cities exactly once and returns to the starting point. This basic problem has many different applications in a wide variety of fields, including the obvious use in transportation but also, for example, for controlling machines for drilling holes (where the holes and the distances between them take the place of the

[1] http://www.claymath.org/millennium/P_vs_NP/, accessed September 2010

cities), and modified versions have even more. Various EDA problems like equivalence checking of Boolean circuits (determining if two combinational logic circuits have the same behavior) and *Bounded Model Checking* (BMC) (a method for checking temporal properties of a sequential logic circuit) are also NP-complete. Instead of building a specific solver for each of these problems it would be more convenient, given a powerful SAT solver, to translate them into SAT first and then use the SAT solver on the translated problem. The result returned by the SAT solver is then translated back into a solution of the problem. A SAT solver therefore can be seen as a generic "problem-solving engine" (Fig. 1.2).

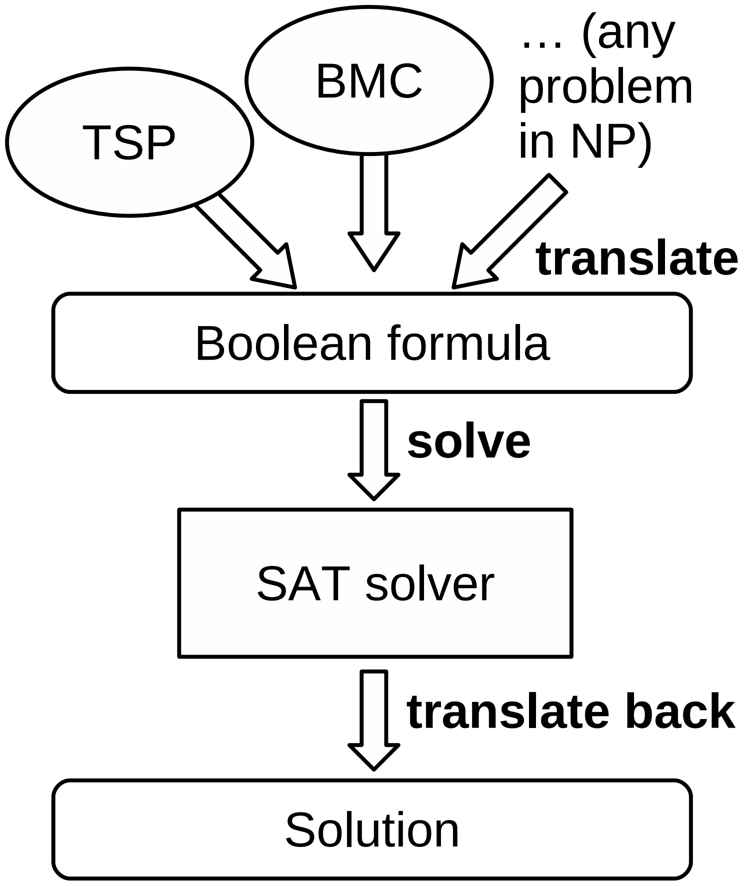


Figure 1.2: SAT solvers can be used as generic problem solvers

1.4 The DPLL algorithm

The most successful SAT solvers in use today are almost all based on the *DPLL algorithm* framework, which is named after the researchers Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland who worked on DPLL [Davis62] and/or its predecessor, the DP algorithm [Davis60]. The DPLL algorithm solves SAT by performing a search of all possible combinations of the Boolean formula's variable values until either a satisiying combination is found or the formula is found to be unsatisfiable after all combinations are exhausted. The algorithm avoids many obviously unsatisfying combinations by determining "implications" (logical inferences) of previously assigned values on other variables during the search. It is therefore more efficient than the naive solution of computing the result of every possible combination.

Although the original DPLL algorithm was already described in 1962, the performance of DPLL SAT solvers in industrial applications improved particularly after the invention of the CHAFF solver [Moskewicz01] in 2001, which combined various optimized implementations of the basic operations of the algorithm with improved *heuristics*. A heuristic is an inexact method for solving, or aiding in solving a problem [Pearl84, pg. 3]:

> "Heuristics are criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal. They represent compromises between two requirements: the need to make such criteria simple and, at the same time, the desire to see them discriminate correctly between good and bad choices.
>
> A heuristic may be a *rule of thumb* that is used to guide one's actions. It is the nature of good heuristics both that they provide a simple means of indicating which among several courses of action is to be preferred, and that they are not necessarily guaranteed to identify the most effective course of action, but do so sufficiently often."

Due to the NP-completeness of SAT there can be no exact algorithm that computes ideal results for the various decisions that must be made inside DPLL in feasible time (unless P=NP), therefore all DPLL solvers must use heuristics. Usually a solver implementation has a number of settings and parameters for the heuristics that influence the performance to a large degree. The heuristic parameters are usually numbers (real-valued or integers) for which sensible values have to be found. Unfortunately the connection between the parameter values and the time the solver requires to solve a problem is usually not obvious (in the literature sometimes these values are actually referred to as "magic numbers"). The preset values that are used in publicly available solvers are based on experimentation and the implementor's experience.

When a new solver is implemented, it can be tested and optimized using a large number of SAT problems from a variety of fields that has been made available on the Internet for scientific purposes. For example, the benchmark problems for the annual *SAT competitions*[2] that are held concurrently with the yearly international SAT conference are also available for download. In the SAT competitions many SAT solvers run against each other by solving a portfolio of SAT problems in different categories. There is also a *time-out period* which is the maximum allowed time a solver can use to solve a problem before it is stopped and the solving attempt is marked as failed. The solvers are ranked higher the more problems they solve before timeout, and the quicker they solve them.

Similar in procedure to the SAT competition, optimizing a SAT solver can be performed manually by systematically testing a number of configurations for its heuristic parameters on a *training set* of SAT problems. Configurations which result in a shorter, total solving time would be seen as "better", and the settings can be refined by varying the parameters slightly and re-testing until the end result is satisfactory. This is obviously a time-consuming process, since solving even a single SAT problem may take hours, depending on the difficulty of the problem, and solving the entire training set takes accordingly longer. There is also the question which parameter values should be tried out, and how to vary them after each testing iteration. The brute force approach consists of testing every single one of all possible configurations. If there is only a single parameter to optimize and it is decided to test 10 different values for that parameter, then the entire set of training problems has to be solved 10 times over to rank the 10 configurations by their quality. When two parameters have to be optimized, with 10 possible values each, there are already $10 \times 10 = 10^2 = 100$ different configurations to test. Using the brute force approach, the number of possible configurations, and with it the number of necessary SAT solver evaluations, rises exponentially with the number of parameters and very quickly becomes unfeasible to compute. This general problem which always occurs when exhaustively searching in higher-dimensional spaces is also known as the *curse of dimensionality*. In the brute force approach a lot of time is also wasted on testing configurations which are very likely not optimal. For example, if the configuration $(0.5; 0.3; 1.9)$ for 3 heuristic parameters was found to have very low performance, a human optimizer would likely not waste time testing the very similar configuration $(0.4; 0.3; 1.9)$ which differs only very slightly in the first parameter. Instead, he would wish to avoid wasting time on unpromising configurations and try a variation of a more promising configuration. In any case, optimizing the SAT solver manually is a tedious, time-consuming process. An automated or at least semi-automatic approach would be much preferable that chooses the heuristic parameters itself and steers the search in promising directions depending on previously gained knowledge. Such applications are the domain of *optimization algorithms*.

[2]Website at http://www.satcompetition.org

1.5 Optimization

The problem of finding good settings for the parameters of a SAT solver algorithm is a type of *optimization problem*. An optimization problem consists of finding the best possible solution out of a set of all possible solutions [Gill81, pg. 1]:

> "An optimization problem begins with a set of independent variables or parameters, and often includes conditions or restrictions that define acceptable values of the variables. Such restrictions are termed *constraints* of the problem. The other essential component of an optimization problem is a single measure of "goodness", termed the *objective function*, which depends in some way on the variables. The solution of an optimization problem is a set of allowed values of the variables for which the objective function assumes an "optimal" value. In mathematical terms, optimization usually involves *maximizing* or *minimizing*; for example, we may wish to maximize profit or minimize weight."

The term *fitness* or *fitness function* is sometimes used to mean "objective function", especially in the context of Evolutionary Algorithms. The *search space* (the set of all possible solutions) of optimization problems can be visualized as a *fitness landscape*. The fitness landscape (the term originates from genetics [Wright32]) is like a map of the search space, where each X-Y-coordinate represents one possible solution and its respective objective function value is represented by the height (Z-coordinate) of a point at those coordinates. The sum of all the points forms a "landscape" of valleys, planes and peaks. Fitness landscapes can contain a large number of valleys and peaks, where the highest peak or lowest valley (depending on if the problem is one of minimization or maximization) is the *global optimum*, and all smaller peaks or valleys are *local optima*. For experimental purposes, researchers often use test functions whose fitness landscapes have known properties, for example Fig. 1.3 shows a part of the fitness landscape of the *six-hump camel back function* $f(x, y) = (4 - 2.1x^2 + x^{4/3})x^2 + xy + (-4 + 4y^2)y^2$ [Dixon78].

An optimization algorithm then can be imagined as an explorer with a limited range of vision wandering in the fitness landscape, seeking the highest peak or the deepest valley. In practice the objective of the problem might not be identifying the global optimum but rather finding any peak above a certain level, or finding a set of peaks of equivalent fitness, or simply finding any highest point in the allotted time. Bäck writes on the complexity of optimization [Bäck96, pg. 56] (referencing [Murty87]):

> "The result implies that any global optimization problem that goes beyond a very low complexity is an NP-complete problem."

Optimization problems can be classified by the properties of the objective function and the constraint functions. Which optimization algorithm is most efficient (or even

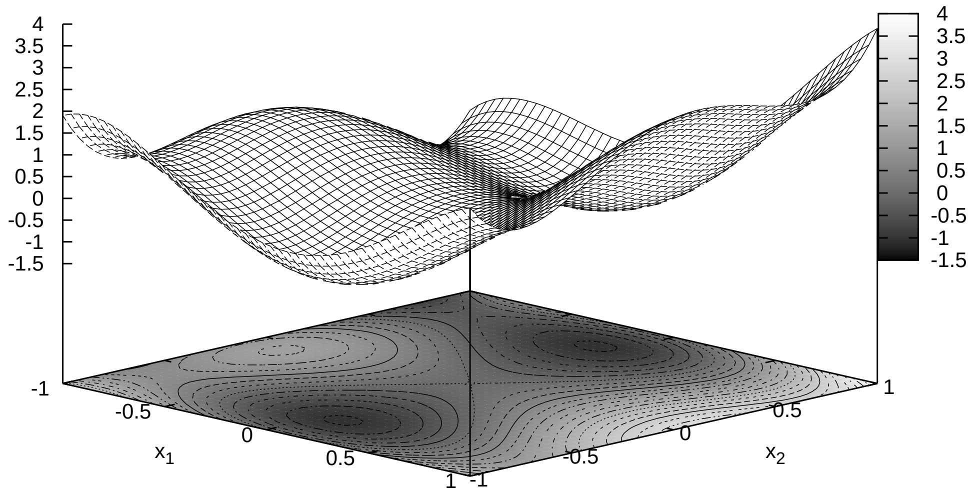


Figure 1.3: Fitness landscape of the six-hump camel back function

applicable at all) depends on the properties of the problem. For example, the objective function may have a single variable (*univariate* function) or many variables (*multivariate* function). Another classification scheme considers how much information can be extracted from the objective function during the course of an optimization algorithm: if the objective function is differentiable, first-order derivatives can be used to guide the process (this is generally only possible if the objective function is given in some algebraic form). Some optimization algorithms can also make use of second-order derivatives, if they are available. In the metaphor of the optimization algorithm as a wandering explorer this would mean that the explorer can measure the steepness of the ground at his current position and choose to walk in the direction that slopes upwards most strongly (where presumably some peak/optimum lies). If the objective function provides no such information, the optimization algorithm can only use comparisons of the objective function values visited so far to guide itself.

In the SAT solver optimization problem the variables are the heuristic parameters, and the objective function is the accumulated solving time for the training set of SAT problems. The constraints are the ranges for the values of the parameters (for example $[0,1]$). Differential information is not available because the objective function is not an algebraic formula but rather a program.

When the objective function is not differentiable, *function comparison methods* (sometimes also called *direct search methods*) [Gill81] must be used that rely only on the objective function values themselves. Such methods, while usually easy to implement, can never be guaranteed to converge and usually become very slow when having to deal with higher dimensions. Many direct search methods were developed for specific problems and are not useful in the general case, and in addition need optimal settings of their own parameters to be successful. *Evolutionary Algorithms* (EAs) are a class of optimization algorithm which are based on simulations of natural processes like evolution. They are direct search methods which have shown surprising success in a large variety of fields and applications. EAs are a sub-field of the larger field of *Soft Computing*.

1.6 Soft Computing

Soft computing (SC) stands for a set of problem solving techniques which allow and exploit a degree of imprecision to deal with hard problems efficiently. The term is relatively new and was coined around the early 1990s. The American mathematician Lotfi A. Zadeh, inventor of *fuzzy logic*, states in [Zadeh94]:

> "In traditional—hard—computing, the prime desiderata are precision, certainty, and rigor. By contrast, the point of departure in soft computing is the thesis that precision and certainty carry a cost and that computation, reasoning, and decision making should exploit—wherever possible—the tolerance for imprecision and uncertainty."

SC is now an established field with dedicated conferences and scientific journals from large publishers, for example *Soft Computing* by Springer Verlag and *Applied Soft Computing—The Official Journal of the World Federation on Soft Computing (WFSC)* by Elsevier, which defines SC as such in its mission statement[3]:

> "Soft computing is a collection of methodologies, which aim to exploit tolerance for imprecision, uncertainty and partial truth to achieve tractability, robustness and low solution cost."

The *Berkeley Initiative in Soft Computing* (BISC) group at the University of California, Berkeley which is headed by L. Zadeh, states in its current mission description[4]:

> "The principal constituents of soft computing (SC) are fuzzy logic (FL), neural network theory (NN) and probabilistic reasoning (PR), with the latter subsuming belief networks, evolutionary computing including DNA computing, chaos theory and parts of learning theory."

The various sub-fields of SC complement each other rather than being in competition, for example FL can be used to control NN parameters [Bonissone97]. This work deals with the application of algorithms from the *Evolutionary Computation* (EC) sub-field of SC.

1.7 Evolutionary Computation

Optimization problems can be tackled with *search algorithms* [Hoos04, pg. 23]:

> "Basically all computational approaches for solving hard combinatorial problems can be characterized as search algorithms. The fundamental idea behind the search approach is to iteratively generate and evaluate candidate solutions; in the case of combinatorial decision problems, evaluating a candidate solution means to decide whether it is an actual solution, while in the case of an optimisation problem, it typically involves determining the respective value of the objective function."

Search algorithms can be further classified into *systematic search algorithms* and *local search algorithms* [Hoos04, pg. 24]:

> "*Systematic search algorithms* traverse the search space of a problem instance in a systematic manner which guarantees that eventually either a solution is found, or, if no solution exists, this fact is determined with

[3]http://www.elsevier.com/wps/find/journaldescription.cws_home/621920/description, accessed September 2010

[4] http://www-bisc.cs.berkeley.edu/BISCProgram/default.htm, accessed September 2010

> certainty. This typical property of algorithms based on systematic search is called *completeness*. *Local search algorithms*, on the other hand, start at some location of the given search space and subsequently move from the present location to a neighbouring location in the search space, where each location has only a relatively small number of neighbours and each of the moves is determined by a decision based on local knowledge only. Typically, local search algorithms are *incomplete*, i.e., there is no guarantee that an existing solution is eventually found, and the fact that no solution exists can never be determined with certainty. Furthermore, local search methods can visit the same location within the search space more than once. In fact, many local search algorithms are prone to getting stuck in some part of the search space which they cannot escape from without special mechanisms like a complete restart of the search process or some other sort of diversification steps."

Local search algorithms that make use of randomness are called *stochastic local search (SLS) algorithms* [Hoos04, pg. 30]:

> "Many widely known and high performance local search algorithms make use of randomised choices in generating or selecting candidate solutions for a given combinatorial problem instance. These algorithms are called *stochastic local search (or SLS) algorithms*, and they constitute one of the most successful and widely used approaches for solving hard combinatorial problems. SLS algorithms have been used for many years in the context of combinatorial optimisation problems. Among the most prominent algorithms of this kind we find the Lin-Kernighan algorithm [Lin and Kernighan, 1973] for the Traveling Salesperson Problem, as well as general methods like Evolutionary Algorithms [Bäck, 1996], and Simulated Annealing [Kirkpatrick et al., 1983]."

In any search, there are two basic objectives: *exploration* and *exploitation* [Michalewicz94] [Coley98]. Exploration means sampling many different areas of the search space, which is important for finding the global optimum, and exploitation means examining a narrow area of the search space, which helps to pinpoint a local optimum with high precision. Ideally a search algorithm finds a good compromise between exploration and exploitation, since concentrating too much on just one can lead to either missing the global optimum (too much exploitation) or being unable to locate any optimum precisely (too much exploration). The *Evolutionary Computation* (EC) sub-field of SC encompasses a number of SLS algorithms that share a population-based approach [Bonissone97]. Some authors also refer to EC as *Evolutionary Computing*, but the term Evolutionary Computation seems to be the prevalent one, for example MIT Press publishes a journal of this name. EC algorithms "simultaneously" explore and exploit the search space by searching centered around a several starting locations (the *population*) rather than just one.

Various problem-solving algorithms inspired by evolution were tested in the mid-1950s already [De Jong97], but the three main forms still in use today had their start in the mid-1960s: *Evolution Strategies* (ES) (Ingo Rechenberg, 1965), *Evolutionary Programming* (EP) (Lawrence J. Fogel, *et al* 1966) and *Genetic Algorithms* (GA) (John Holland, 1967). These three methods are the main representatives of what is known as *Evolutionary Algorithms* (EA) [Bäck96], which developed independently of each other for a long time until an effort was made in the early 1990s to bring together the various approaches to the same paradigm of population-based search under the term *Evolutionary Computation*. All these methods simulate a form of *evolution* (hence the name "evolutionary"), though they differ in the details.

An EA starts with a set of randomly generated candidate solutions (usually called the *population*) encoded in some particular format (Fig. 1.4). The first such population is the first *generation* of the EA. The population of each following generation is generated by choosing and modifying solutions from the population of the previous generation. The members of the population are often called *individuals*, *genotypes* or *chromosomes* (much of the EA terminology is borrowed from biology). In biology, the *genotype* of an organism is its entire set of "blueprint" information stored as DNA, while the *phenotype* is the resulting organism itself. In EAs the phenotype means the (translated) actual solution while the genotype refers to its encoding. The encoding depends on the type of the EA. The solutions are akin to organisms living in an environment, and the encoding of each solution is analogous to the DNA of that organism.

Each solution is ranked according to how well it can solve the problem. The ranking position is determined by the value of a *fitness* measure, which is usually a single, real-valued number that is higher the better the quality of the solution is. This quality measure is basically equivalent to the objective function of the optimization problem, but in EAs a higher fitness is always assumed to be better. Therefore in the case of a minimization problem there must always be some transformation (*fitness scaling*) step that translates low ("good") objective values into corresponding high (also "good") fitness values. Even for maximization problems it may be necessary to apply some form of transformation, since for example not all EAs can deal with negative fitness values.

Once all individuals are assigned a fitness value, the *selection* operator chooses which solutions are allowed to live and which die out. Selection preferentially chooses individuals with a high fitness to live on, for example it can always choose the 5 individuals with the highest fitnesses out of a population of 10: the 5 weaker individuals would always die off then. This is an example of *deterministic* selection. The selection operator can alternatively be *stochastic*, so that there is some randomness to which individuals are chosen as parents (this gives individuals with lower fitness which would always be culled by a deterministic operator a chance to live). Depending on the EA, there are a variety of selection operator implementations that can be used. The individuals chosen by the selection operator are called *parents*.

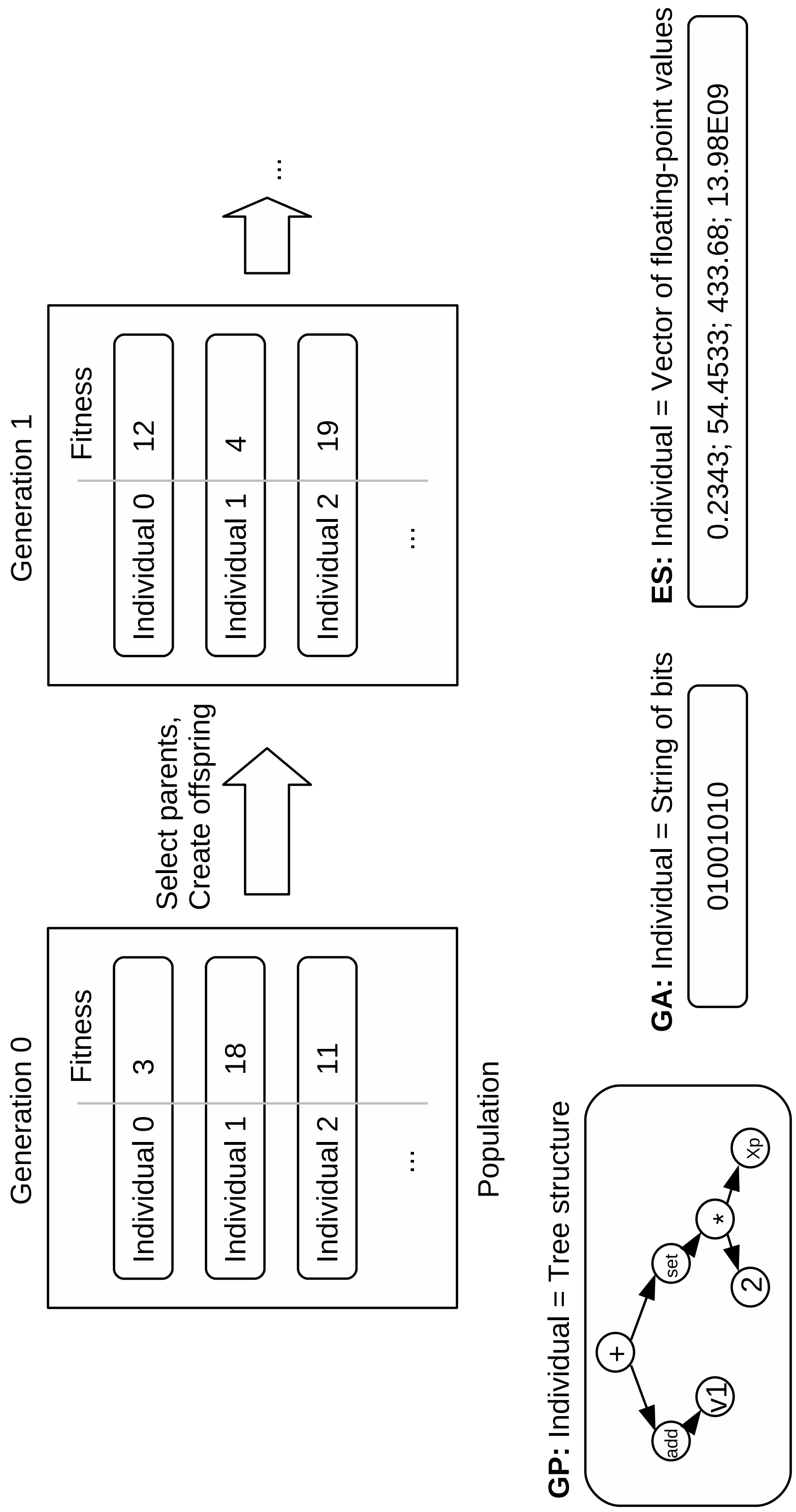


Figure 1.4: The basic operation principle of Evolutionary Algorithms

Parents are either copied verbatim into the next generation, or several parents can be *recombined* into one *offspring* which is placed in the next generation, or a parent can be randomly modified (*mutated*) before being copied. In any case, all non-parent individuals die out (they do not influence the next generation at all). *Recombination operators* combine the genotypes of two or more individuals to form new solutions (offspring); this is an analogy to *chromosomal crossover* in biology. A *mutation operator* creates some random change in the genes of a selected solution similar to small random defects caused in DNA by radiation or chemicals in nature. It is also possible to recombine parents, then mutate the resulting offspring. Recombination, mutation and other operators which are responsible for creating variation and diversity in the population are called *variation operators* [Eiben08]. Not all EAs use both recombination and mutation, but all need some form of selection that drives the evolution. The effects of the selection operator cause *natural selection*, because helpful heritable traits (those which result in higher fitness) become more common in the population with each successive generation. If the parameters of the EA are chosen appropriately (for example the population is large enough) then over the course of time the solutions should become better and better (similar to *adaptation* of natural organisms).

All EAs share a population-based approach, but differ greatly in the implementation details, for example the format of the genotype and the operators used (Fig. 1.4). Today the main sub-fields of EAs are [Bonissone97]:

1. Evolution Strategy (ES): the genotypes are real-valued vectors.
2. Evolution Program (EP): originally used for evolving finite state machines.
3. Genetic Algorithm (GA): the genotypes are binary strings.
4. Genetic Programming (GP) [Koza92]: the genotypes are computer programs represented as trees.

EAs have been used to find efficient and often non-intuitive solutions to complex problems in a wide variety of fields, for example building a multiplier circuit with as few logic gates as possible [Vassilev00], but also in the arts [Todd94] and medicine [Pena-Reyes00].

1.8 Thesis Contributions

As has been stated earlier, tuning the heuristics of DPLL SAT solvers is a complex and time-consuming process. It would greatly reduce the demand on expensive programmer's time if a reliable automatic process for optimizing DPLL heuristics existed. This would speed up the development of new SAT solvers, but another use of such an automatic optimizer would be to allow users of the solver to adapt the heuristics to the particular class of SAT problems they work with most often, even if they have little expertise or even no knowledge at all of the underlying algorithms. For example, hardware verification engineers could "train" the SAT solver that powers their verification tools to solve the SAT problems generated from their designs faster. The optimizations could be performed in the idle time of the workstations while the designers continue working on less computationally expensive tasks; alternatively, since computer clusters and distributed services like cloud computing are becoming more readily available as sources of cheap computing power, the highly parallelizable EA-based optimization could be shifted to other machines partially or entirely. This work investigated the use of EAs as fully automatic optimizers for DPLL heuristics. In the following some prior work on this subject will be reviewed.

1.8.1 Prior work

It was found that different values of heuristic parameters lead to substantially different behaviors of the solver, to such an extent that the same solver with different settings can appear like a totally different solver [Audemard08]. Despite the importance of using "good" values for these parameters, there is very little literature available on how to determine them. Authors of new solvers seem to rely on manual testing and optimization, and do not publish the procedures with which they arrive at the default values used in the solvers. One of the few works in the literature that investigated an automatic approach to SAT solver heuristic optimization is [Hutter07], which is the most similar in approach to this work. It is stated there:

> "During the typical development process of a heuristic solver, certain heuristic choices and parameter settings are tested incrementally, typically using a modest collection of benchmark instances that are of particular interest to the developer. Many choices and parameter settings thus made are "locked in" during early stages of the process, and typically, only few parameters are exposed to the users of the finished solver. In many cases, these users never change the default settings of the exposed parameters or manually tune them in a manner similar to that used earlier by the developer. Not surprisingly, this manual conguration and tuning approach typically fails to realize the full performance potential of a heuristic solver. ... There are almost no publications on automated parameter optimization for decision procedures for formal verication. ... The only other work we are aware of is unpublished, ad hoc work in industry."

At the time of writing this state of affairs had seemingly not changed. It is further stated in [Hutter07] about the manual heuristics optimization process:

> "After the first version of SPEAR was written and its correctness thoroughly tested, its developer, Domagoj Babič, spent one week on manual performance optimization, which involved: (i) optimization of the implementation, resulting in a speedup by roughly a constant factor, with no effects on the search parameters, and (ii) manual optimization of roughly twenty search parameters, most of which were hard-coded and scattered around the code at the time. The manual parameter optimization was a slow and tedious process done in the following manner: the SPEAR developer collected several medium-sized benchmark instances which it could solve in at most 1000 seconds and attempted to come up with a parameter conguration that would result in a minimum total runtime on this set. The benchmark set was very limited and included several medium-sized BMC and some small software verication (SWV) instances generated by the CALYSTO static checker. Such a small set of test instances facilitates fast development cycles and experimentation, but has many disadvantages. Quickly it became clear that implementation optimization gave more consistent speedups than parameter optimization. Even on such a small set of benchmarks, the variations due to different parameter settings were huge. . . . Given the costly and tedious nature of the process, no further manual parameter optimization was performed after finding a configuration that seemed to work well on the chosen test set. . . . For most decision procedures, the process of finding default (or hard-coded) parameter settings resembles the manual tuning described above. Furthermore, most users of these tools do not change these settings, and when they do, they typically apply the same manual approach."

In [Hutter07] 26 heuristic parameters of the SAT solver SPEAR were optimized using a stochastic local search based algorithm named PARAMILS; that algorithm used a simple hill-climbing process to search for optima, where in each step only a single parameter of a single configuration was modified and the changes were discarded if the new configuration did not improve solving times on the training set. This process is analogous to a human designer's manual tuning technique, as they state. To get out of local optima, PARAMILS cyclically enters a perturbation phase where all parameters are randomly modified at once, followed by another hill-climbing phase; afterwards the starting point of the next cycle is the better of the last two local optima found. The training sets used in that work were hundreds of SAT problems in size, and their experiments were performed on a cluster of 55 computers. Automatic optimization led to major improvements, which were especially large when performed on homogenous classes of similar problems. In contrast to the approach in [Hutter07] where the optimization algorithm is partially modeled on a "human" technique, this work uses EAs which operate on a whole population of configurations and modify all of the parameters at once in a semblance of natural evolution. Also in contrast, EAs do not usually destroy worse configurations outright but rather keep them around in the population.

While there have been attempts to use EAs in conjunction with SAT solving, these works usually deal with local-search based SAT solvers, not complete DPLL-based ones (for example [Fukunaga04]). The author has previously published two works that deal with optimizing DPLL SAT solver heuristics with EAs. In [Kibria06] it was attempted to find an optimal “initialization” for the decision heuristic in MINISAT using Genetic Programming. The GP algorithm generated small programs which computed initial values for the variable activities in MINISAT, which are normally all 0 at the beginning. Since the generated configurations were not substantially better than the standard initialization and also only were tested on a small set of 13 SAT problems, the results are unfortunately questionable. It could also not be made clear if the initialization was not simply a random factor, which makes it harder to determine the success of the approach. In [Kibria07] the author attempted to use a neural net to control the heuristics of MINISAT. The outputs of the neural net influenced variable decisions, restarts, and learned clause reduction; the inputs were various statistics of the search like the current number of learned clauses and so on. The multilayer feedforward neural network contained a large number of bias values and edge weights (318 total), which were evolved with an ES. In the experiments some amount of improvement of solving times for the training problems could be observed, but the resulting solver was weaker than standard MINISAT. Additionally the neural net computations took between 5% and 30% of the total run time, so that such a solver is not useful in a production setting.

1.8.2 Objective of this work

The objective of this work is to design and test a fully automatic optimization procedure for the heuristic parameters of DPLL SAT solvers. The procedure assumes that one SAT solver with a fixed number of (real-valued) heuristic parameters is being optimized.

1.8.2.1 The optimization procedure

The underlying principle behind the optimization procedure presented here is to use optimization algorithms to find heuristic parameters with which small *training sets* of SAT problems are solved as well as possible. The features that the “good” parameter configurations have in common are extracted by some automatic means, yielding a number of candidates for the final result. A large test set of SAT problems is solved using these candidates and the best, as determined by some scoring scheme, is then chosen as the final result. The whole procedure should only require some basic human intervention at the start to select the problem sets and determine the optimization algorithm settings, then run automatically until the final result is generated. The core of the procedure is an algorithm (or several) that searches for optimal heuristic parameters for some fixed training set of SAT problems; in this work the focus was on using population-based EAs for this purpose, namely the GA and ES. Fig. 1.5 shows an overview of the entire procedure. The steps of the proposed procedure are, in sequence:

1. PROBLEM POOL SELECTION: Choose a set of SAT problems that will be used for *training* (to use during the optimization) and one for *testing* (for testing the results after the optimization). The former set is the *training pool* $\mathcal{P}_{\text{train}}$, the latter the *test pool* $\mathcal{P}_{\text{test}}$. The sets can be, but need not necessarily be disjoint.

2. TRAINING SET SELECTION: Choose the *training sets* $\mathcal{T}_i = \{p_{i,1}, p_{i,2}, \ldots\}, i = 1 \ldots N_{\text{train}}$ where each $p_{i,j} \in \mathcal{P}_{\text{train}}$. Different training sets may contain the same problems.

3. OPTIMIZATION ON TRAINING SETS: Prepare *optimization algorithm instances* $\mathcal{A}_i = (\mathcal{M}_i, \mathcal{E}_i, \mathcal{C}_i), i = 1 \ldots N_{\text{train}}$. Each tuple $\mathcal{A}_i$ stands for some algorithm that generates a set of *solver configurations* $\mathcal{C}_i = \{C_{i,1}, C_{i,2}, \ldots\}$; each $C_j \in \mathcal{C}_i$ is a complete set of heuristic parameters for the SAT solver (in EA terms, each C_j is a single individual that needs to be evaluated). The set $\mathcal{M}_i$ completely defines the machine that $\mathcal{A}_i$ is executed on, including the timeout period T_i used ($T_i \in \mathcal{M}_i$). The internal parameters that control the optimization algorithm itself (like the maximum number of generations for an EA) are defined in the set $\mathcal{E}_i$. One algorithm instance $\mathcal{A}_i$ is synonymous with a single "run" of the optimization.

   Apply all algorithms $\mathcal{A}_i = (\mathcal{M}_i, \mathcal{E}_i, \mathcal{C}_i)$ on the respective training set $\mathcal{T}_i = \{p_{i,1}, p_{i,2}, \ldots\}$. Each pair of configuration and problem in $\mathcal{C}_i \times \mathcal{T}_i = \{(C_{i,j}, p_{i,k}) | C_{i,j} \in \mathcal{C}_i \text{ and } p_{i,k} \in \mathcal{T}_i\}$ has to be evaluated. The function $\text{solve}(C_{i,j}, p_{i,k}, \mathcal{M}_i)$ yields a *solving result* $R_{i,j,k}$ for solving problem $p_{i,k}$ on the machine defined by $\mathcal{M}_i$ using the timeout $T_i \in \mathcal{M}_i$ and the solver heuristic parameter configuration $C_{i,j}$. $R_{i,j,k}$ is a set of statistical values stemming from the solving attempt; at the very least it includes the solving time $t_{i,j,k} \in R_{i,j,k}$ (when a timeout occurred, this is indicated by a special value of $t_{i,j,k}$, for example $t_{i,j,k} = 2T_i$). Each configuration $C_{i,j}$ is associated with a real number called its *objective value* $V_{i,j}$ which rates its ability to solve training set $\mathcal{T}_i$ (lower $V_{i,j}$ are better); $V_{i,j}$ can be computed using all the statistics in all the relevant $R_{i,j,k}$, but at the simplest it is $V_{i,j} = \sum_{r=1} t_{i,j,r}$ (the sum of the solving times for all problems in $\mathcal{T}_i$ using $C_{i,j}$). All computed *configuration trials* $L_{i,j} = (\mathcal{M}_i, \mathcal{T}_i, C_{i,j}, V_{i,j})$ are entered into the *trial database* $\mathcal{D}$[5].

[5]A variant of the procedure (which has not been tested in this work) would store the results of every single SAT problem evaluation in the trial database instead of just the objective value of the

4. RESULT CANDIDATE SELECTION: Once the trial database $\mathcal{D}$ is completed, candidates for the final result can be generated from the collection of configurations rated by objective value; what is sought is a configuration that solves as many of the training problems as possible, as fast as possible. There are many ways how this can be accomplished: the simplest possibility would be to make all the best configurations found for each training set a candidate. It is also possible to try combining several solutions, for example by averaging some parameter over many configurations (this is why the training and test pools can contain the same problems: none of the candidate configurations may be the exact same as those generated during training). Each configuration chosen (or newly generated from several configurations) from $\mathcal{D}$ as a candidate is added to the *candidate set* $\mathcal{K}$.

5. RESULT CANDIDATE TESTING: In the final step, each configuration C from the candidate set $\mathcal{K}$ is used to solve all problems in the test pool $\mathcal{P}_{\text{test}}$. A *scoring scheme* rates the configurations by assigning them a number that is higher the more problems they could solve, the faster they could solve them compared to the rest of the candidates etc. The final result is the configuration that has been scored the highest (ties need to be broken in some way).

Many parts of the procedure are parallelizable, but the most important part is the evaluation of the configurations by running the SAT Solver. The optimization algorithms that generate the configurations are typically relatively fast and a modern machine could likely handle all of them at once, if the actual SAT solver runs are distributed on a computer cluster.

Fig. 1.6 shows the general concept of how the EAs were used to optimize the SAT solver on a training set of problems. The EA population contains different solver configurations encoded in the appropriate format. The SAT solver is initialized with each configuration in turn and then used to solve the training set. The solving times are recorded and an objective value is assigned to the individuals (meaning the configurations) afterwards. The EAs are then iterated for a number of generations, and the progression of the quality of the solutions is recorded for analysis. Only an allowed range for every parameter was provided to the EAs, but no known to be reasonable starting values. Two types of EA were applied and compared: a Genetic Algorithm (GA) and an Evolution Strategy (ES); GAs use binary strings as the genotype, while ESs use a vector of floating-point numbers. In general ESs are known to be more efficient than GAs when dealing with numerical optimization problems [Bäck96], it was investigated if this is also true for the problem of SAT solver heuristic optimization.

The target of optimization in this work was the open-source SAT solver MINISAT [Eén04]. This solver is very small (about 2000 lines of C code) and easy to modify,

configuration for the full training set. This would provide more detailed information to the later stages but requires some changes to how those work.

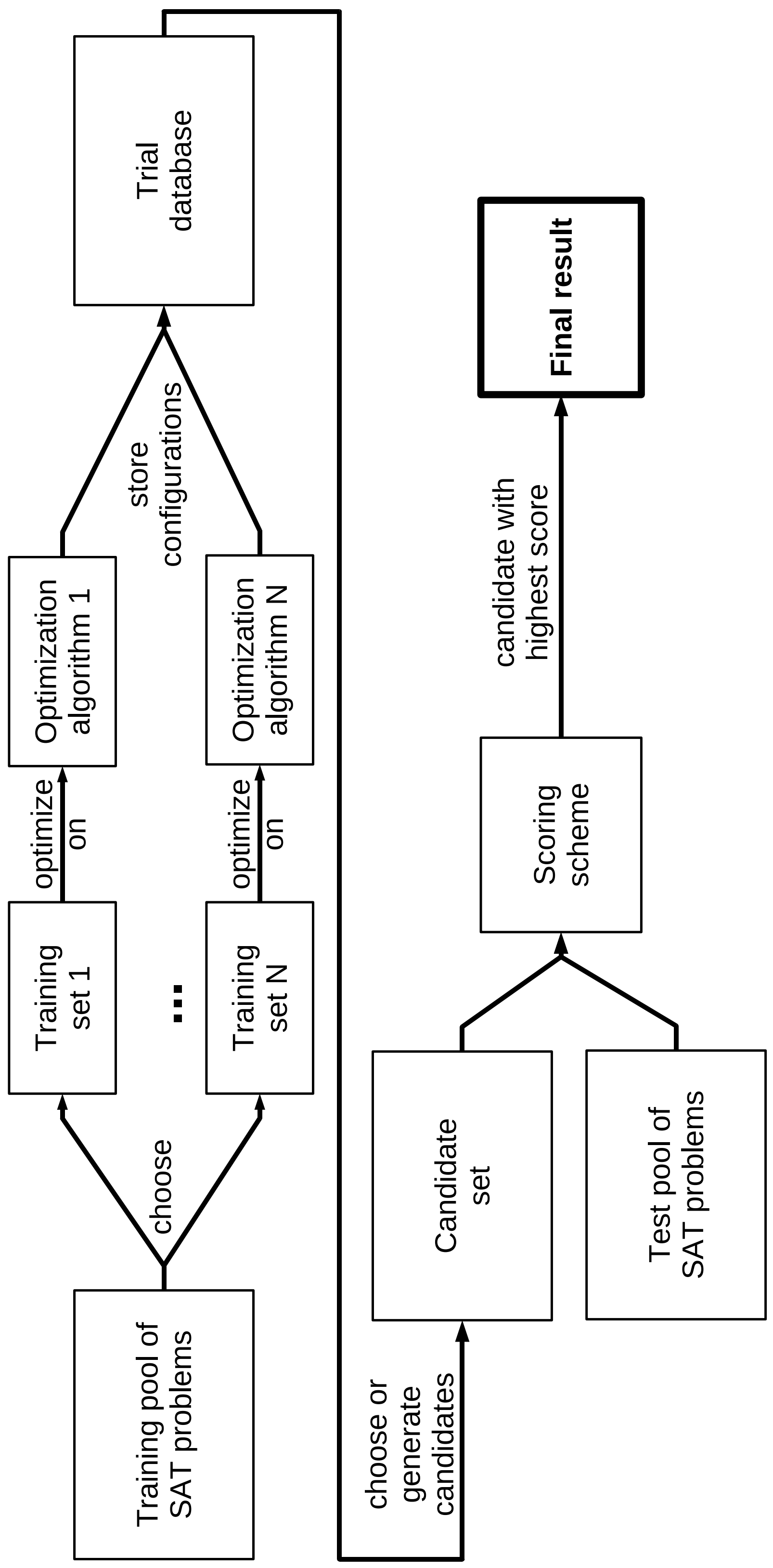


Figure 1.5: The automatic optimization procedure

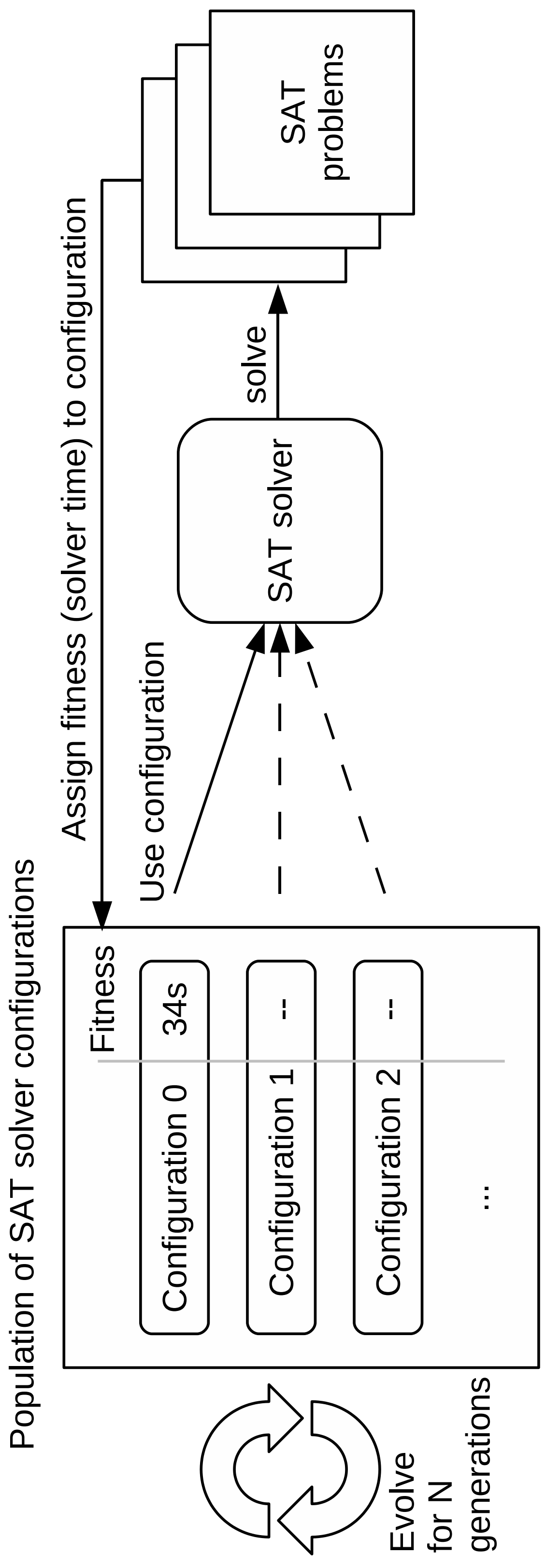


Figure 1.6: Optimizing a SAT solver with EAs using a training set of problems

yet also very efficient, and has won several SAT competitions. MINISAT seemed a particularly good choice for the target since it has very compact and clear code, but also relatively simple yet efficient heuristics, which would help to analyze the results of the optimization. Seven numerical parameters that affect various heuristics of the solver were the variables of the *SAT solver optimization problem*. All SAT problems were taken from the publicly available SAT competition portfolio, in particular the SAT-Race 2006.

1.9 Thesis Organization

The initial chapters of this work are a recapitulation of the basics and the state of the art of the subjects relevant for this work. Chapter 2 introduces the Boolean satisfiability problem and some related mathematical concepts, some variants of SAT and some applications and their transformation into SAT problems. Chapter 3 explains the details of the DPLL algorithm and the many extensions that make it so effective for solving real-world SAT problems. Chapter 4 introduces Genetic Algorithms and the features used in the implementation for this work, and chapter 5 does the same for Evolution Strategies.

Chapter 6 contains the main part of this work, the experimental results. The chapter starts with an investigation of the imprecision of run time measurements under Linux; this was examined because the objective value of the SAT solver optimization problem was the program run time. The following section tests the EAs for correctness by applying them on test functions with known structures. After this the EAs are tested on noisy test functions, to investigate the effect of random variations (like the solving times later) on the convergence speed and reliability of GAs and ESs. In the next section fitness landscapes were generated for the MINISAT heuristic parameters (to the author's knowledge the use of fitness landscapes is novel in the context of SAT solver optimization and is a first in this work). This was to investigate what kind of effect changing these values has on the solving time, and also to have data to compare the EA's results against. In the following section the GA and ES are applied on the SAT solver and the results compared, with the goal to decide which of the EAs is the more appropriate to use for SAT solver optimization. Next, a large optimization experiment with many training problem sets was performed using the better of the two EAs as determined in the previous section. In the last section, result candidates are chosen from the data gathered in the last experiment, and the candidates are tested on a large set of benchmark problems and the results compared to those using the default parameters.

Chapter 2

The Boolean Satisfiability Problem

2.1 Boolean algebra

Boolean algebra, named after its inventor, the English mathematician George Boole [Boole54], deals with the *truth values* "true" and "false" rather than the set of real numbers as in the algebra taught in school. A Boolean *variable* can take either of the two truth values from the set $B = \{\textit{true}, \textit{false}\}$ or it can be unassigned (*free*). For brevity, the truth values are often represented by the digits 1 for *true* and 0 for *false* respectively.

Boolean *operations* take truth values as operands and result in another truth value; they can be represented as *truth tables* which list the result of the operation for all possible combinations of the operand values. Tab. 2.1 shows the truth tables of the standard Boolean operations AND, OR and NOT.

In the SAT literature and in electronics the symbol for the Boolean OR operator is often written as "$+$" rather than the "$\vee$" used in mathematics and logic. Boolean AND can be indicated by simple concatenation of literals or expressions without an operator between them, or the symbol "$\cdot$" rather than "$\wedge$". Negation can be indicated by a line over the negated part of the formula rather than a prefixed "$\neg$". The Boolean formula $a \vee (b \wedge \neg c)$ can therefore also be written as $a + b \cdot \overline{c}$ or $a + b\overline{c}$. Furthermore the XOR (exclusive-OR) operator is sometimes represented as $\oplus$, and its complement XNOR (equivalence) as $\equiv$. The Boolean truth value *true* can be represented by 1 and

p	q	NOT p	p AND q	p OR q
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Table 2.1: Truth table of the basic Boolean operations

false by 0. This work uses the symbols 1, 0, $+$, $\cdot$ (or concatenation where clear), overlined variables or sub-expressions, $\oplus$ and $\equiv$ for *true*, *false*, OR, AND, NOT, XOR and XNOR respectively.

A Boolean *function* in n Boolean variables maps the n-dimensional space B^n on the Boolean space B. Any Boolean function can be represented by a truth table. Truth tables are a *canonical representation*, meaning that for any Boolean function there is exactly one and only one representation. A *propositional Boolean formula* contains only the Boolean operations AND, OR and NOT together with truth value constants and/or Boolean variables. Eq. 2.1 shows an example for a propositional Boolean formula (the computation of the Boolean XOR operation using the basic AND, OR and NOT operations).

$$f(v_1, v_2) = (v_1 \cdot \overline{v_2}) + (\overline{v_1} \cdot v_2) = v_1 \oplus v_2 \tag{2.1}$$

2.2 SAT

The objective of the *Boolean satisfiability problem (SAT)* is to prove that a Boolean formula f has a set of assignments to its propositional variables so that $f = 1$ or that there is no such assignment (in that case, it follows that $f = 0$). In the former case, a *model* (a set of variable assignments that make $f = 1$) of f was found and f is *satisfiable* (*SAT*), in the latter f is *unsatisfiable* (*UNSAT*).

Related to the SAT problem is the *tautology problem* which asks if a given Boolean formula evaluates to 1 for *all* possible assignments to its variables. If it does, the formula is called a *tautology*. Tautologies are always satisfiable, and the inverse of a tautology is always unsatisfiable.

2.2.1 Conjunctive normal form

SAT problems are usually given in *conjunctive normal form* (CNF), sometimes also called *product of sum* (POS) form. A *literal* is either a Boolean variable (for example x, a *positive* literal) or the complement of a variable (for example $\overline{x}$, a *negative* literal). A *clause* is a disjunction (Boolean OR) of literals, and a CNF is conjunction (Boolean AND) of clauses. For example, the following CNF contains three clauses consisting of two, three and one literals each:

$$f = (v_1 + v_2)(v_2 + \overline{v_3} + v_4)(v_5) \tag{2.2}$$

CNF is not a *canonical* representation, meaning that for the same Boolean function there may be many CNFs that describe it, for example an infinite number of (unsatisfiable) CNFs describe the constant function 0. Because $x + x = x$ for any Boolean formula x, multiple occurrences of the same literal can be removed from clauses without changing the function the CNF represents. A CNF can contain clauses that may be removed entirely without changing the function the CNF represents. Such clauses are called *redundant* clauses, and all the clauses that can not be removed without changing the represented function are called *irredundant*. If a clause C_{large} has all the same literals as another clause C_{small} as well as a some additional literals, the clause C_{large} is said to be *subsumed* by the clause C_{small}. For example, the clause $(v_1 + v_2)$ is subsumed by the clause (v_1). Subsumed clauses are redundant. If a clause contains a positive literal and a negative literal of the same variable, the clause is called a *tautology clause*. Tautology clauses are equal to 1 because $x + \overline{x} = 1$ for any Boolean formula x. Since $c \cdot 1 = c$ for any Boolean formula c, tautology clauses are always redundant.

The value of a CNF is 1 if and only if every clause contains at least one literal that evaluates to 1. The example CNF Eq. 2.2 is satisfiable because it has the model $\{v_1 = 1, v_2 = 1, v_3 = 0, v_4 = 1, v_5 = 1\}$. There can be many models for a given SAT problem, for example $\{v_1 = 1, v_2 = 1, v_3 = 1, v_4 = 1, v_5 = 1\}$ is also a model of the CNF. Finding one model is enough to prove satisfiability or respectively show that the CNF is not unsatisfiable.

2.2.1.1 Boolean constraint propagation

Clauses with only one literal are called *unit-literal clauses* or *unit clauses*, for example the clause (v_5) in Eq. 2.2 is a unit clause. Unit clauses force an assignment to the literal in them because the CNF could not be satisfied otherwise, and are therefore also called *implications* [Silva96]. The following CNF is unsatisfiable because the unit clauses (v_1) and $(\neg v_1)$ require the mutually exclusive assignments $v_1 = 1$ and $v_1 = 0$.

$$f = (v_1)(\overline{v_1})(v_2) = 0 \tag{2.3}$$

It is therefore equivalent to the truth value 0.

Implications can lead to further implications, which have to be assigned iteratively until no further implications arise. In SAT solvers the iterative assigning of implications is called *Boolean constraint propagation (BCP)* [Silva96] and is a major part of the algorithm.

2.2.1.2 Pure literal rule

If a variable occurs in only one *polarity* (only as positive literals or only as negative literals) in the CNF, the literal can be assigned *true* without changing the satisfiability of the CNF. This is called the *pure literal rule*.

2.2.1.3 Consensus and resolution

The *consensus theorem* of Boolean algebra exists for *disjunctive normal form* (DNF, a sum of products form) (Eq. 2.4) as well as for CNF (Eq. 2.5).

$$xy + \overline{x}z + yz = xy + \overline{x}z \tag{2.4}$$

$$(x + y)(\overline{x} + z)(y + z) = (x + y)(\overline{x} + z) \tag{2.5}$$

The *consensus term* is the redundant one in each form (yz and $(y + z)$). Obviously the theorem works in both directions, so from the two clauses $(x+y)(\overline{x}+z)$ the new, redundant in respect to the originals clause $(y + z)$ can be generated. The consensus theorem can be generalized to the *consensus law* [Hachtel96] which gives a rule how to create a new, redundant clause from two clauses. Given two sets of literals S_1 and S_2 and a literal x, the consensus law is Eq. 2.6.

$$\begin{aligned}&(x + \sum S_1)(\overline{x} + \sum S_2)\\ = &(x + \sum S_1)(\overline{x} + \sum S_2)(\sum S_1 + \sum S_2)\end{aligned} \tag{2.6}$$

The generated clause $(\sum S_1 + \sum S_2)$ in Eq. 2.6 is called the *resolvent* of clauses $(x+\sum S_1)$ and $(x+\sum S_2)$, and the operation that generates the new clause $(\sum S_1 + \sum S_2)$ is called *resolution*. The two clauses $(x + \sum S_1)$ and $(\overline{x} + \sum S_2)$ are called *antecedents* and x is called the *pivot* variable. The resolvent is always redundant in respect to the original clauses. For example, from the clauses $(x + a + b)$ and $(\overline{x} + \overline{c})$ with x as the pivot the resolvent $(a+b+\overline{c})$ can be generated using the consensus law.

The predecessor of the DPLL algorithm, usually called the *DP algorithm* [Davis60], was based on iterative resolution of the SAT problem's clauses until either the CNF consists of only a single clause (which is obviously satisfiable) or a 0 is generated (the CNF is unsatisfiable). The DP algorithm suffers from rapid memory exhaustion due to the potentially huge number of generated clauses and is therefore usually not used in practice. Resolution is also important for understanding the learning of new clauses in DPLL SAT solvers.

2.2.1.4 Distance

Given two sets of literals S_1 and S_2 the *distance* between them is the number of literals x so that the literal x is in one clause ($x \in S_1$) and the inverse of the literal $\overline{x}$ is in the other ($\overline{x} \in S_2$). The distance can also be defined for two clauses C_1 and C_2 by defining the set $S1$ as the set of literals of C_1 and the set S_2 as the set of literals of C_2. For example the distance between the clauses $(x+a+b)$ and $(\overline{x}+\overline{c})$ is 1, because both contain the variable x, as a positive literal in the first clause and as a negative literal in the second clause. The resolvent of two clauses that have a distance greater than 1 is a tautology clause, for example $(a + b + c)$ resolved with $(\overline{a} + \overline{b} + d)$ results in $(b+\overline{b}+c+d)$. Therefore resolution in CNFs only results in non-redundant clauses when the two clauses resolved have a distance of 1.

2.2.2 Complexity of SAT

Verifying a possible solution of a SAT problem means checking if the model (the set of satisfying variable assignments) actually does make the formula evaluate to 1. SAT is very easy to verify, especially if the problem was given as a CNF: given a prospective model, simply check if every clause in the CNF contains at least one satisfied literal. If that is the case, the model is valid and the expression is satisfiable.

Solving SAT, on the other hand, is very hard, meaning computationally expensive in time and used memory for problems with even a moderate number of variables and clauses. There is so far no known algorithm which does not in the worst case need an exponentially increasing number of steps in the number of the variables. SAT is the seminal *NP-complete* problem (first proved by S. Cook in 1971). SAT was the first problem that was proved to be NP-complete, and it is still an open question if efficient algorithms exist for NP-complete problems. In the following an overview of the notation for describing the complexity of an algorithm and the problem classes P and NP will be presented.

2.2.3 Big O notation

Computational complexity theory is concerned with the mathematical assessment of the resources required to solve a problem depending on the size of the problem instance. Resources are for example time steps and memory space. The problems can then be sorted into *complexity classes*.

An *algorithm* is a method that computes the correct answer for a problem for every possible valid input. A very simple algorithm is for example the manual computation of the sum of two numbers. If each number has n digits, then the sum is computed in about $2n$ steps (including carry-over additions). The effort (in time steps) to compute the sum of two numbers therefore progresses in a linear fashion depending on the size of the numbers n.

Different computers that run this algorithm may differ greatly in the execution time required due to their architecture, but ultimately a computer that is 10 times as fast as another will also compute a sum about 10 times faster. To abstract away such implementation differences, the complexity of algorithms is usually given in *Big O notation*. Big O notation removes any additive and multiplicative constants and only gives the dominant term of a complexity measure. For the addition algorithm, the required $2n$ operations is noted as $O(n)$ in Big O notation.

Another simple algorithm is the multiplication of two numbers with n digits, which requires about n^2 operations (additions) when using the "long multiplication" method that is taught in school. Long multiplication is therefore an algorithm with a $O(n^2)$ time complexity. Multiplication is not inherently $O(n^2)$ though, just the long multiplication algorithm is; in fact, a faster algorithm for muliplication exists that is only $O(n^{\log_2 3})$ [Karatsuba62] and even faster ones have been invented.

Algorithms whose time complexity is *polynomial* ($O(n^k), k > 0$) are called *feasible*. These are of the greatest interest for actual applications due to their good behavior for large problem sizes. Problems that have no feasible algorithms are called *intractable*.

As computers become faster, the maximum problem size solvable in the same time becomes greater to some degree. In the following it is assumed that a "new" computer is 1000 times as fast as the "old" computer. For a problem whose algorithm runs in $O(n)$, the new computer will be able to solve a problem that is 1000 times larger in the same time as the old computer. But for a $O(n^2)$ algorithm the problem size solvable in the same time with the new computer is only about 31.6 ($\sqrt{1000}$) times the original size, and for a $O(n^3)$ algorithm only 10 times. Larger exponents of n mean even lower solvable problem size gains, although it is at least multiplied with some constant factor. Very large exponents are uncommon though for natural problems, which are usually limited to about $O(n^4)$.

In contrast, for a problem with a $O(2^n)$ algorithm the solvable problem size increases only about by a constant 10, an additive rather than a multiplicative gain. Clearly very large problems remain de facto unsolvable if the algorithm has exponential time complexity.

2.2.4 Complexity class P

Problems that have a "yes or no" answer are called *decision problems* or *languages*. For example deciding if a graph contains an *Euler tour*, a path that visits each edge of the graph exactly once, is a decision problem. This problem is named after Swiss mathematician Leonhard Euler (1707 - 1783), who was the first to solve it.

The first instance of the problem is known as the *Seven Bridges of Königsberg problem*, because Königsberg has two large islands formed by the Pregel River, which are connected to the mainland and each other by seven bridges (Fig. 2.1). The question is if it is possible to walk on a path through the city so that each bridge was crossed only and exactly once. Euler proved in 1736 that this was not possible.

Fig. 2.1 shows a simplified map of Königsberg with the seven bridges and the two islands. The four distinct landmasses (the two sides of the river and the two islands) can be abstracted to the vertices, and the bridges to the edges of a *graph* (a set of vertices connected with edges), as shown on the right of Fig. 2.1. The problem is now simplified to finding a path through the graph that traverses each edge exactly once. Euler found that a graph contains an Euler tour if and only if every vertex has an even number of incoming edges. Therefore the algorithm for solving the Euler tour problem has a time complexity of $O(m)$ where m is the number of vertices. The Seven Bridges of Königsberg problem was shown to have a negative solution since every vertex had an uneven number of incoming edges.

The *Boolean formula evaluation problem* is another decision problem. It asks for a given Boolean formula and values to assign for all of its variables if the formula

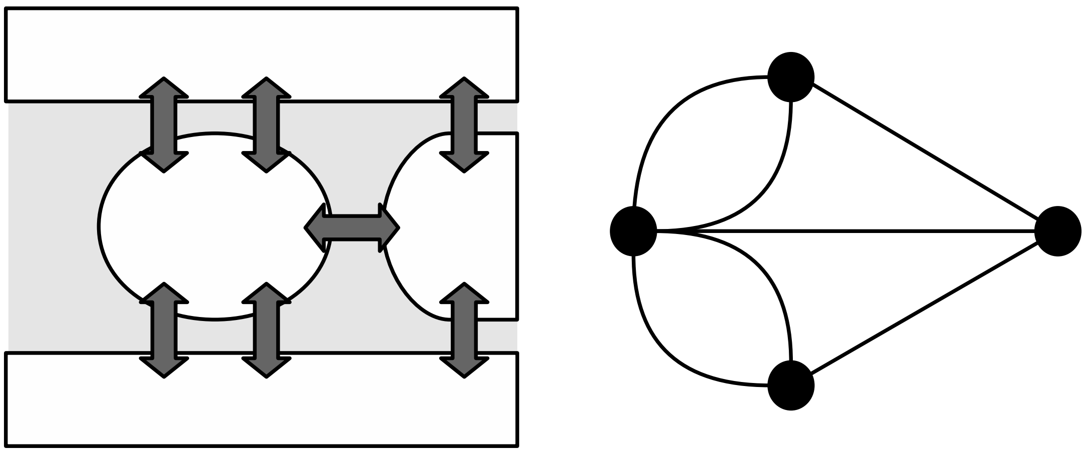


Figure 2.1: Illustration of the Seven Bridges of Königsberg problem

evaluates to true. When the Boolean formula is given as a CNF of the length n, the algorithm that solves this problem is $O(n)$.

This leads to the definition of P (for polynomial), which is the class of decision problems or languages that are solvable in polynomial time on a deterministic Turing machine. A Turing machine is a very simple model of computer with a memory tape that is still an accurate representation of the much more sophisticated machines that are actually used in practice. It was invented in 1936 by English mathematician Alan Turing (1912 - 1954), who is considered to be the father of modern computer science and after whom the Turing Award, the computer science equivalent of the Nobel Prize, is named. Both the Euler tour problem as well as the Boolean formula evaluation problem belong to P. Problems that are in P are by definition feasible.

2.2.5 Complexity class NP

As opposed to the Euler tour problem, there are also problems from graph theory that do not belong to P. For example a *Hamiltonian cycle* is a path in a graph that visits each vertex exactly once and also returns to the starting vertex. There is no known algorithm that can decide in polynomial time if a given graph contains a Hamiltonian cycle. Closely related to this problem is the well-known *Traveling Salesman Problem (TSP)*, where each edge of the graph has a cost associated with it and the question is if there is a Hamiltonian cycle so that the sum of costs of the traversed edges is less than a given budget. There is also no known polynomial time algorithm for TSP. SAT has a time complexity of $O(m2^n)$ for m being the size of the CNF and n variables and is therefore also not in P.

Although solutions for all these problems can apparently not be computed in polynomial time, it is possible to *verify a candidate solution in polynomial time* (verify here means checking the validity of the solution). For the Hamiltonian cycle and TSP, a path through the graph can be verified quickly, while for SAT the verification is in essence the same as the Boolean formula evaluation problem.

These decision problems have polynomial time verification algorithms, and for every positive ("yes") answer there is a proof that is polynomial in size in respect to the problem size, for example a Hamiltonian cycle, a cycle with the smallest cost for TSP or a satisfying assignment for SAT.

These problems belong to the complexity class NP (which stands for *Non-deterministic Polynomial time*), the class of decision problems whose proofs *can be verified in polynomial time* by a deterministic Turing machine. As an equivalent definition NP is the class of decision problems whose solutions can be computed in polynomial time on a non-deterministic Turing machine. Any language in P is also automatically in NP, since problems in P are solvable in polynomial time and therefore obviously also verifiable in polynomial time.

2.2.6 The P=NP problem

One of the most important open problems in Mathematics is the P=NP problem: since problems in NP are verifiable in polynomial time and with polynomial size proofs, could they be solvable in polynomial time as well? Or are these classes truly distinct?

P=NP is one of the Millenium Prize Problems: originally 7, now 6 remaining open questions in Mathematics that are considered to be of particular historic interest or importance. They were named by the Clay Mathematics Institute of Cambridge, Massachusetts (CMI), which awards 1 million US$ for solving one of the problems.

2.2.6.1 NP-completeness

An important concept for the P=NP problem is *NP-completeness*. A language is *NP-complete* if it is in NP and every other language in NP can be transformed into it (is *reducible* to it) in polynomial time. If a language satisfies only the latter condition (every language in NP is reducible to it) then it is called *NP-hard* no matter if it is itself in NP or not. It was also shown that P=NP if and only if a NP-complete language is in P.

Stephen Cook proved in his 1971 paper "The Complexity of Theorem Proving Procedures" [Cook71] that SAT is NP-complete (*Cook's theorem*), and received the Turing Award for this work in 1982. Richard Karp published the paper "Reducibility Among Combinatorial Problems" [Karp72] in 1972 in which he proved the NP-completeness of 21 problems (including many industrial problems), for which and other contributions he would receive a Turing Award in 1985. Then in 1979 Michael R. Garey and David S. Johnson published their book "Computers and Intractability: A Guide to the Theory of NP-Completeness" [Garey90] which contains over 300

NP-complete problems. At present, thousands of NP-complete problems are known, many of which have important applications.

The large number of applications in which NP-complete problems are to solve indicates how important P=NP is. But there are also some important problems that are in NP, but are not NP-complete. These include the problems of determining graph isomorphism and proving that a number is a prime or a composite. NP is not by far the most difficult problem class though. For example, there are problems that are doubly exponential, as well as undecidable problems for which no algorithm can be designed at all, for example the Halting Problem for Turing machines.

2.2.7 Variants of SAT and related problems

In the following some variants of SAT and some problems that are closely related to SAT will be presented. Some are only different in the prescribed form of the CNF, others change the objective of the problem substantially. The changes can lead to the problem becoming easier or harder than the general SAT problem.

2.2.7.1 k-SAT

The problem of determining the satisfiability of a CNF in which all clauses contain exactly k literals is called *k-satisfiability* (*k-SAT*). For $k = 3$ this is called *3-satisfiability* (*3-SAT*). Any k-SAT problem of $k > 3$ can be reduced to a 3-SAT problem (this introduces new variables into the transformed CNF) [Karp72]. Since 3-SAT was proved to be NP-complete [Karp72], it must follow that k-SAT for $k > 3$ is NP-complete.

In contrast, the *2-satisfiability* problem (*2-SAT*), where each clause has exactly 2 literals, is easier than 3-SAT and there exists a polynomial-time algorithm that solves it.

2.2.7.2 Horn-satisfiability (HORNSAT)

A clause that contains at most one positive literal is called a *Horn clause* (after American mathematician Alfred Horn). The *Horn-satisfiability* problem (*HORNSAT*) is determining the satisfiability of a CNF containing only Horn clauses. HORNSAT is solvable in polynomial time and is in fact P-complete.

2.2.7.3 Maximum Satisfiability (MAX-SAT)

Given a CNF, the *Maximum Satisfiability problem* (*MAX-SAT*) asks for a set of assignments to the CNF's variables that maximizes the number of satisfied clauses (or in other words minimizes the number of unsatisfied clauses). This makes the SAT problem a special case of MAX-SAT. MAX-SAT is *NP-hard* rather than NP-complete, even for CNFs where all clauses have exactly 2 literals (*MAX-2-SAT*). NP-hard problems are problems that are *at least as hard as problems in NP* but not necessarily in NP themselves.

2.2.7.4 Unsatisfiable cores

An *unsatisfiable core* of an unsatisfiable CNF is a subset of its clauses that form a still unsatisfiable CNF. A *minimal unsatisfiable core* contains a subset of the CNF's clauses that is unsatisfiable, and which becomes satisfiable if any of the clauses is removed. A *minimum unsatisfiable core* is the smallest possible subset of the CNF's clauses that is unsatisfiable. Computing unsatisfiable cores is important for various industrial applications, and DPLL SAT solvers can be adapted to find minimal cores [Lynce04], though no practical algorithm is known that can find minimum cores.

2.2.7.5 Satisfiability Modulo Theories (SMT)

Some problems of industrial interest are easier to formulate in logics that are more expressive than propositional logic and therefore allow a higher level of abstraction. An example is the verification of pipelined processors [Burch94], where the logic of *Equality with Uninterpreted Functions* (*EUF*) can be used to abstractly represent the data operations in the processor. Due to the great success of SAT solvers, DPLL-based and otherwise, it has been attempted to translate the progress made there on the more expressive logics. This is the *Satisfiability Modulo Theories (SMT)* problem: given a *background theory* T and a formula F containing propositional atoms as well as atoms over the theory T, determine if F is *T-satisfiable*.

One way to solve SMT is called the *eager approach* and consists of translating the SMT instance into a Boolean SAT instance, which can then be solved with any SAT solver. For example in [Bryant01] methods are described for how to efficiently translate pipelined processor verification SMT problems as generated in [Burch94] into Boolean SAT. The eager approach has the obvious advantage of being able to rely on off-the-shelf SAT solver technology, but a major drawback is that it can not take advantage of simplifying properties of the background theory T during the solving process. Therefore different ways of solving SMT were investigated that integrate DPLL and a solver for theory T directly: this is called the *lazy approach* (for example the *DPLL(T)* procedure [Nieuwenhuis06]). For an overview of SMT, see for example [Nieuwenhuis05].

2.2.7.6 Quantified Boolean Formulas (QBF, QSAT)

The *Quantified Boolean Formula* problem (QBF, also known as *QSAT* (Quantified SAT)) is closely related to, and is in fact a generalization of SAT. A QBF has the following form:

$$Q_1 x_1 \dots Q_n x_n \phi \tag{2.7}$$

ϕ is a Boolean expression in n variables x_i $(i = 1 \dots n)$. The Q_i are *quantors*. Every Q_i is either a *existential quantifier* (symbol $\exists$) or a *universal quantifier* (symbol $\forall$). In words $\exists x \phi$ means "there exists (at least) one truth value (1 oder 0) for x so that the expression ϕ becomes true", and $\forall x \phi$ means "for all truth values (1 oder 0) of x the expression ϕ becomes true". Different quantifiers can be combined by concatenation, for example $\exists x \forall y \phi$ means "there exists a truth value for x so that for all y the expression ϕ becomes true". SAT is essentially a QBF with only existential quantifiers. Every QBF can be transformed into an equivalent, quantifier-free Boolean expression that can be SAT-checked with any SAT solver.

In contrast to a SAT problem, whose proof takes space that grows linearly in respect to the problem size, the proof of a QBF can take much more space. The complexity class *PSPACE*, the set of all decision problems which can be solved by a Turing machine using a polynomial amount of space, is a superset of the complexity class NP and QBF is the canonical PSPACE-complete problem. QBF solving has been applied to industrial problems like Bounded Model Checking [Dershowitz05] where they allow the formulas to be checked to be in a much more compact form.

2.3 Applications of SAT

In this section examples for SAT encodings of various problems will be presented. See for example [Marques-Silva08] for a list of applications of SAT.

2.3.1 Pigeonhole principle

The *pigeonhole principle* states that when given n objects and m containers $(n > m)$, then after placing the objects in the containers at least one container will hold more than one object.

The pigeonhole principle can be expressed as a SAT problem that asks if n pigeons can be placed in $n-1$ holes so that no hole holds more than one pigeon (the problem is obviously unsatisfiable since the pigeonhole principle states that this is impossible). Assuming n pigeons and $n-1$ holes, the variable $x_{i,j}$ is true if pigeon i is placed in hole j, therefore there will be $n(n-1)$ propositional variables in the SAT problem. A CNF can then be constructed that evaluates to true if two conditions are met: all pigeons are placed in at least one hole, and no two pigeons are placed in the same hole. The first condition can be expressed as a set of n clauses in the following form:

$$(x_{i,1} + x_{i,2} + ... + x_{i,n-1}), i \in \{1, ..., n\} \tag{2.8}$$

The second condition requires that there is no occurence of any combination of two pigeons occupying the same hole, which can be expressed as another set of clauses:

$$\begin{gathered}\overline{(x_{i,k} \cdot x_{j,k})} = (\overline{x_{i,k}} + \overline{x_{j,k}}); \\ k \in \{1, ..., n-1\}; i, j \in \{1, ..., n\}; i \neq j\end{gathered} \tag{2.9}$$

Since both conditions have to be fulfilled, the complete CNF consists of the combination of both clause sets. The CNF for n pigeons and $n-1$ holes has $n(n-1)$ propositional variables and $n + (n-1)(n-1)\frac{n}{2}$ clauses. As an example, for the case of $n = 4$ pigeons and 3 holes the first clause set is:

$$\begin{gathered}(x_{1,1} + x_{1,2} + x_{1,3}) \\ (x_{2,1} + x_{2,2} + x_{2,3}) \\ (x_{3,1} + x_{3,2} + x_{3,3}) \\ (x_{4,1} + x_{4,2} + x_{4,3})\end{gathered}$$

The second clause set for $n = 4$ is:

$$\begin{aligned}(\overline{x}_{1,1} + \overline{x}_{2,1})(\overline{x}_{1,1} + \overline{x}_{3,1})(\overline{x}_{1,1} + \overline{x}_{4,1}) \\ (\overline{x}_{2,1} + \overline{x}_{3,1})(\overline{x}_{2,1} + \overline{x}_{4,1}) \\ (\overline{x}_{3,1} + \overline{x}_{4,1}) \\ (\overline{x}_{1,2} + \overline{x}_{2,2})(\overline{x}_{1,2} + \overline{x}_{3,2})(\overline{x}_{1,2} + \overline{x}_{4,2}) \\ (\overline{x}_{2,2} + \overline{x}_{3,2})(\overline{x}_{2,2} + \overline{x}_{4,2}) \\ (\overline{x}_{3,2} + \overline{x}_{4,2}) \\ (\overline{x}_{1,3} + \overline{x}_{2,3})(\overline{x}_{1,3} + \overline{x}_{3,3})(\overline{x}_{1,3} + \overline{x}_{4,3}) \\ (\overline{x}_{2,3} + \overline{x}_{3,3})(\overline{x}_{2,3} + \overline{x}_{4,3}) \\ (\overline{x}_{3,3} + \overline{x}_{4,3})\end{aligned}$$

Using these clauses as input for a SAT solver will make the solver look for an assignment to the variables so that the entire formula evaluates to true, which will of course fail. Encoded pigeonhole problems for large n are used as benchmark problems to test SAT solvers. It was proved that any DPLL algorithm based SAT solver takes exponentionally rising time to solve pigeonhole problems [Haken84].

2.3.2 Tseitin transformation

Not every SAT encoding of a problem will likely result in a CNF directly. However, any Boolean formula can be transformed into CNF, but unfortunately this can take space and time that is exponential in the size of the original formula. For the purposes of SAT checking the *Tseitin transformation* [Tseitin68] can convert a Boolean formula into a CNF that has the same satisfiability as the original formula, in space

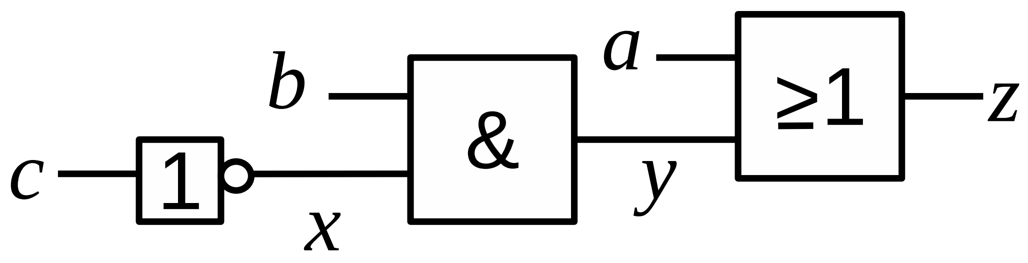


Figure 2.2: Example circuit for Tseitin transformation

and time linear to the original's size. This is achieved by introducing new variables for the sub-expressions of the original formula, so while the satisfiability is preserved the CNF is not the same formula as the original.

Tseitin transformation is especially useful when a *combinational Boolean circuit* has to be translated into CNF. A combinational Boolean circuit consists of Boolean logic gates and wires that connect them. The formula $a + b\overline{c}$ (which is in disjunctive normal form) can be tranformed into CNF by applying the negation, DeMorgan- and distribution rules:

$$\begin{aligned} a + b\overline{c} &= \overline{\overline{a + b\overline{c}}} = \overline{\overline{a}\,\overline{(b\overline{c})}} = \overline{\overline{a}(\overline{b} + c)} \\ &= \overline{\overline{a}\overline{b} + \overline{a}c} = \overline{(\overline{a}\overline{b})}\,\overline{(\overline{a}c)} = (a + b)(a + \overline{c}) \end{aligned}$$

Fig. 2.2 shows the representation of $a + b\overline{c}$ as a combinational Boolean circuit with the new variables x, y and z that are the "outputs" of the sub-expressions. The relation between the inputs, the operation and the outputs of every gate in the circuit can be described by the Boolean logic equality function (XNOR), yielding one *characteristic function* each. For example the characteristic function of the last OR gate in the circuit is $(z \equiv a + y)$. The characteristic function evaluates to 1 if the assignments to the input and output variables describe a valid configuration for the circuit, for example $a = 0, y = 0, z = 0$ is a valid state of the last gate and the characteristic function will be $(0 \equiv 0 + 0) = (0 \equiv 0) = 1$, while $a = 0, y = 0, z = 1$ is impossible and the characteristic function is $(1 \equiv 0 + 0) = (1 \equiv 0) = 0$.

The conjunction of the characteristic functions of all sub-expressions of the formula gives the characteristic function of the whole formula, which evaluates to 1 if the truth value assignments to the variables describe a valid state of the corresponding circuit and 0 if not. The characteristic function of the corresponding circuit of $a + b\overline{c}$ is $(x \equiv \overline{c})(y \equiv bx)(z \equiv a + y)$.

Translating the characteristic function into CNF is very easy because it is possible to pre-compute the CNF of the characteristic functions of the standard gates and then simply replace the variables of the inputs and outputs accordingly (NOT: Eqn. 2.10, OR: Eqn. 2.11, AND: Eqn. 2.12, XOR: Eqn. 2.13).

$$(x \equiv \overline{y}) = (\overline{x} + \overline{y})(x + y) \tag{2.10}$$

$$(x \equiv y + z) = (x + \overline{y})(x + \overline{z})(\overline{x} + y + z) \tag{2.11}$$

$$(x \equiv yz) = (\overline{x} + y)(\overline{x} + z)(x + \overline{y} + \overline{z}) \tag{2.12}$$

$$\begin{aligned}(x \equiv y \oplus z) &= \\ (x + y + \overline{z})(x + \overline{y} + z)(\overline{x} + y + z)&(\overline{x} + \overline{y} + \overline{z})\end{aligned} \tag{2.13}$$

For the example circuit in Fig. 2.2 the Tseitin transformation results in the following CNF:

$$\begin{aligned}(x \equiv \overline{c})(y \equiv bx)(z \equiv a + y)& \\ = (\overline{x} + \overline{c})(x + c)& \\ (\overline{y} + b)(\overline{y} + x)(y + \overline{b} + \overline{x})& \\ (z + \overline{a})(z + \overline{y})(\overline{z} + a + y)&\end{aligned}$$

2.3.3 Combinational equivalence checking

A recurring problem in combinational circuit design is the question if two circuits implement the same Boolean function. This can be formulated into a SAT problem by building a *miter circuit*. Both circuits have to have the same number of outputs obviously. To build the miter circuit, every corresponding output of the two circuits is connected to a XOR gate, and all outputs of the XOR gates are connected to an OR gate. The output of the OR gate is only ever 1 if and only if the two circuits implement different Boolean functions.

To formulate a SAT problem, the miter circuit is translated into a CNF using Tseitin transformation and the condition "output of the OR gate is 1" is enforced by adding a unit clause with the positive literal of the OR gate output. This means that the CNF is unsatisfiable if and only if the two circuits are equivalent.

Fig. 2.3 shows an example miter circuit. The two circuits to compare implement the formulae $(\overline{ab})$ and $\bar{a} + \bar{b}$, which is the same Boolean function (NAND). The new variables c, v, w, x, y, z have been introduced for the results of sub-expressions, where z is the output of the miter circuit's XOR gate. Since the original circuits have only one output, no OR gate is necessary for the miter circuit. The Tseitin transformation of this circuit, including the condition that the output of the XOR gate is 1 (adding a unit clause (z)), results in the following CNF:

$$\begin{aligned}(x \equiv \overline{a})(y \equiv \overline{b})(c \equiv x + y)(w \equiv ab)(v \equiv \overline{w})(z \equiv v \oplus c)(z)& \\ = (\overline{x} + \overline{a})(x + a)\ (\overline{y} + \overline{b})(y + b)\ (c + \overline{x})(c + \overline{y})(\overline{c} + x + y)& \\ (\overline{w} + a)(\overline{w} + b)(w + \overline{a} + \overline{b})\ (\overline{v} + \overline{w})(v + w)& \\ (z + v + \overline{c})(z + \overline{v} + c)(\overline{z} + v + c)(\overline{z} + \overline{v} + \overline{c})& \\ (z)&\end{aligned}$$

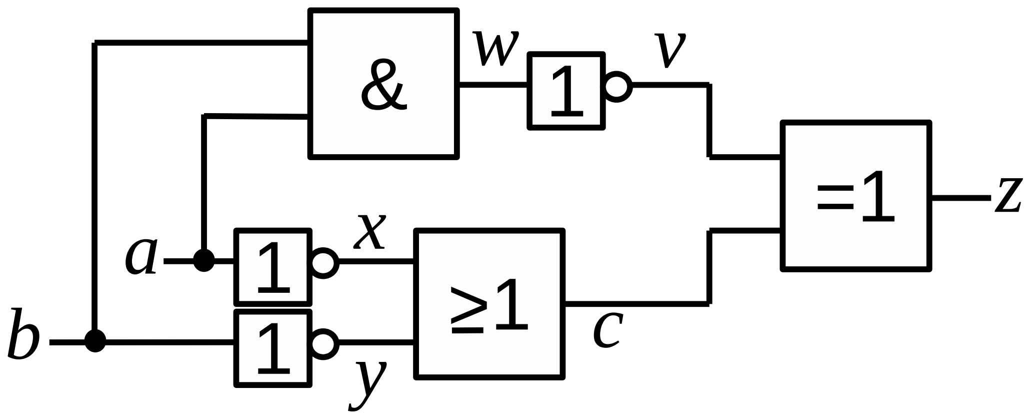


Figure 2.3: Example miter circuit for Combinational Equivalence Checking

The CNF is unsatisfiable because the two circuits under comparison both implement the same function, hence the XOR gate output can never be 1.

2.3.4 DIMACS CNF file format for SAT problems

The standard file format that is used for interchanging SAT problems in CNF was proposed by the *Center for Discrete Mathematics and Theoretical Computer Science (DIMACS)* [Dim]. It is a simple text file format that contains the clauses of the CNF as lines of numbers. DIMACS files usually have the `.cnf` suffix. MiniSAT, Chaff and most other SAT solvers can read DIMACS files.

DIMACS files consist of a preamble section that contains information about the problem instance, and the clause section that contains the actual clauses. In the preamble, *comment lines* start with the letter `c` followed by a space. The *problem line* is always the last line of the preamble and has the format "`p FORMAT VARIABLES CLAUSES`". FORMAT is always `cnf` for CNF data and VARIABLES and CLAUSES respectively give the number of variables and clauses of the problem.

The clause section follows after the problem line. Each variable of the problem has a non-zero integer value, and literals are written as positive or negative integers. The positive literal of the variable is written as the positive number, and the negative literal as the corresponding negative number. For example if the variable x was given the number 5, the positive literal x would be represented by 5 and the negative literal $\bar{x}$ would be represented as -5. The literals are separated by whitespace characters and the end of each clause is indicated by a zero digit. The CNF of the example miter circuit in Fig. 2.3 results in the DIMACS file shown in Fig. 2.4.

```
c Example miter circuit for combinational equivalence checking
c Corresponding numbers of the original variables:
c a=1, b=2, c=3, v=4, w=5, x=6, y=7, z=8
p cnf 8 17
-6 -1 0
6 1 0
-7 -2 0
7 2 0
3 -6 0
3 -7 0
-3 6 7 0
-5 1 0
-5 2 0
5 -1 -2 0
-4 -5 0
4 5 0
8 4 -3 0
8 -4 3 0
-8 4 3 0
-8 -4 -3 0
8 0
```

Figure 2.4: Sample DIMACS SAT problem file

2.3.4.1 Shuffled problems

A shuffled SAT problem is one in which the variable indices and the order of the clauses has been randomly permutated (additionally the polarity of all literals may be inverted). It is essentially the same problem as the original, but due to the heuristics used in SAT solvers a shuffled problem can take a very different amount of time to solve; the time may be shorter than for the original problem too though [Audemard08]. In various parts of the experiments in this experiments shuffled problems were used. The SAT problems were shuffled with the REORDER program from the SAT competition toolbox[1]. REORDER takes a DIMACS CNF file and a random *seed* (an integer number) that determines the changes made to the CNF file as arguments, and outputs the shuffled problem in DIMACS format as well.

2.3.5 Bounded model checking

The *model checking* problem consists of automatically (exhaustive) testing if a given (simplified) model of a software or hardware system meets some specification. These are often *temporal properties* that describe the state of the model at some point in

[1]Source code was taken from http://www.satcompetition.org/2003/TOOLBOX/reorder.c

time or throughout a time period, for example "after a request the system will send an acknowledgement in at most 2 time steps" or "the valid output signal will be held for at least 5 time steps". Temporal properties can be specified with *temporal logic*, for example *Linear Temporal Logic* (LTL) or *Computation Tree Logic* (CTL). It is often required to ensure *safety conditions*, for example that a state machine never enters a deadlock, and *liveness conditions*, for example that it always acknowledges a request.

In the 1980s the first model checking methods that were implemented explicitly enumerated states of a system to check properties. Such methods are only usable for very small systems, since the number of states grows exponentially with the number of state variables (the *state explosion problem*). Even a small sequential digital circuit with 20 binary storage elements (latches or Flip-Flops) will have $2^{20} \approx 10^6$ possible states, making explicit storage of states and computation of successor and predecessor states unfeasible very quickly.

Symbolic model checking [Burch92] was invented around 1990 and is much more efficient than explicit enumeration. It uses *Binary Decision Diagrams* (BDDs), a canonical, graph-based representation of Boolean functions, to efficiently represent sets of states and the system's transition behavior. With BDDs the successor and predecessors of an entire set of states can be computed much more efficiently than with a explicit representation. To check a specification, a breadth first search of the state space is performed. A major success of BDD based symbolic model checking was achieved by Clarke and his students 1992 at Carnegie-Mellon University using the SMV model checker. They found several errors in the cache coherence protocol of the IEEE Futurebus+ Standard, which had not been detected previously with other non-formal methods. This was the first time errors were found in an IEEE standard using formal methods. BDD based symbolic model checking can handle systems with roughly 50-100 storage elements, but often requires some amount of manual optimization for larger systems because BDDs suffer from exponential memory requirement.

Bounded model checking (BMC) [Biere99], invented around 1999, is a SAT-based model checking method that is capable of checking larger systems than symbolic model checking. The underlying principle of BMC is replicate the transition system under test several times and connecting these copies in series (this is called *unfolding*, Fig. 2.5). Each copy represents the system at a certain time step (for example a clock cycle in the case of a digital circuit), and the copies that follow after it represent the system at later time steps, so the first copy in the sequence represents the system's state at time $t+0$, the next at time $t+1$ and so on. This also means that any properties are only checked for the bounded number of time steps for which the circuit is unfolded (although it is possible to check infinite looping paths). The unfolded system, together with terms for the initial state and the specification to be checked, is then translated into a propositional formula. If this formula is unsatisfiable the specification is correct, or if it is satisfiable the solution encodes a *counterexample* or *witness*

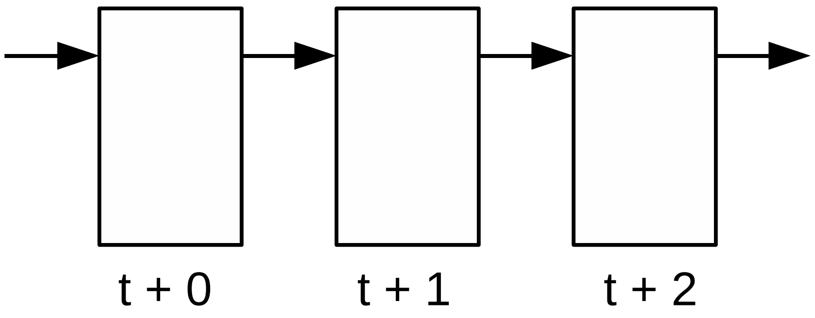


Figure 2.5: Principle of BMC: unfolding a system over several time steps

that describes a situation where the specification does not hold.

The advantages of BMC are that they avoid the memory requirement problem of BDDs and that checking even of large systems can be fast. The disadvantage of BMC is that it is generally not complete, meaning that not all specifications can be proved to hold or not due to limits on the length of unfolding. BMC complements rather than replaces other model checking methods.

Generating the propositional formula of the unfolded transition system is accomplished by using the system's *transition relation*. A transition system with n storage elements $s_1, \ldots, s_n$ and m inputs $i_1, \ldots, i_m$ can be described with n *transition functions* $\delta_1(s_1, \ldots, s_n, i_1, \ldots, i_m)$ - $\delta_n(s_1, \ldots, s_n, i_1, \ldots, i_m)$ (the input functions of the storage elements). By introducing the new variables $s'_1 \ldots, s'_n$ for the next values of the state variables the system's transition behavior can also be described by a single formula, its *transition relation* T (Eq. 2.14). For example, Eqn. 2.15 shows the transition relation of a simple 2-bit counter (a is the low bit and b is the high bit).

$$T = \prod_{r=1}^{n} (s'_r \equiv \delta_r(s_1, \ldots, s_n, i_1, \ldots, i_m)) \tag{2.14}$$

$$T = (a' \equiv \overline{a})(b' \equiv a \oplus b) \tag{2.15}$$

When dealing with several time steps, each time step t has a set of corresponding current state variables $s_{t,1} \ldots s_{t,n}$ and current input values $i_{t,1} \ldots i_{t,m}$. The transition relation of a system unfolded over k time steps is simply the the conjunction of all transition relations of each single time step, where the next and current state variables and inputs are replaced accordingly.

An example specification to check for the 2-bit counter (Eqn. 2.15) is "Can the counter reach the state where both variables a and b are 1 within two time steps starting from the state where both variables are 0". Obviously this is impossible since the counter will go through the states in the order $(0,0)$, $(0,1)$, $(1,0)$, $(1,1)$ and therefore reaches the $(1,1)$ state in the third time step at the earliest. Let the state variables of the counter at time step 0, 1 and 2 be be a_0, b_0, a_1, b_1 and a_2, b_2. This system has no input variables. The start condition that the counter starts with $a_0 = 0$ and $b_0 = 0$ can be encoded as the Boolean term $(\overline{a_0}\overline{b_0})$. The specification that the state variables are both 1 at time step 0, 1 or 2 can be encoded as $(a_0b_0 + a_1b_1 + a_2b_2)$. The resulting BMC propositional formula is the conjunction of the starting condition, the transition relation of the counter for two time steps and the specification:

$$\begin{aligned}
(\overline{a_0}\overline{b_0})\\
(a_1 \equiv \overline{a_0})(b_1 \equiv a_0 \oplus b_0)\\
(a_2 \equiv \overline{a_1})(b_2 \equiv a_1 \oplus b_1)\\
(a_0b_0 + a_1b_1 + a_2b_2)
\end{aligned}$$

This formula can be translated into CNF and checked for satisfiability.

Chapter 3
DPLL Algorithm

A number of algorithms exist for solving SAT problems. They are divided into *complete* and *stochastic* methods: complete algorithms can always prove that a given Boolean formula is either satisfiable or unsatisfiable given enough runtime and memory, while stochastic algorithms can only find a solution for satisfiable problems, but are unable to prove that a formula is unsatisfiable. A very detailed overview of complete and stochastic SAT solving algorithms can be found in [Gu96]. Stochastic algorithms are applied when proving unsatisfiability is not required, for example for FPGA routing [Nam99], but for verification problems complete algorithms are needed.

The currently most efficient SAT solvers for industrial applications are based on extended versions of the algorithm proposed by Davis, Logemann and Loveland [Davis62] in 1962 which is usually called the *DPLL* or *DLL* algorithm. DPLL is a complete, backtracking-based algorithm that performs a depth-first search of all possible variable assignments to find a model. If no model is found, then the problem must be unsatisfiable. Later versions of the algorithm introduce learning new clauses during the search and various other improvements that substantially improve the "classic" version, especially for structured, industrial problems. A formal proof of DPLL's correctness can be found in [Marić10].

In this chapter the basic DPLL algorithm as well as the improvements that are in use in modern SAT solvers will be presented, with a focus on their implementation in MINISAT. For more details, various overview articles on modern SAT solvers exist in the literature, for example [Zhang02; Dixon04; Eén04] and [vanHarmelen07, ch. 2].

3.1 Classic DPLL

Fig. 3.1 shows the pseudocode of the DPLL algorithm in its recursive form (taken from [Marić10]). This form of the DPLL algorithm is known as the "classic" DPLL algorithm as presented in [Davis62]. The SAT problem to be solved is a CNF formula ϕ. The expression $\phi(l)$ stands for the CNF that is obtained by replacing all occurrences of the literal l in ϕ with 1 and of $\bar{l}$ with 0, then simplifying (this effectively deletes clauses which contained l from the CNF and removes $\bar{l}$ from the clauses containing it).

```
Procedure DPLL(CNF formula $\phi$)
    **if** $\phi$ is empty **return** yes.
    **else if** there is an empty clause in $\phi$ **return** no.
    **else if** there is a pure literal $l$ in $\phi$ **return** DPLL($\phi(l)$)
    **else if** there is a unit clause $\{l\}$ in $\phi$ **return** DPLL($\phi(l)$)
    **else**
        select a variable $v$ occurring in $\phi$
        **if** DPLL($\phi(v)$) = yes
            **return** yes.
        **else**
            **return** DPLL($\phi(\bar{v})$).
        end
    end
```

Figure 3.1: The DPLL algorithm in recursive form [Marić10]

If ϕ consists only of clauses that contain a 1 (ϕ is "empty"), then the CNF is satisfied and the algorithm exits after backtracking with the result "yes" (*SAT*). Otherwise if at least one clause in ϕ evaluates to 0 (the clause is "empty" because all literals in it are valued 0) then the entire formula evaluates to 0 and DPLL returns "no" (*UNSAT*). This situation is also called a *conflict*), which causes DPLL to *backtrack* to an earlier recursion level. The clause that is the cause of the conflict is called the *conflicting clause* [Zhang03].

If neither of these apply, the CNF is simplified by assigning 1 to any *pure literals* (whose opposite literal does not occur in the CNF), which effectively removes all clauses that contained the literal. When no pure literals are left, DPLL assigns 1 to all literals that are in *unit clauses* (clauses that contain only 1 literal). In real SAT solvers the pure literal rule is often not used during the search, but only once at the beginning since the effect of this rule tends to be negligible and it is costly to implement.

The simplifications result in recursive calls to the DPLL procedure with successively more simplified CNFs. When neither pure literals nor unit clauses are left in the CNF, a variable v to branch the search on is chosen (this is called making a *decision*) and DPLL is recursively called on the CNF $\phi(v)$. If this partial CNF becomes *SAT* at some point, DPLL returns *SAT*, otherwise DPLL is recursively called on the CNF $\phi(\bar{v})$ with the opposite assignment of v. This is called the *splitting rule*, since the assignments split the CNF into smaller partial problems where a variable has been replaced with either a constant 1 or 0.

The search space of a SAT problem can be represented by a binary tree, also called a *decision tree* [Silva96]. Fig. 3.2 shows an example tree for the formula $(a + b)$, which is obviously satisfiable. DPLL starts at the root node at the top of the tree,

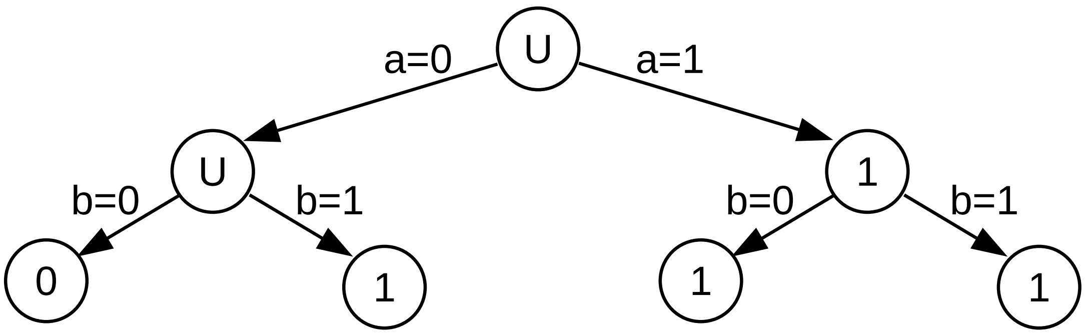


Figure 3.2: Binary tree representation of a SAT problem's search space

where no variables are so far assigned and the value of the formula is still undetermined (nodes where the formula's value is undetermined are marked with a "U" for clarification). At every node an assignment to one of the variables is made, with the search progressing down either to the left edge for 0-assignments, or down the right edge for 1-assignments. Depending on the assignment the value of the formula can remain undetermined or become 0 (conflict) or 1 (*SAT*); the node is then marked with "0" or "1".

If DPLL encounters a conflict, some assignments are taken back and the algorithm backtracks to a node further up the tree, then continues there with the opposite, untried assignment (Fig. 3.3). At the leaf nodes at the very bottom all the variables have been assigned and the formula must have a determined value. The formula is *UNSAT* if all the leaf nodes result in a 0 for the value of the formula, and it is *SAT* if there is at least one (leaf) node that has the value 1.

3.2 Conflict-driven clause learning (CDCL)

The features that make DPLL successful on industrial problems are relatively recent developments. Foremost of these is *conflict-driven clause learning* (CDCL), which was introduced in [Silva96]. The original DPLL algorithm uses *chronological backtracking*, so called because the backtrack steps always go in the strict order of the previously assigned decision variables, never "skipping" over decisions which have been tried in only one polarity so far. When the solver needs to backtrack, it looks for the last decision that has not been tried in both polarities and assigns the opposite truth value (the decision is *flipped*). In contrast, SAT solvers with CDCL use *non-chronological backtracking* which can "jump" back over several not yet flipped decision variables at once, pruning a large part of the search space, by analyzing the conflict more thoroughly. Non-chronological backtracking is intertwined with *clause learning*: the information gained from analyzing a conflict is expressed as a *learned*

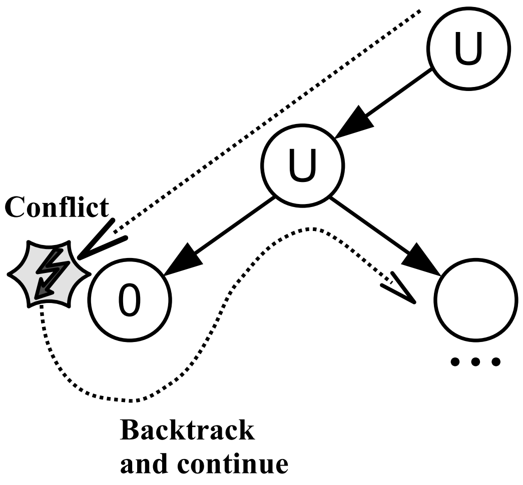


Figure 3.3: Backtracking in the DPLL algorithm

clause (also *conflict clause*) that is added to the CNF, which also serves to avoid arriving at the same conflict later. The underlying proof system of CDCL is fundamentally more powerful than chronologically backtracking DPLL, which is vital for solving many structured problems, but classic DPLL is more efficient for randomly generated problems [Beame04].

In [Silva96] DPLL was rewritten as an iterative algorithm, and the structure of the algorithm was logically partitioned into three *engines*: the *decision*, *deduction* and *diagnosis engines*. Fig. 3.4 shows the pseudocode of the iterative form of the DPLL algorithm (taken from [Zhang02]). One of the advantages of the iterative form is that it makes non-chronological backtracking easier to implement. Implementations of DPLL usually differ mostly in the details of the three engines. The decision engine's responsibility is choosing branching variables (decisions), and the deduction engine computes the implications of assigned decisions and detects resulting conflicts. If conflicts occured, the diagnosis engine analyzes them and deduces how far to backtrack.

3.2.1 Preprocessor

In addition to the engines the *preprocessor* (*preprocess()* in line 1 of Fig. 3.4) is some algorithm or set of algorithms that transform the CNF in some way beneficial to solving before the actual DPLL algorithm is entered. Examples for such techniques are simple removal of redundancies like duplicated clauses, duplicate literals in clauses and tautology clauses, but also the application of the unit clause rule and the pure literal rule to ensure that the CNF contains no obvious required assignments before the search. MINISAT 1.14 for example only removes redundancies and applies BCP,

```
1  status = preprocess();
2  if (status!=UNKNOWN)
3    return status;
4  while(1)
5  {
6    decide_next_branch();
7    while (true)
8    {
9      status = deduce();
10     if (status == CONFLICT)
11     {
12       blevel = analyze_conflict();
13       if (blevel == 0)
14         return UNSAT;
15       else
16         backtrack(blevel);
17     }
18     else if (status == SAT)
19       return SAT;
20     else
21       break;
22   }
23 }
```

Figure 3.4: Pseudocode of the DPLL algorithm in iterative form [Zhang02]

but does not use the pure literal rule.

[Lynce01] contains an overview of preprocessing techniques and investigates their effectiveness experimentally. Apart from the previously mentioned basics it investigated techniques for detecting equivalency of variables, limited forms of resolution for generating new clauses and *probing* techniques that identify necessary assignments. A surprising result of that study was that preprocessing does not necessarily reduce search time even if it significantly reduces the number of variables and infers a lot of additional clauses.

[Eén05b] introduced efficient preprocessing techniques that were shown to decrease formula sizes further and also improve solving times for industrial problems more consistently than previous algorithms. The techniques are implemented in the SATELITE preprocessor, which became a part of MINISAT following version 2.

3.2.2 Decision engine

The decision engine is implemented in the *decide_next_branch()* function (line 6 of Fig. 3.4) and chooses a currently unassigned (*free*) variable to branch on, as well as the truth value that will be assigned to it. Decision variables are given a *decision* *level* that starts with 1 for the first decision and is incremented by 1 each for every following decision. Any variables assigned during preprocessing have the decision level 0.

3.2.3 Deduction engine

After the decision the *deduce()* (line 9 of Fig. 3.4) function (deduction engine) will determine and apply all forced assignments and detects if a conflict arises from these. All implied variables that follow from a decision with the decision level n also are given the decision level n. Running the deduction engine will result in the formula either remaining undetermined or becoming *SAT* or *UNSAT*. A *SAT* result will end the algorithm, or if the formula remains undetermined another decision is made (increasing the decision level by 1). If the formula becomes *UNSAT*, a conflict has occurred and the diagnosis engine has to analyze it.

3.2.4 Diagnosis engine

If the formula became *UNSAT*, the *analyze_conflict()* function (diagnosis engine, line 12 of Fig. 3.4) will determine the reason for the *UNSAT* state, meaning it will analyze why a clause contains only 0-valued literals. This will result in a *backtrack level*, which is the decision level to which the algorithm needs to revert to resolve the current conflict. If the backtrack level is 0, it means that resolving the conflict is not possible at all and the algorithm exits with the result that the formula is *UNSAT*. On the other hand if the conflict is resolvable, the diagnosis engine will prepare some assignment that will be used for the following call of *deduce()*, and it can store some information that will avoid the same conflict in the future (usually in the form of *learned clauses* that are added to the CNF).

In the following sections some of the more popular implementations of the engines will be presented in detail.

3.3 Decision engine

The purpose of the decision engine is to choose one free variable that will be branched on as well as the truth value to assign to it. In the pseudocode of the DPLL algorithm in Fig. 3.4 it is implemented inside the *decide_next_branch()* function. The variable order should cause the DPLL algorithm to traverse as small as possible a part of the entire search space until it finds a model or proves that the CNF is *UNSAT*. Computing the optimal variable order for DPLL was proven to be both NP-hard and coNP-hard (and also to belong in the complexity class PSPACE, which is a superset of NP) [Liberatore00], therefore practical decision-making algorithms are always heuristic in nature. While the decision heuristic is an important part of the solver, it was found that for real-world SAT instances techniques for pruning the search space (like clause learning) generally have an even larger impact on solving time [Silva99].

The decision algorithm is either *static* if it uses a predetermined variable order that never changes, or it is *dynamic* if the variable order changes depending on how the search goes. Static variable orders are very slow in practice, so solvers use some heuristic that dynamically chooses branching variables depending on the current state of the CNF and/or statistics of the search progress so far. Lastly, the decision heuristic can also be a meta-heuristic that dynamically chooses among a set of decision heuristics during the course of DPLL (for example [Herbstritt04]). This section presents a number of successful (dynamic) decision heuristics that have been used in solvers over the years in roughly chronological order.

3.3.1 Random heuristic (RAND)

The RAND heuristic selects a random free variable and randomly assigns either 0 or 1 to it. Obviously this heuristic does not take any underlying structure of the CNF or any information collected during the search into account. As the simplest possible heuristic RAND can serve as a baseline to compare against when investigating the power of other heuristics.

3.3.2 Böhm's heuristic

The solver using *Böhm's heuristic* had the best performance in the 1992 SAT competition [Buro93]:

> "The idea of the heuristic used in step 4 is based on the idea of selecting a literal for assignment occuring as often as possible in the shortest clauses of the formula. Therefore, a shortest clause is either removed or reduced in size by one. Performing this step a few times, clauses of length 1 will result often, hence the formula collapses fast."

For every variable x of the CNF a vector $(H_1(x),\ H_2(x),\ ...,\ H_n(x))$ is computed according to Eq. 3.1.

$$H_i(x) = \alpha \cdot max(h_i(x), h_i(\overline{x})) + \beta \cdot min(h_i(x), h_i(\overline{x})) \tag{3.1}$$

The length n of a vector is equal to the length of the largest (*unresolved*, meaning not already satisfied) clause that contains the variable x. Each $h_i(x)$ is the number of unresolved clauses of length i that contain the literal x. The parameters α and β are weights that have to be set by the user ([Buro93] used $\alpha = 1$ and $\beta = 2$). The variable that is ultimately chosen as the decision is the one that has the largest vector according to a lexicographical sorting of all vectors.

3.3.3 MOMS heuristics

In solvers without clause learning the main objective of decision heuristics is to cause as many unit propagations per branching variable as possible, which should quickly lead to smaller subformulas and possibly a solution [Dixon04]. A popular implementation of this idea was *MOMS*, sometimes also written as *MOM's*, which stands for branching on the variable with M*axmimum* O*ccurrences in clauses of* M*inimum* S*ize* [O. Dubois93] [Freeman95] [Pretolani93] [Zabih88] (minimum size meaning 2 literals and up). Such variables are the most highly constrained ones in the CNF and are intuitively more likely to lead the search to a dead end quickly for *UNSAT* problems or to force values for yet unassigned variables for satisfiable problems [Dixon04]. There are various methods to make the choice. In [Freeman95] it is suggested to choose the variable x for which the expression $H(x)$ (Eq. 3.2) is maximal, where $w(l)$ is the number of occurences of the literal l in the smallest not yet satisfied clauses.

$$H(x) = w(\bar{x}) * w(x) * 2^k + w(\bar{x}) + w(x) \tag{3.2}$$

The SAT solver SATZ [Li97] uses $k = 10$ for example.

3.3.4 Jeroslow-Wang heuristics

The *Jeroslow-Wang (JW) heuristics* [Jeroslow90] compute the value $J(l)$ for literals l (Eq. 3.3) where ω is a clause of the CNF and $|\omega|$ is the number of literals in the clause.

$$J(l) = \sum_{l \in \omega} 2^{-|\omega|} \tag{3.3}$$

The *one-sided JW heuristic* [Jeroslow90] chooses the literal with the highest $J(l)$. The *two-sided JW heuristic* [Hooker95] selects the variable x which has the largest sum of $J(x) + J(\bar{x})$ and assigns $x = 1$ if $J(x) \geq J(\bar{x})$, otherwise $x = 0$. The motivation of the JW heuristics is that shorter clauses are considered to be exponentially better than longer ones [Giunchiglia02]. In contrast to MOMS all clauses are considered, not just the shortest ones.

3.3.5 Unit Propagation (UP) heuristics

Unit Propagation (UP) heuristics [Freeman95; Crawford96; Li96] are another approach to decision heuristics which test the impact of a variable assignment by actually assigning both 0 and 1 and counting the number of resulting implications through unit propagation. UP heuristics can be considered an "exact" variant of MOMS, but of course this approach is very costly in computation time due to the need to fully propagate assignments. A hybrid approach combining MOMS and UP was presented in [Li97].

3.3.6 DLIS/DLCS/RDLIS (GRASP)

Literal count heuristics [Silva99] count the number of times literals occur in still unresolved clauses of the current CNF. If p is a literal, let $h(p)$ be the number of times p occurs in all unresolved clauses. The *Dynamic Largest Combined Sum* (*DLCS*) heuristic will select the free variable x for which $h(x) + h(\bar{x})$ is maximal (ties are broken randomly). In other words, DLCS selects the variable occuring most often in either polarity currently. If $h(x) \geq h(\bar{x})$ then $x = 1$ will be assigned, otherwise $x = 0$. DLCS attempts to satisfy or shorten (because literals that evaluate to 0 "disappear" from the clauses) as many clauses as possible with each branch assignment.

Similar to DLCS, the *Dynamic Largest Individual Sum* (*DLIS*) heuristic chooses the *literal* rather than the variable that occurs most often in the still unresolved clauses, or in other words the literal p for which $h(p)$ is maximal (ties can be broken randomly). If the literal p is positive, then the variable of p ($var(p)$) will be assigned $var(p) = 1$, otherwise $var(p) = 0$. DLIS attempts to satisfy as many clauses as possible with one branch assignment and to shorten as many clauses as possible with the opposite branch assignment. DLIS is the default heuristic in the ground-breaking SAT solver GRASP [Silva96], which introduced clause learning and conflict-driven non-chronological backtracking.

RDLIS is a variant of DLIS that randomly chooses if a positive or negative assignment is made to the selected variable first, rather than letting the respective number of literals decide the polarity. This is because DLIS is sometimes too greedy, which leads to worse performance. A similar *RDLCS* variant of DLCS can be made the same way.

3.3.7 VSIDS (CHAFF)

The SAT solver CHAFF [Moskewicz01] was a major step in the evolution of SAT solvers due to its use of both watched literals for BCP (deduction engine) and the light-weight but efficient decision heuristic VSIDS. Unlike GRASP'S DLIS the VSIDS heuristic does not keep track of variable or literal counts in the current CNF, which is quite expensive, but rather only keeps a "score" (also called *activity*) for each literal (positive and negative) that is increased when a new clause is learned that contains the literal. The algorithm works as follows:

1. Before the search starts, initialize all activities with the numbers of the respective literal in the original CNF.

2. When a new learned clause is generated, increase the activity of all literals that occur in the clause.

3. When a decision is needed, choose the unassigned literal with the highest activity (ties are broken randomly).

4. Periodically divide all activities by a constant.

The advantage of VSIDS is its high performance while requiring low computation overhead.

3.3.8 MINISAT heuristic

MINISAT [Eén04] uses a slightly modified variant of VSIDS. The MINISAT heuristic (MSH), as implemented in version 1.14 of the solver, is the target for optimization in this work and will therefore be explained in detail here. In MHS each *variable* rather than each literal as in VSIDS has an activity associated with it. Rather than dividing the activities by a constant periodically as in VSIDS, the increment value for the activities rises itself: every time a new clause is learned, the activity of each variable occuring in the clause is increased by a value (“bumped”) which is itself increased multiplicatively afterwards. The algorithm of the decision heuristic is as follows:

1. The activities are stored in a vector of double precision floating-point numbers named `activity[N]` and are all 0 before the search starts (N is the number of variables).

2. When a decision has to be made, the variable with the currently highest activity is chosen (ties are broken randomly). Decision variables are always assigned the truth value *false*.

3. When a new learned clause is generated, increase the activity of all variables that occur in the clause by the current *variable incremement value* `var_inc`, which has the starting value 1. Afterwards, `var_inc` is increased itself by multiplying it with the constant VARDECAY (default value is $\frac{1}{0.95} \approx 1.053$). It is also checked if any activity exceeds 10^{100}: if yes, all activities and also `var_inc` are multiplied with 10^{-100}.

4. With a small probability defined by the constant $\mathrm{RANDOMVARFREQ}$ (default value 0.02 (2 %)), MINISAT sometimes chooses a random variable; this has been found to help solving some problems without causing too much overhead for other problems [Eén04].

The parameter $\mathrm{VARDECAY}$ is actually given in inverted form to the solver and is re-inverted for use for the search (since it must be value > 1 to serve as an increasing, multiplicative factor). The effective range for `var_decay` is then $(0, 1]$, with the "aging" effect becoming more pronounced the closer to 0 the value gets, but negative values for $\mathrm{VARDECAY}$ can be used to enforce a static variable ordering in MINISAT. The highest sensible value is 1, since for values greater than 1 the increase value `var_inc` would become smaller after every learned clause.

The solver parameters $\mathrm{VARDECAY}$ and $\mathrm{RANDOMVARFREQ}$ were targets for optimization with EA in this work.

3.3.9 Progress saving

It was found in [Pipatsrisawat07] that standard VSIDS-style decision heuristics do not deal efficiently with SAT problems that consist of several independent components, meaning problems where the CNF can be partitioned into sets of clauses that do not have variables in common. Non-chronological backtracking leads to already assigned variables being unassigned, undoing the progress achieved in one partial problem which has to be redone afterwards, possibly multiple times. To remedy this loss of information [Pipatsrisawat07] introduces an efficient *component caching* technique called *progress saving* for the decision heuristic: every time the solver backtracks and erases an assignment, the last assigned value is saved in a *saved-literal array*. When a decision is later made for a variable whose value has been saved, the saved assignment is used. Progress saving is easy to implement, has low overhead and was shown to be effective for real-world SAT instances.

3.4 Deduction engine

The deduction engine (*deduce()* function in line 9 of Fig. 3.4) infers forced variable assignments due to decisions to avoid unnecessary assignments, and also detects conflicts resulting from the assignments. Iteratively detecting and assigning implications due to unit literal clauses is the simplest and also a very effective method of deduction, therefore all DPLL SAT solvers perform this operation. It is known that in most cases greater than 90% of the computation time in DPLL is spent propagating chains of implications [Moskewicz01], making a powerful BCP (Boolean constraint propagation) engine vital for the efficiency of the solver.

Other deduction mechanisms have been implemented (for example the pure literal rule and *equivalency reasoning* [Li00]), but are in practice only effective for some problem classes while making performance worse in the general case [Moskewicz01].

SAT solvers usually store the clauses of the CNF as lists of literals in memory. For example, MINISAT represents variables as integer values, and literals as integers where the least significant bit indicates the polarity of the literal and the rest of the bits contain the variable's number. Clauses are represented as an array of literals, with an additional integer that stores the size of the clause and in which the least significant bit indicates if the clause is learned or it was in the original CNF.

After assigning a variable value the solver must detect if any clause now contains only one unassigned literal and the rest are all 0-valued (then it is a unit clause which creates an implication), and also when any clause contains only 0-valued literals (then a conflict has occurred). Clauses that contain at least one 1-valued literal are satisfied and have no consequences for BCP, and neither do clauses containing at least two unassigned literals and no 1-valued literals. Various methods to detect implications and conflicts have been implemented.

3.4.1 Counter-based BCP

A simple method for detecting clauses becoming unit or conflicting is to use counters [Crawford96]. For example, the solver GRASP [Silva96] uses two counters per clause, one for the number of 0-valued literals and one for the 1-valued literals in it. Every variable has two lists associated with it that point at the clauses that contain its positive or negative literal. When a variable is assigned a value, the respective counters in the clauses indicated in the lists are updated accordingly. When a clause becomes unit, its 0-literals counter is exactly one less than the clause size and for a conflict clause the counter is equal to the clause size. While this scheme is easy to understand and implement, a major drawback is that the number of necessary updates rises roughly linearly with the number and size of the clauses [Zhang02]. The updates must be performed when assigning a variable as well as when backtracking, to revert the counters. Moreover, clause-learning solvers tend to add a large amount of relatively long clauses during solving, making this problem even worse.

3.4.2 2-literal watching

The counter scheme for BCP is inefficient because it visits every clause (with all the memory, cache and CPU operations that implies) that contains a literal that has just been assigned, even though many of these clauses will likely not generate implications or become conflicting. Ideally a BCP mechanism would only access clauses where an implication or a conflict are imminent. Satisfied clauses and not yet satisfied clauses that contain more than one unassigned literal have no consequence for BCP and should not be visited at all. A scheme known as *head/tail lists* that was introduced in the SAT solver SATO [Zhang96] goes towards this ideal because it requires no operations on clauses that contain literals evaluating to 1 after assigning. The underying concept is further improved in the solver CHAFF [Moskewicz01], which implements a BCP mechanism called *2-literal watching*.

In the scheme, as the name implies, 2 unassigned literals in every clause are "watched" and the clause is only inspected when 0 is assigned to either of them. Every variable has two *watch lists* that contain pointers to clauses that contain either the positive or the negative literal. Initially, any two (unassigned) literals in each clause of the CNF are picked and entered into the respective watch lists. When, for example, 1 is assigned to the variable v, the negative literals $\bar{v}$ now all evaluate to 0; the respective watch list of variable v that contains pointers to watched occurrences of $\bar{v}$ is iterated and all clauses in it are inspected. Four different situations are possible:

1. All literals in clause C are 0: exit with result *UNSAT*.
2. At least one in clause C literal is 1: continue with next clause.
3. C contains exactly one unassigned literal l and the rest are all 0: found the implication l, continue with next clause.
4. There is at least one unassigned literal l in the clause that is not the other watched literal: remove the watch on literal $\bar{v}$ and create a new watch in the respective watch list of l.

The 2-literal watching scheme requires much fewer clause inspections on assignments than the counting scheme. Additionally, when backtracking the watches can simply be left in place, requiring no operations at all.

3.4.2.1 Binary clauses

MINISAT also uses an implementation of the 2-literal watching scheme, and additionally (starting with version 1.13) handles binary clauses in a special way. Since binary clauses consist only of two literals, an assignment of 0 to either of the literals will result in an implication for the other literal (or a conflict) immediately. It is possible and more efficient to store the respective “other” literal directly in the watch lists of the relevant variables. This avoids the overhead associated with having to create and manage clause data structures in heap memory and improves performance because there are no clauses that might need to be loaded from memory into the CPU cache on inspection. Industrial SAT problems tend to have a large number of binary clauses, making this relatively simple measure particularly effective.

3.4.2.2 Blocking literals

It was noticed by several authors [Jain07; T. Schubert07] that for industrial problems watched clauses are often already satisfied. Accessing the clause in memory is likely to cause a cache miss, therefore some solvers (MINISAT in version 2.2) implement so-called *blocking literals* in the watch lists. Every pointer to a watched clause is paired with a blocking literal from that clause, which can be immediately checked if it is satisfied or not. If yes, the clause can be skipped, avoiding a costly memory access.

3.5 Diagnosis engine

The purpose of the diagnosis engine (*analyze_conflict()* function in line 12 of Fig. 3.4) is analyzing the reasons for a conflict and computing a backtrack level.

3.5.1 Chronological backtracking

When a conflict has occured, it means that the current subtree (for example Fig. 3.2 after assigning $a = 0$ and $b = 0$) of the search space can not contain a satisfying set of variable assignments and the algorithm has to back up to a previous node to explore a different subtree. The simplest way to deal with conflicts is to backtrack to the last tree node for which both assignments have not been tried yet; this is *chronological* backtracking and was used in the original DPLL algorithm. It was implemented by storing a Boolean flag for each variable that indicates if the variable was already tried in both polarities (*flipped*) or not. Chronological backtracking is competitive on randomly generated SAT problems [Zhang01], but on more structured instances like transformed EDA problems it is inefficient.

3.5.2 Non-chronological backtracking

With *non-chronological* backtracking the DPLL algorithm can jump back over several decision levels (Fig. 3.5) rather than simply to the last decision variable that was not yet tried in both polarities. This feature, which has its origins in the *Constraint Satisfaction Problem* (CSP) domain [Prosser93], was first used in in the SAT solvers GRASP [Silva96] and RELSAT [Bayardo97].

To avoid missing a possible *SAT* solution in the part of the search space that is jumped over, the diagnosis engine must determine exactly which of the previous decisions were involved in the conflict. For example [Zhang03, pg. 40], if the CNF contains these 4 clauses:

$$(\bar{a} + \bar{b} + c)(\bar{a} + \bar{b} + \bar{c})(\bar{a} + b + c)(\bar{a} + b + \bar{c})$$

and the solver sets variable $a = 1$ on decision level 3, then the remaining clauses:

$$(\bar{b} + c)(\bar{b} + \bar{c})(b + c)(b + \bar{c})$$

can not be satisfied by any combination of assignments to variables b and c. The solver will not notice this though until it actually assigns b or c (assigning one will directly generate an implication for the other, followed by a conflict). The solver continues for example with variables x and y on decision levels 4 and 5. If b or c are then assigned at decision level 6, then the solver will immediately encounter conflicts for any values of b and c. With chronological backtracking the solver then would return to decision level 5 and flip the variable that was the decision there.

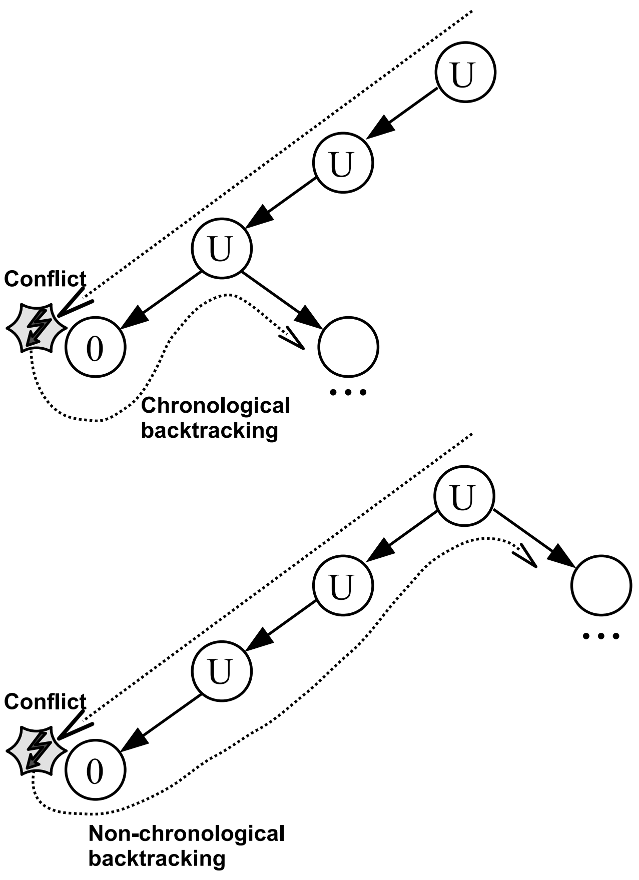
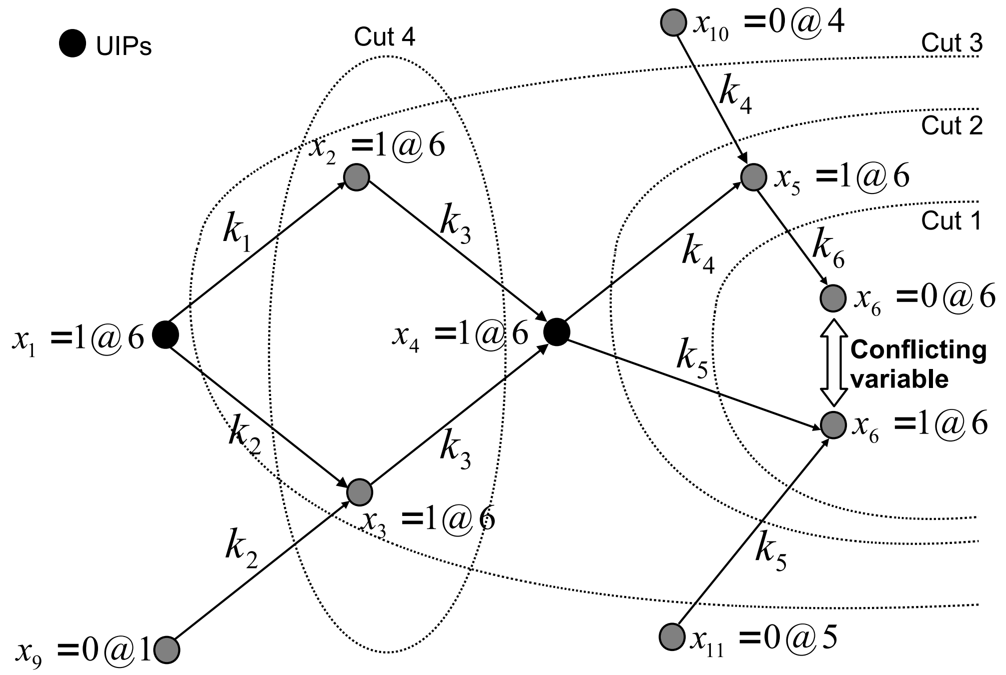


Figure 3.5: Chronological vs. Non-chronological backtracking

However, following the decision $a = 1$ at decision level 3 the CNF can never be satisfied no matter what assignments are made to any other variables, and the solver uselessly tries out assignments on decision levels 4 and 5 until eventually it can return to decision level 3 and flip a. A solver with non-chronological backtracking would have analyzed the conflict at decision level 6 and found that it can only be resolved by going back to decision level 3 directly.

3.5.2.1 Implication graphs

The process of conflict analysis can be visualized using an *implication graph* (IG); Fig. 3.6 shows an example IG [Silva96] (clause k_6 was the conflict clause). An IG is a *directed acyclic graph* (DAG) where each vertex is labeled with a variable assignment (and its decision level, given after the @) made during the search. The incident edges of each vertex (which stands for a variable assignment) connects it to the vertices/assignments that were the *reason* for its assignment, and the edge is labeled with the clause that caused the implication (its *antecendent clause*). From this follows that decision variables have no incident edges. Fig. 3.6 shows the clauses k_1 to k_9 of the CNF relevant for the given IG; it can be seen that for example the assignment $x_3 = 1$ was an implication following from clause k_2 due to assigning $x_1 = 1$ and $x_9 = 0$. When a conflict has occurred, the IG contains two opposite assignment vertices for the same variable (the *conflicting variable*); in Fig. 3.6 this is variable x_6.



$$f = \underbrace{(\overline{x_1} + x_2)}_{k_1} \cdot \underbrace{(\overline{x_1} + x_3 + x_9)}_{k_2} \cdot \underbrace{(\overline{x_2} + \overline{x_3} + x_4)}_{k_3} \cdot \underbrace{(\overline{x_4} + x_5 + x_{10})}_{k_4} \cdot \underbrace{(\overline{x_4} + x_6 + x_{11})}_{k_5}$$

$$\cdot \underbrace{(\overline{x_5} + \overline{x_6})}_{k_6} \cdot \underbrace{(x_1 + x_7 + \overline{x_{12}})}_{k_7} \cdot \underbrace{(x_1 + x_8)}_{k_8} \cdot \underbrace{(\overline{x_7} + \overline{x_8} + \overline{x_{13}})}_{k_9} \ldots$$

Figure 3.6: Implication graph

A new clause can then be generated from the IG by making a *cut*; the cut bipartitions the IG so that one side contains the conflicting variable (both its vertices). The side containing the conflicting variable is called the *conflict side*, the other is the *reason side*. Fig. 3.6 shows 3 such cuts. The vertices on the reason side whose edges cross the cut line signify assignments that will always lead to a conflict. For example, cut 1 corresponds to the assignments $x_5 = 1$, $x_4 = 1$ and $x_{11} = 0$, which must not occur together. Using DeMorgan's law this condition can be turned into a clause:

$$\overline{x_5 \cdot x_4 \cdot \overline{x_{11}}} = (\overline{x_5} + \overline{x_4} + x_{11})$$

The new clause is redundant to the original CNF and can be added to it without changing its satisfiability. This is also true of the clauses corresponding to the other two cuts: cut 2 results in the clause $(x_{10} + \overline{x_4} + x_{11})$ and cut 3 in $(x_{10} + \overline{x_1} + x_9 + x_{11})$, both of which could be added to the CNF. It is furthermore possible to extract clauses from reconverging parts of the IG that do not involve the conflict directly; this is called a *cut not involving conflicts* [Zhang01]. In Fig. 3.6 cut 4 is one such cut, from which the clause $(\overline{x_1} + x_9 + x_4)$ could be learned (because the IG shows that when $x_1 = 1$ and $x_9 = 0$ are assigned, the implication $x_4 = 1$ follows).

The process of generating learned clauses from an IG can alternatively be understood as a series of resolution operations (see Sec. 2.2.1.3) on the clauses involved in the conflict [Zhang02]. Replacing a literal in the conflict clause with its reason assignments is equivalent to performing the resolution operation on the conflict clause with the respective antecedent clause, with the reason variable as the pivot.

Indiscriminately adding large numbers of clauses to the CNF generated from the IG is not useful, because it slows down the BCP engine. Instead, clauses should be chosen that are likely to aid the further search. Which clauses to extract and learn from the IG is the objective of the solver's *learning scheme*. It was shown that solving times differ greatly depending on the learning scheme used [Zhang01]. For non-chronological backtracking, the clause also has to have the property of being *asserting*. A clause generated by some cut that contains exactly one literal assigned at the current decision level and where all the other literals are assigned at lower decision levels is asserting; after backtracking to the second-highest decision level in the clause all literals except the one at the highest decision level will have the value 0, meaning that this clause becomes a unit literal clause that causes an implication for the remaining literal (it is asserted). Such an asserting clause can always be constructed.

Unique Implication Points (UIPs) are defined as such [Zhang01]:

> "In an implication graph, vertex a is said to dominate vertex b iff any path from the decision variable of the decision level of a to b needs to go through a. A Unique Implication Point (UIP) [Marques-Silva99] is a vertex at the current decision level that dominates both vertices corresponding to the conflicting variable. ... The decision variable is always a UIP. Note that there may be more than one UIP for a certain conflict. ... Intuitively, a UIP is the *single* reason that implies the conflict at the current decision level. We will order the UIPs starting from the conflict."

In Fig. 3.6 there are 2 UIPs, x_1 and x_4 on the current decision level 6. Asserting clauses can be constructed by making sure that a UIP is on the boundary of the cut [Zhang01]:

> "To make a conflict clause an asserting clause, the partition needs to have one UIP of the current decision level on the reason side, and all vertices assigned after this UIP on the conflict side. Thus, after backtracking, the UIP vertex will become a unit literal, and make the clause an asserting clause."

Actual SAT solvers do not explicitely build an IG on encountering a conflict, but rather store a pointer to the antecedent clause of each implication during BCP. The IG can then be traversed by following the pointers.

3.5.2.2 Learning schemes

The solver's *learning scheme* determines which and how many clauses to learn from a conflict.

The solver RELSAT [Bayardo97], which was one of the first SAT solvers using learning and non-chronological backtracking, makes the cut in the IG such that all variables of the current decision level except the decision variable are on the conflict side, and all variables assigned at earlier levels and the current decision variable are on the reason side. In the example in Fig. 3.6, the cut chosen by the *relsat scheme* corresponds to cut 3.

In the solver GRASP [Silva96] the learning scheme tries to learn as much about the conflict as possible. It adds all clauses resulting from the IG reconverging on UIPs (cuts not involving conflicts), and in addition a cut is made that puts the variables assigned on the current decision level after the first UIP (the closest to the conflict vertices) on the conflict side, and everything else on the reason side. This results in an asserting clause. In the example in Fig. 3.6, this *first UIP cut* corresponds to cut 2.

Adding only the clause resulting from the first UIP cut is called the first UIP learning scheme (also known as *1 UIP scheme*); it was experimentally shown to be superior [Zhang01] to the relsat and GRASP schemes (which learns more clauses in addition to the 1 UIP clause). The CHAFF solver uses the 1 UIP scheme.

MINISAT also uses the 1 UIP scheme and gives a simple description of the algorithm that computes the learned clause [Eén04]:

> "In a breadth-first manner, continue to expand literals of the current decision level, until there is just one left."

This algorithm starts with the conflict clause, in which all literals are valued 0, and expands (replaces with the reasons for the assignment) literals in a loop until the clause contains only one literal that is on the current decision level.

3.5.2.3 Conflict clause minimization

Conflict clause minimization attempts to eliminate superfluous literals from the newly generated learned clause, making it a stronger constraint (the conflict clause is *strengthened*). It has become a standard feature in many solvers and is present in, among others, MINISAT in all versions following 1.13 [Eén05a]. In MINISAT, after the learning scheme generates the 1 UIP clause, the solver attempts to remove literals from it by using the resolution operation (Sec. 2.2.1.3) on it with antecedent clauses from the IG. For example, if the clause $(a + b + c)$ is the 1 UIP clause and the antecedent of the assignment $a = 0$ was the clause $(\overline{a} + b)$, then the resolution of these clauses with a as the pivot variable results in the clause $(b + c)$, which is a stronger constraint than the original clause $(a + b + c)$: clause $(a + b + c)$ is *self-subsumed* by $(\overline{a} + b)$ w.r.t. a. Conflict clause minimization dramatically reduces memory usage (about half that of without minimization) and also reduces the search space, leading to faster solving times [Sörensson09].

3.5.2.4 Assignment stack shrinking

Assignment stack shrinking [Nadel10] is a technique that modifies the backtracking behavior of a CDCL SAT solver under certain circumstances; it was introduced in the solver JERUSAT in 2002. If a *shrinking condition* is satisfied, the solver applies a *sorting scheme* to the conflict clause literals and then computes and backtracks to a *shrinking backtrack level*. For Jerusat these are:

- Shrinking condition: "conflict clause contains no more than one variable from each decision level"
- Sorting scheme: "sort literals by decision level from lowest to highest"
- Shrinking backtrack level: "The shrinking backtrack level for Jerusat is the highest possible decision level where all the literals of the conflict clause become unassigned. Jerusat then guides the decision heuristic to select the literals of the conflict clause according to the sorted order and assign them the value false, whenever possible."

The effect of assignment stack shrinking is to make assignments more relevant to the current area of the search space. Irrelevant variables are unassigned and the decision heuristic is steered toward assigning variables that occur in the recently learned clauses. This leads to shorter learned clauses and faster solving times.

3.6 Restarts

Complete search methods for combinatorial problems (like DPLL SAT solvers) often display wildly different solving times for nearly identical problems, for example simply changing the numbering of the variables in a SAT problem can make a formerly easy problem suddenly unsolvable. This phenomenon also appears in search algorithms of different problem domains, for example CSP. In [Gomes98] it was found:

> "Unpredictability in the running time of complete search procedures can often be explained by the phenomenon of "heavy-tailed cost distributions", meaning that at any time during the experiment there is a non-negligible probability of hitting a problem that requires exponentially more time to solve than any that has been encountered before."

When such a distribution is present, it was found that randomization techniques can be used to stabilize solving times somewhat or even make it possible to solve problems that were impossible before [Gomes98].

These techniques are the introduction of randomization into the decision step of the backtrack search algorithm, and *restarts*: the solver is stopped when it has spent too much time in some area of the search space (which is likely very hard and will not yield a solution quickly) and forced to restart the search from the beginning (taking back all decisions assgigned so far), which may lead it to a different search space section where the solution is easier to find. Restarts have become an important standard feature in DPLL SAT solvers, including CHAFF and MINISAT.

There are various techniques in use to schedule the restarts (as described in [Ryvchin08]):

1. *Arithmetic series*: restart after x conflicts, increase this by y after every restart (used for example in CHAFF with $x = 700$ and $y = 0$).

2. *Geometric series*: restart after x conflicts, multiply this by y after every restart (used in MINISAT prior to version 2).

3. *Inner-Outer Geometric series*: a nested loop creates an oscillating series of restarts with geometrically increasing number of conflicts (used in the solver PICOSAT).

4. *Luby series*: [Luby93] proved that in the absence of any knowledge about the search space *Las Vegas algorithms* (these always produces the correct answer eventually, but with random runtime) have optimal runtime if they are restarted according to the *Luby-sequence* $1, 1, 2, 1, 1, 2, 4, \ldots$ (in words: "One way to describe this strategy is to say that all run lengths are powers of two, and that each time a pair of runs of a given length has been completed, a run of twice that length is immediately executed" [Luby93]). This was empirically shown

to be effective for DPLL restarts in [Huang07], though the Luby numbers are multiplied with a constant factor since a limit of only 1 or 2 conflicts would be too low to be efficient (used in MINISAT 2.2).

The restart algorithm in MINISAT 1.14 is the relatively simple geometric series (Fig. 3.7). The *search()* function that executes the actual DPLL search has a parameter `nof_conflicts` that sets the maximum number of conflicts that are allowed before it stops, takes back all decisions and returns the solver to decision level 0, then returns to an outer search loop. Initially `nof_conflicts` is 100 (conflicts), and every time *search()* returns to the outer loop its value is multiplied with a constant. The constant's default value is 1.5, which is placed directly in the source code. Hereafter, the initialization value of `nof_conflicts` will be referred to as NOFCONFLICTSBASE, and the incremental constant as NOFCONFLICTSINC. These two parameters were among the objects of optimization for the evolutionary algorithms in this work.

```
NOFCONFLICTSBASE = 100; // default values
NOFCONFLICTSINC = 1.5;

nof_conflicts = NOFCONFLICTSBASE;
while (status == undefined) {
  status = search(..., nof_conflicts, ...);
  nof_conflicts *= NOFCONFLICTSINC;
}
```

Figure 3.7: MINISAT 1.14 restart strategy

3.6.1 Learned clause removal

Clause-learning SAT solvers generate a large amount of learned clauses over the course of the search. After a restart, the previously learned clauses are still there and help the solver avoid already explored areas of the search space. Unfortunately the learned clauses also slow down the BCP engine more and more unless at least some of them are cleared away regularly. Determining which learned clauses to remove is the purpose of a learned clause removal heuristic.

In MINISAT 1.14 the lifetime of a learned clause is governed by its *clause activity*, a single-precision floating point value similar to the variable activities that control decisions. The clause activity is attached to learned clauses (except binary clauses, which are stored directly in the watch lists and therefore have no activity and consequently never get deleted). A clause's activity rises when it is touched during conflict analysis (while traversing the IG when generating the 1 UIP clause). To simulate an "aging" of the clause activities the "bump" value that increases the activity is incremented over time, similar to the way variable activities age (see Sec. 3.3.8). Periodically learned clauses whose activity is below a limit are removed.

The solver variable `cla_inc` contains the bump value with which the clause activities are incremented; its starting value is 1. After every conflict the new learned clause's activity is set to the current value of `cla_inc`. Afterwards `cla_inc` is multiplied with the solver parameter constant CLAUSEDECAY, which is larger than (or equal to) 1, though the value is given as the inverse to the solver. The default value of CLAUSEDECAY is $\frac{1}{0.999} \approx 1.001$. If a learned clause is used during conflict analysis, its clause activity is also bumped up (`cla_inc` is added to its current value). If the new value of a clause activity exceeds 10^{20}, the clause activities of all learned clauses as well as `cla_inc` itself is multiplied with 10^{-20}.

The *search()* function (see Fig. 3.7) that executes the actual DPLL search also has a parameter `nof_learnts` that sets the limit of learned clauses that are allowed for this search run. The initial value of `nof_learnts` is the number of original CNF clauses divided by 3. This division factor will hereafter be referred to as NOFLEARNTSDIVISOR and is a target of optimization for the EA. After every restart `nof_learnts` is multiplied with the solver parameter constant NOFLEARNTSINC, so that the number of learned clauses allowed steadily rises. Inside *search()*, if the current number of learned clauses exceeds the limit set by `nof_learnts` a cleanup function is called. The cleanup function always deletes half of all currently learned clauses (unless they are “locked”, meaning they are reason clauses for some assignment) and furthermore all clauses whose activity is below `cla_inc` divided by the current number of learned clauses.

The parameters NOFLEARNTSDIVISOR, NOFLEARNTSINC and CLAUSEDECAY were objects of optimization for the EA in this work.

Chapter 4

Genetic Algorithms

Genetic Algorithms (GAs) are one of the four main sub-fields of Evolutionary Algorithms. In GAs, the candidate solutions are always represented as strings of bits (binary strings). How to interpret the strings as solutions on the problem. EAs that use binary string encodings are usually called "classic GAs" today. The foremost early practicioner of GAs is John Holland, who wrote his groundbreaking work "Adaptation in natural and artificial systems" in 1975 [Holland92]. Holland also provided some theoretical foundations on why GAs work (known as *schema theory*).

The implementation of a basic GA is very simple and can be done in a few hundred lines of code. There are a great number of variations of the classic GA procedure, for example [Goldberg89] describes the *Simple Genetic Algorithm* (SGA) and [Coley98] the *Little Genetic Algorithm* (LGA), which are basically equivalent. The GA implementation used in this work is based on SGA/LGA. The SGA procedure works roughly as follows (details of the operators will be described in the following):

1. Generate the initial population of bit strings.

2. Decode all individuals into their phenotype.

3. Compute the objective function value for each decoded individual.

4. Fitness scaling: transform the objective function value to a fitness value (the fitness and objective function value can also be the same).

5. Use the *selection operator* to choose pairs of individuals to be parents.

6. Depending on a *crossover probability* P_c either apply the *crossover operator* (a type of recombination) to create two new offspring from a parent pair, or copy the parents unchanged to the temporary new population. Repeat selection and crossover/copying until the temporary population is the same size as the original.

7. Apply the *mutation operator* on every individual in the temporary population. A bit in an individual has a *mutation probability* P_m to change to its opposite value.

8. Delete the old population and replace it with the temporary one.
9. Increment the generation counter. If the maximum number of generations has been reached then exit, otherwise continue at step 2.

4.1 Encoding numbers as binary strings

In a numerical optimization problem the solutions are vectors of numbers. To solve such a problem with a GA, the vector has to be encoded as a binary string. Usually this is achieved by encoding each variable as one section of the whole string (Fig. 4.1 shows an example for a problem with 2 variables encoded as 4 bits each). How many bits are needed for each section depends on the allowed range of the corresponding variable (and in the case of real-valued variables also on the precision which the result should have). A section consisting of l bits can represent 2^l different values.

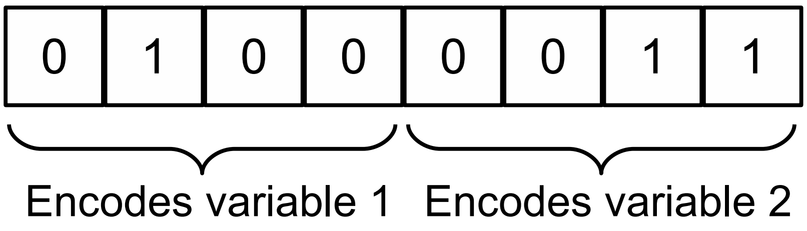


Figure 4.1: For numerical optimization problems the binary string encoding a GA individual is partitioned into sections that encode one object variable each

4.1.1 Integers

If the variable of the optimization problem is an integer value with the range $r_{\min} \leq x \leq r_{\max}$ ($r_{\min} < r_{\max}$), then the length l of the bit vector has to be at least $l = \lceil log_2(r_{\max} - r_{\min} + 1)\rceil$. Each binary code is mapped to one integer value. Assuming the standard binary encoding ("1" = 1, "10" = 2, "11" = 3 etc.) the code for the lower limit $r_{\min}$ is then "00...0", $r_{\min} + 1$ is represented by "00...1" and so on.

For example, to encode an integer variable with the range $[0, 1000]$ a section at least $\lceil log_2(1000 - 0 + 1)\rceil = \lceil log_2(1001)\rceil \approx \lceil 9.96\rceil = 10$ bits long is needed. A string of length 10 can encode 1024 values, meaning that apart from the 1001 valid integers in the range $[0, 1000]$ there are also 23 superfluous binary codes that do not correspond to an allowed value. Since the genetic operators are blind as far as the validity of the resulting bit string is concerned, the GA has to deal with individuals that encode invalid solutions in some way. This can be achieved by imposing fitness penalties on illegal individuals, or by ensuring that individuals are always decoded into legal phenotypes [Davis87]. In either case there are unfortunately drawbacks, like lost computation time and wasted population space for illegal individuals, or the extra effort required to build advanced decoding algorithms. For the example treating every decoded value over 1000 as the value 1000 is a simple way to enforce legality, but for different optimization applications with non-numerical domains this problem is harder to deal with.

4.1.2 Real values

If the domain of a variable is real numbers in the range $r_{\min} \leq x \leq r_{\max}$ ($r_{\min} < r_{\min}$), the most common representation is a linear mapping between the binary codes and the real numbers in the range $[r_{\min}, r_{\max}]$ [Coley98, pg. 19].

The (partial) binary string of length l is first translated into a base-10 integer value z between 0 and the maximum value $2^l - 1$. The code for $z = 0$ translates to $r_{\min}$ and the code for $z = 2^l - 1$ translates to $r_{\max}$. The granularity with which values between the two bounds can be described depends on the string length, which must be adjusted accordingly. The variable value r can be computed as in Eq. 4.1 [Coley98, pg. 21].

$$r = \frac{r_{\max} - r_{\min}}{2^l - 1} z + r_{\min} \tag{4.1}$$

4.1.3 Gray coding

The format that is used to represent the values is also not irrelevant and influences the behavior of the GA. The standard binary format for numbers is just one possible way to encode decimals as binary strings, but there are many more permutations to translate the codes into numbers. One of the reasons that the standard format may not be the ideal representation is that two very similar solutions of the problem may have substantially different encodings. For example, if the solution of a numerical optimization problem is the decimal number 128 and the encoding is 8 bit unsigned integers in the standard format between 0 and 255, the optimal solution will be the string $10000000_2 = 128_{10}$. Decoded values that are close to, but lower than 128, for example $127_{10} = 01111111_2$, have very different encodings (in this case even the bit-wise negation of the optimum), so that changing one such suboptimal solution into the optimal one would take either many mutation operations over several generations or several mutations all at once, with either event having a low probability. This phenomenon is called a *Hamming cliff* [Goldberg89], because there is a large *Hamming distance* (number of differing bits) between the genotypes of two very close phenotypes. Intuitively, similar solutions should have similar encodings so that small changes in the encoding space are equivalent to small changes in the solution space, making it more likely that a random small mutation will bring a near-optimum solution closer to the optimum rather than transforming it into one that is far from the optimum. Such an encoding should improve a GA's ability to pinpoint an optimum.

One way to remedy Hamming cliffs while still using bit string encoding is to use *Gray coding* (named after its inventor, American physicist Frank Gray). Gray code represents numbers in such a way that each subsequent number's bit string encoding differs in exactly 1 bit (Tab. 4.1 shows the first 8 Gray codes). Gray coding can be used both for integer and real-valued variables. It should be noted that while a single-bit mutation in Gray code may change the decimal value by only 1, it can also change it by a different amount depending on the position of the bit being changed, including one that is larger than for the equivalent binary representation.

Decimal	Binary	Gray code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Table 4.1: Gray codes

4.2 Initial population

The initial population contains randomly generated individuals. Tab. 4.2 shows a sample initial population assuming a population size $N = 8$ and a binary string length of $l = 12$. The strings were generated by setting the bits of each string to 0 or 1 with a 50% probability.

The univariate objective function for the example is $F(x) = x^2$, which is to be maximized (the fitness is assumed to be equal to the objective function). The encoding is standard binary encoding with a variable range of $x \in \{0, 1, \ldots, 4095\}$. The optimal solution for the example objective function using the 12 bit binary string as genotype would be "111111111111", which is the encoding of the highest number (4095) in the search space. The best solution of the initial population is number 4, with $x = 3161$ and $F(x) = 9991921$. The worst solution is number 6 with $x = 163$ and $F(x) = 26569$. It should be noted that although solution 6 is very bad in quality (low fitness), it is also the only string in the population that has a 1 in the second position from the right. If this string were discarded early on, and the GA had no mechanism to generate another string with a 1 in the second position from the right, the optimal solution "111111111111" could never be found due to the missing 1.

Individual Nr.	String	Decoded value	Fitness
1	100100011100	2332	5438224
2	000110111000	440	193600
3	100101110101	2421	5861241
4	110001011001	3161	9991921
5	101011110001	2801	7845601
6	000010100011	163	26569
7	011001000000	1600	2560000
8	011101100101	1893	3583449

Table 4.2: Sample initial GA population, $N = 8$, $l = 12$, objective function = fitness $F(x) = x^2$

This emphasizes the necessity of the mutation operator, which can regenerate lost (or never previously present) genetic information. In this case the mutation operator could regenerate the 1 in the second position from the right in some other individual. In contrast, crossover (a recombination operator) can not replace such lost information because it does not create new values but only swaps parts of parent strings.

4.3 Fitness scaling

In GAs, the fitness of individuals is always assumed to be a non-negative number and the higher the fitness, the "better" (more likely to create offspring) the individual is considered to be. For a minimization problem lower objective values are better, therefore some transformation is necessary that generates the highest fitness values for the lowest objective values. Even for maximization problems where the objective function is known to be non-negative it can be beneficial to apply a *fitness scaling* rather than just using the objective value as the fitness.

For example, in the first generation of a GA run there are often a few highly fit "superindividuals" and a lot of fairly bad ones. Without scaling, the few good individuals are likely to take up most of the space in the next generation, forcing out all the weak individuals that may possibly contain important genetic information. The genetically impoverished new population is then stuck at the solutions encoded by the superindividuals and can not move away easily. This phenomenon is known as *premature convergence* to a local optimum and is a general problem of optimization algorithms [Michalewicz94, pg. 57]. On the other hand, towards the end of the GA run the population likely contains individuals that are all fairly similar to each other and have similar fitness values. Then it is hard for the selection operator to discern between good and bad individuals and the GA progresses very slowly, if at all, towards the optimum. To improve performance, fitness scaling would decrease the differences in fitness at the beginning of the GA and increase the differences later when the population contains mostly similar individuals.

Unfortunately many different scaling algorithms have been designed (for example *linear static scaling*, *logarithmic scaling*, *exponential scaling* etc.), but it is not obvious which of these are optimal for a given application. Bäck writes in [Bäck96, pg. 112]:

> "However, even empirical comparisons of the performance of Genetic Algorithms relying on different scaling methods have not been performed, such that the choice of a scaling method and its parameterization can be thought of as a "black art"."

Bäck goes on to suggest using John J. Grefenstette's *scaling window* method [Grefenstette] when using GAs for minimization tasks.

4.3.1 Scaling window method

Simple to implement, the scaling window method is used when applying a GA for minimization. It transforms small objective values into large fitness values and also ensures that the resulting fitnesses are always non-negative. The fitness of an individual is computed by subtracting its objective value from the largest (meaning the worst) objective value observed in the last w generations (w is called the *window size*). Bäck suggests a window size of $w = 5$.

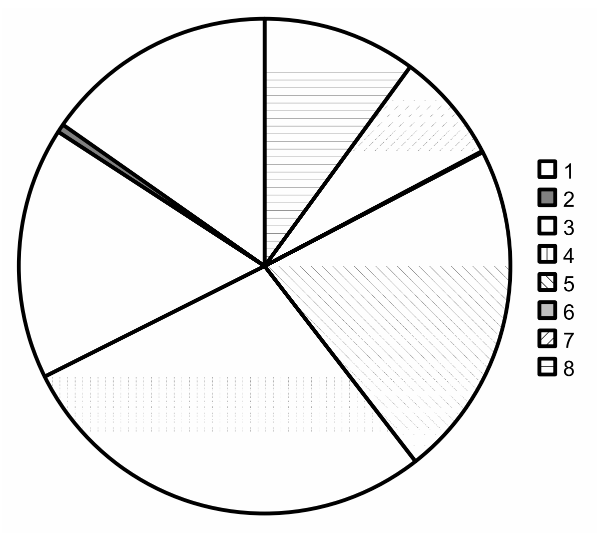
4.4 Selection operator

After the evaluation of a generation (meaning that all fitnesses of the individuals have been computed) and fitness scaling, the parents of the next generation are chosen by the *selection operator*. In GAs *proportional selection* [Bäck96, pg. 117] is used which chooses the parents of the next generation randomly from the population with *survival probabilities* that are proportional to the fitness value of the individuals.

Proportional selection can only work correctly if the fitness values are non-negative (since a negative value does not make sense as a proportion). If the problem is one of minimization rather than maximization then some sort of scaling is needed to translate small objective values into large fitness values. Such transformations always have an effect on the survival probabilities.

4.4.1 Roulette wheel selection

The simplest implementation of the proportional selection operator is called *roulette wheel selection*. As the name implies, roulette wheel selection works like a simulated throw of a ball in a roulette wheel. Each individual is represented by one “slot” of the wheel, but in contrast to a real roulette wheel each slot is different in size, proportional to the fitness of the individual. To select a parent, the wheel is spun and the individual in whose slot the ball stops is chosen. This is implemented by generating a random number between zero and the sum of all fitnesses in the population and selecting the individual at the position at which the running sum of fitnesses exceeds the random number. Fig. 4.2 shows the roulette wheel for the example population in Tab. 4.2 and the survival probabilities of the individuals.



Individual	Probability
1	15.32%
2	0.55%
3	16.51%
4	28.15%
5	22.10%
6	0.07%
7	7.21%
8	10.09%

Figure 4.2: “Roulette wheel” for the first generation of the example problem

4.4.2 Stochastic universal sampling

Stochastic universal sampling (SUS) selection [Baker87] is an improved implementation of proportional selection that is faster than roulette wheel selection and also avoids random sampling errors to large degree [Bäck96, pg. 120]. SUS also uses a roulette wheel with the spokes sized proportionally to the fitness of the respective individual, but instead of throwing the ball once for each selection, the entire set of parents is selected in one go. This is achieved by making as many evenly spaced markers around the wheel as parents are needed, and then spinning the wheel just once. The individuals that are under the markers are the ones that will be selected. SUS is the recommended selection operator for GA [Bäck96].

4.5 Recombination operator

The selection operator could be used alone to create the next generation, by applying it N times to choose the N individuals that make up the next generation. But if only selection were used, the population would eventually consist only of the fittest individual of the initial population (provided it isn't lost by chance) or another relatively fit individual that was present there, but the GA would not be able to find any solutions that were not present from the beginning. Additionally applying mutation on the offspring would generate individuals that were not part of the initial population, but at reasonable (meaning low) mutation rates progress would be very slow. *Recombination* greatly speeds up the evolution process by combining the genes of parent individuals to form offspring.

In GAs the *crossover operator* provides a way to recombine the already present solutions, potentially creating even better solutions. Crossover constructs the offspring strings from alternating parts of the parent strings. It is inspired by the biological DNA crossover process that generates the DNA of a child from the DNA of its parents by assembling it from pieces of the parent's strands. Crossover is a vital part of GAs [Bäck96, pg. 114]:

> "While mutation in Genetic Algorithms serves as an operator to reintroduce "lost alleles" into the population, i.e. bit positions that are converged to a certain value throughout the complete population and therefore could never be regained again by means of recombination, the *crossover* operator is emphasized as the most important search operator of Genetic Algorithms."

4.5.1 Single point crossover

The simplest form of recombination for GA is the *single point crossover* operator, which takes two parent strings and cuts them at a random position (the same in both strings, sometimes called *locus* after the analogous expression from biology) and then exchanges the two tailing parts, yielding two new offspring strings. For example a crossover of the strings "0000" and "1001" when cut exactly in the middle results in the offspring strings "0001" and "1000" (Fig. 4.3).

Using the crossover operator, the new population can be generated thus: use selection twice to choose two parent strings from the current population. With a certain probability, the *crossover probability* P_c, the two strings will undergo single point crossover. If they do, choose a random locus, perform crossover, and copy the two resulting strings into the temporary new population. If they do not, copy the two parent strings unchanged to the new population. This is repeated until the new population has as many members as the previous one. Then the previous population can be deleted and the new one takes its place.

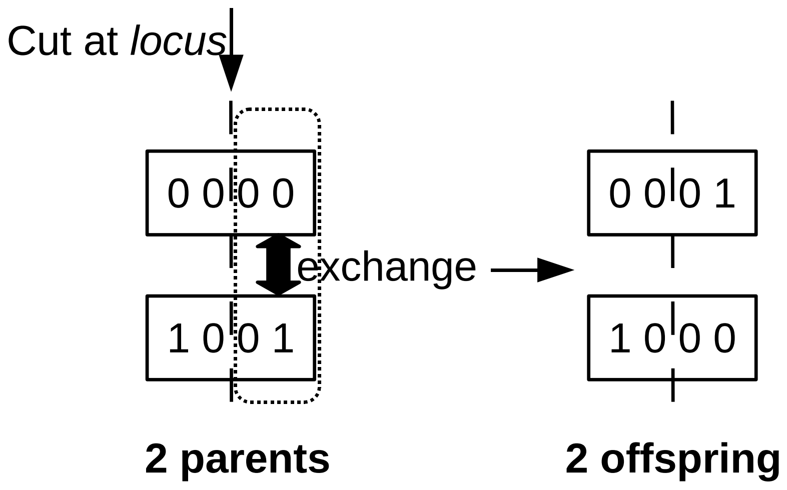


Figure 4.3: Single point crossover example

Multi-point crossover (two-point, three-point etc.) introduces more locus points to make the cuts. In this work, only single point crossover was used.

4.6 Mutation operator

Once the new population is created with selection and crossover, there is a chance that some of the individuals mutate. Mutation is named after biological mutation, where sometimes random copying errors will create base pair differences in a child's DNA strand.

The implementation used in this work is known as *flip mutation* because the operator loops through all bits of a string and each bit has a *mutation probability* P_m to be flipped to its opposite value. P_m should generally be very low because otherwise the GA degenerates into a random search. Mutation is applied on all members of the new population (after crossover has occured).

4.7 Schema theory

John Holland is the inventor of *schema theory*, which attempts to explain why classic GAs work. Though it is not without controversy concerning its conclusiveness, a short overview will be given here (for details see [Holland92], [Goldberg89] or [Coley98]).

A *schema* (plural *schemata*) is a string template made up of the original representation's characters and in addition a *wildcard* character or *meta-symbol* ("#"). One schema therefore represents a set of strings, for example the schema "1#001" represents both the bit strings "10001" and "11001". The *order* of a schema is equal to the number of non-wildcard characters in it, and its *defining length* is the distance between the first and the last non-wildcard character.

The *Schema Theorem* that follows from schema theory computes that "short, low-order, above-average schemata will be given exponentially increasing trials in subsequent generations" [Goldberg89] (essentially because these are the schemata that will be least disturbed by the genetic operators). These schemata are also called *building blocks*. The *building block hypothesis* [Goldberg89] states that GAs work by gradually combining building blocks into larger solutions, and that GAs work best if the problem is structured in such a way that this is possible.

According to Goldberg [Goldberg89] the *principle of minimal alphabets* also follows from schema theory. It states that the set of characters (the *alphabet*) used for encoding should be as small as possible for the problem at hand, because the smaller the alphabet, the more schemata are processed per generation. The smallest possible alphabet is the binary alphabet of just 0 and 1, so that one should use binary encoding preferentially.

Chapter 5

Evolution Strategies

5.1 History

In 1964 the two German aerospace engineering students Ingo Rechenberg and Hans-Paul Schwefel tested heuristics for experimental optimization (“experimental” because this experiment was not simulated in a computer, but actually performed with a physical object). The objective was to adjust the angles on a jointed, folding plate so that it would achieve the shape with the minimal drag in a wind tunnel [Rechenberg65]. In the beginning the plate would have a random “snake” shape, and the known optimum was a completely straight plate (Fig. 5.1). They first tried a method that modified only one variable (joint angle) at a time and a gradient based technique, both of which failed because they got stuck prematurely in sub-optimal configurations. Rechenberg then developed a different approach: all angles at once

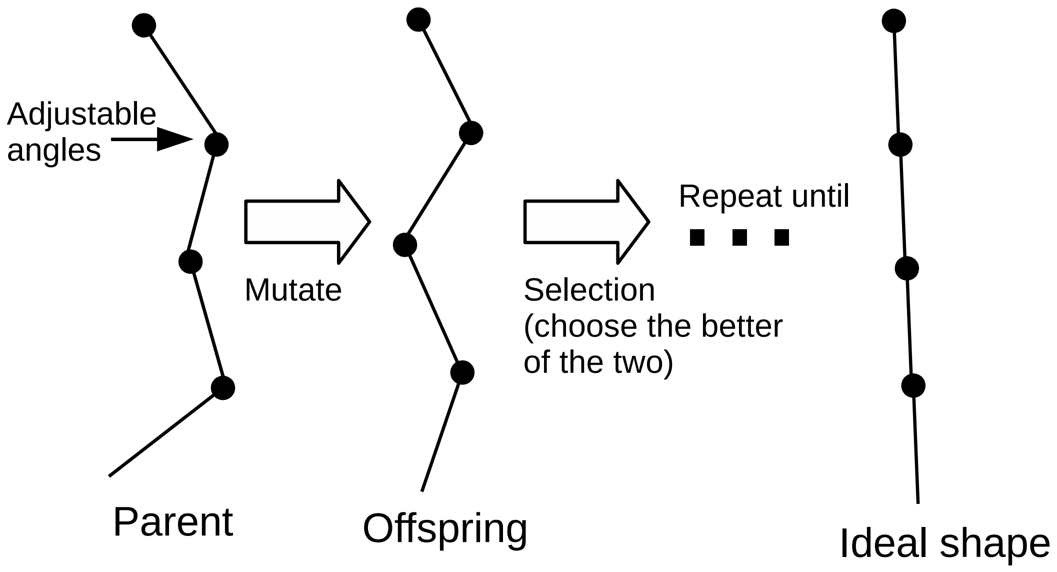


Figure 5.1: The jointed plate *experimentum crucis* for Evolution Strategies

would be changed randomly, and the amount of the change would be more likely to be small rather than large (from an observation that in nature small mutations are more likely than large ones). If the performance of the new shape was better than the previous one, it would become the new starting point for the next mutation (in essence there was one “parent” and one “offspring”). This new method had unexpectedly good results and was applied on a few more hydrodynamical problems with success [Beyer02]. They called their method “cybernetic solution path” or *Evolution Strategy* (ES). The folding plate experiment became to be known as the *experimentum crucis* of ESs.

Schwefel later ran the first computer simulation of the ES on a Zuse Z23 computer [Schwefel65]. Just like the experimental optimization version of the algorithm, this *two-membered ES* used one “parent” and one “offspring”, where the offspring would compete with the parent and replaced it if it was better. The representation of the individuals was a vector of floating point numbers, making ESs more similar to EPs rather than the bit strings of GAs. The mutation operator added a normally distributed random number $N(0, \sigma)$ with a mean of zero and standard deviation σ to every variable v of a solution, so that small changes from the parent would be more likely than large ones:

$$v' = v + N(0, \sigma) \tag{5.1}$$

Rechenberg later found a rule to adjust the standard deviation σ of the random numbers optimally during the run of the algorithm depending on the success of the mutation operator. This is a major difference that distinguishes ESs from both GAs and EPs: in ESs the mutation parameters are evolved along with the solution variables, they use *self-adaptation*.

ESs are generally used for solving problems with continuous values, but they have been extended for solving discrete problems as well. In this work, only the real valued ES is used.

5.1.1 Multimembered ES

The original ES’s use of a temporary “population” consisting of one parent and one offspring, combined with the selection process that always chooses the better of these two individuals is now referred to as a $(1 + 1)$-ES. The first number is the number of parents, the second number is the number of generated offspring and the $+$ sign indicates that the selection operation chooses the new population from the best of *both* parents and offspring.

This notation is used in the literature to describe more advanced forms of ESs with larger populations as well: traditionally the symbol μ stands for the number of parents, λ is the number of offspring and the selection operator is either *plus selection* (choosing the new population from the combined set of parents and offspring) or *comma selection* (choosing *only* from the set of offspring). All ESs can therefore be classified as either a (μ, λ)-ESs or a $(\mu + \lambda)$-ESs. For example, a $(15, 100)$-ES has a parent population size of 15 that generates 100 offspring in each generation, and the new population is chosen from the 100 offspring only.

Multimembered ESs can also use recombination operators, although these work in a different way than those used in GAs and EPs. In ESs several parent individuals can participate in the creation of a single offspring (in contrast to GAs, where the crossover operation creates two offspring from two parents). The number of parents involved in the creation of an offspring is denoted by ρ, which is called the *mixing number* [Beyer02]. The case $\rho = 2$ is the standard case of *sexual recombination* (two parents per offspring), cases with $\rho > 2$ are called *multirecombination* [Beyer01] or *panmictic recombination* [Bäck96]. If $\rho = 1$ then no recombination is used. When recombination is used, this can be indicated in the ES notation as $(\mu/\rho \overset{+}{,} \lambda)$ (read "Mu Slash Rho Plus/Comma Lambda"-ES).

5.1.2 Self-adaptation

Schwefel's mutation operator added a random normal-distributed value with a mean of zero and a standard deviation σ to each solution variable. Rechenberg later derived a rule for automatically adjusting σ during the course of the algorithm to achieve optimal performance. His $1/5$-*success rule* states that if more than $1/5$ of the mutations so far were successful, increase σ to continue with larger steps, otherwise decrease it (by multiplying it with a factor a). The $1/5$-success rule is limited in applicability since it only works on a single *strategy parameter* (a parameter controlling the algorithm that is not part of the problem solution) and is usually only used for $(1+1)$-ESs.

Schwefel [Schwefel74] further improved self-adaptation in ESs by including the strategy parameters into the individuals themselves: individuals contain *endogenous strategy parameters* which are also subject to evolution, meaning that they are modified by mutation and recombination. In practice this means that every *object parameter* (a part of the problem solution) is coupled with its own σ-parameter. This type of self-adaptation is known as σ-self-adaptation or σSA [Beyer95].

Other strategy parameters that are not part of individuals and stay constant over the run of the algorithm are called *exogenous strategy parameters* (for example μ and λ). When self-adaptation with endogenous strategy parameters is used, the ES must include mutation and recombination operators for these. In ESs there is no obligation to use the same type of mutation or recombination on the strategy parameters as for the object parameters, and indeed the strategy parameters of an offspring need not necessarily come from the same parents that provided its object parameters.

In the following the details of the ES implementation used in this work will be presented. Details of the history and theory of ESs can be found in [Beyer02], [Michalewicz94], [Bäck96] and [Schwefel95].

5.2 $(\mu \overset{+}{,} \lambda)$-ES

The ES used in this work can be set to use plus- or comma-selection, and the number of parents μ and the number of offspring λ can be chosen freely. Self-adaptation is based on an endogenous strategy parameter σ coupled to every object variable (so a solution for a problem with n variables will also include n σ-parameters). Recombination uses either two parents ($\rho = 2$) or is panmictic ($\rho = n$). Strategy parameter and object parameter recombination are separate algorithms. Fig. 5.2 shows the pseudocode of the ES.

```
initialize PARENT_SET with mu random individuals;
for (N generations)
{
    new OFFSPRING_SET;
    for (lambda offspring)
    {
        new OFFSPRING;
        strategy parameter recombination(PARENT_SET, OFFSPRING);
        object parameter recombination(PARENT_SET, OFFSPRING);
        add OFFSPRING to OFFSPRING_SET;
    }

    for (all OFFSPRING in OFFSPRING_SET)
    {
        mutate strategy parameters(OFFSPRING);
        mutate object parameters(OFFSPRING);
    }

    compute fitnesses of OFFSPRING_SET;

    if (plus mode)
        add PARENT_SET to OFFSPRING_SET;

    PARENT_SET = best mu individuals in OFFSPRING_SET;
}
```

Figure 5.2: Pseudocode of the $(\mu \overset{+}{,} \lambda)$-ES

5.2.1 Representation

Similar to the EP implementation, an ES individual is represented as a vector of *ES genes*. An ES solution $S = \{G_1, G_2, \ldots, G_n\}$ is a vector of n ES genes G_i, where each gene G_i itself has the structure $G_i = \{v_i, \mathrm{LB_i}, \mathrm{UB_i}, \sigma_i\}$ (all elements of G_i are floating point numbers).

The v_i are the actual variable values and the $\mathrm{LB_i}$ and $\mathrm{UB_i}$ are the respective lower and upper bounds in between which the v_i may assume any value ($v_i \in [\mathrm{LB_i}, \mathrm{UB_i}]$). Each gene also carries σ_i, an endogenous strategy parameter that is used for the σSA mutation operator.

The initial population is generated by assigning each gene of every individual a random value from the allowed range, and the σ_i are set to a chosen constant.

5.2.2 Selection operator

ESs contains essentially two “selection” steps: the first occurs in the stage where the offspring are generated from the parents (lines 5-11 in Fig. 5.2). In this step the actual fitness of the parents does not figure into which parent participates in the creation of an offspring, but rather the parents are chosen totally randomly. This is unlike GAs and EPs where the probability of an individual being chosen for recombination is proportional to its fitness.

The second selection step is the one that determines which of the offspring individuals survive into the next generation (line 24 in Fig. 5.2). In ESs, only the μ best offspring individuals are selected, while the rest are destroyed. This type of selection is called *truncation selection* or *breeding selection* and is completely deterministic, as opposed to GAs and EPs which normally use some probabilistic selection operator like roulette wheel selection.

In the $(\mu+\lambda)$-ES the best μ individuals of the union of the parent and the offspring set are selected to become the next generation. Therefore the $(\mu+\lambda)$-ES is *elitist* because the best solution is never lost. Bäck [Bäck96] notes that while the $(\mu+\lambda)$-ES may seem effective due to the inherent elitism, there are drawbacks as well: the unlimited lifetime of individuals might lead to an inability to leave local optima, and the self-adaptation mechanism can become disturbed by bad strategy parameters that live for too long. Due to these reasons the (μ, λ)-ES is generally recommended.

In the (μ, λ)-ES only the best μ individuals of the parent set containing λ individuals are selected to become the next generation. This means that the parents die off and are lost even if they were better than the offspring, and the best solution in a generation may be less good than the previous best one (in contrast the progression of the best fitness is monotonous in the $(\mu+\lambda)$-ES). For the (μ, λ)-ES λ must be greater than μ since for the case $\lambda = \mu$ every generated offspring is selected, meaning there is no selection pressure at all and the algorithm becomes a random walk. According to Schwefel [Schwefel95], it is considered sufficient that $\lambda \geq 6\mu$ and Bäck [Bäck96] suggests $\mu/\lambda \approx 1/7$. The parameter μ should be chosen larger than 1 so that selective pressure does not become too strong [Bäck96] and the search behaves unstably [Schwefel95].

5.2.3 Recombination operator

In contrast to the recombination operators in GAs and EPs which take 2 parents and create 2 offspring from them (for example single point crossover), ES recombination operators combine 2 (or more) parents into just one offspring. While recombination is not strictly mandatory to use in ESs (when $\rho = 1$ one can simply copy a random individual), Schwefel reported a noticeable improvement of the search process when using it. For real-valued ESs there are two main types of recombination: *discrete* recombination where each component offspring value is chosen randomly from one of the parents, and *intermediate* recombination, which forms the offspring value from averaging all the respective parent values (Fig. 5.3). There are two variations for both

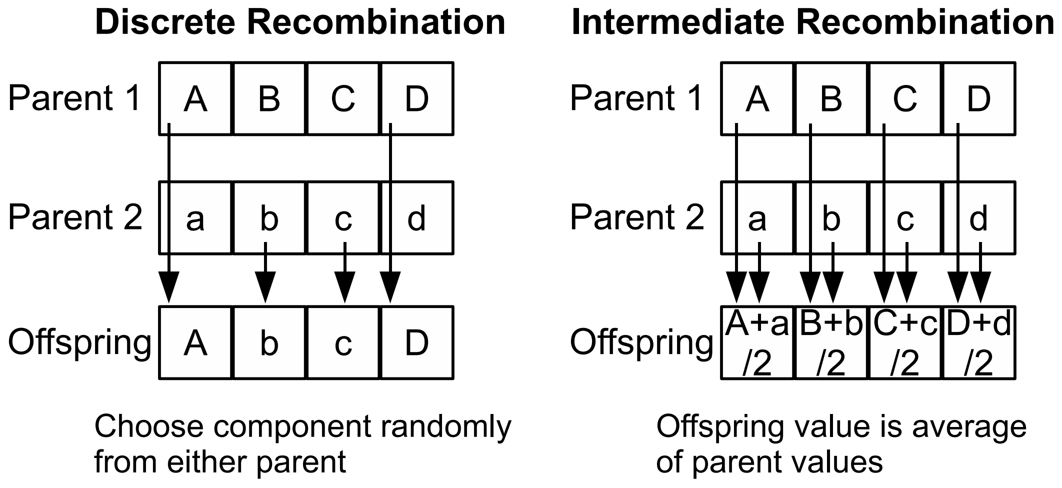

Discrete Recombination
Intermediate Recombination
Parent 1
A
B
C
D
Parent 2
a
b
c
d
Offspring
A
b
c
D
A+a /2
B+b /2
C+c /2
D+d /2
Choose component randomly from either parent
Offspring value is average of parent values

Figure 5.3: Discrete and intermediate recombination in real-valued ESs

discrete and intermediate recombination: the *sexual* (or *local*) form combines two chosen parent individuals and the *panmictic* (or *global*) form holds one parent fixed and uses a random individual chosen from the entire population as the other parent (a new parent is chosen for each component gene of the solution).

Recombination is applied separately on object and strategy parameters and need not use the same type of operator, so there are four possible combinations for applying the two types of recombination operator: object parameter discrete/intermediate and strategy parameter discrete/intermediate. It was found empirically that discrete sexual recombination for object parameters and intermediate panmictic recombination for strategy parameters is generally optimal [Beyer02] [Bäck96].

5.2.4 Mutation operator

In ESs the mutation operator mutates the object parameters and the endogenous strategy parameters contained in an individual. Strategy parameters are always mutated first before the object parameters to ensure a correctly behaving self-adaptation [Beyer02]. This way the newly generated strategy parameter affects the new object parameter immediately. A well adapted endogenous strategy parameter then indirectly contributes to the fitness of the resulting individual.

5.2.4.1 Strategy parameter mutation operator

There are several versions of the self-adapting mutation operator used in ESs. The simplest form uses a single σ for all object variables. A more advanced form uses more than one σ, up to as many as there are solution variables. The *Covariance Matrix Adaptation*-ES (CMA-ES) [Hansen96] goes further and introduces a covariance matrix that correlates the mutation parameters.

The behavior of these mutation operators can be visualized as shown in Fig. 5.4 (for a two-dimensional objective function). The crosses represent the solution values. Drawn around them is the area that contains offspring with the same probability of generation. This is just a circle when a single σ is used, and an ellipse for many σ because the mutation step size may be different for the search space's axes. CMA-ES enables the mutation to generate offspring in a rotated elliptical area, allowing a better adaptation to the shape of the search space. Self-adaptation with a single σ is called *isotropic self-adaptation*, when using many σ it is called *non-isotropic self-adaptation* and CMA-ES is *correlated self-adaptation* [Deb99].

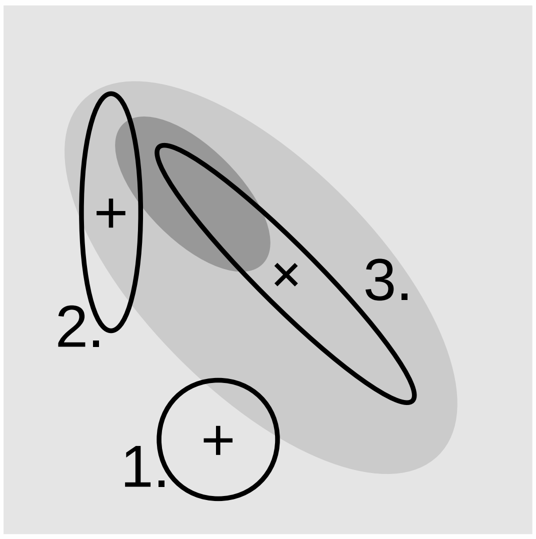


Figure 5.4: Visualization of ES mutation operators: 1. single σ 2. many σ (uncorrelated mutation) 3. CMA-ES

In this work, only the many σ (one σ for each of the n solution variables) mutation operator was used. It modifies the σ_i parameter of each ES gene of a solution as follows: [Bäck96]

$$\sigma_i' = \sigma_i \cdot exp(\tau_0 \cdot N(0,1) + \tau \cdot N_i(0,1)) \tag{5.2}$$

The normally distributed random number $N(0,1)$ (mean zero, standard deviation 1) stays constant for the entire solution, while the $N_i(0,1)$ are generated separately for each ES gene. The σ are modified using multiplication with a random number with a *lognormal distribution*, which was suggested by Schwefel [Schwefel74] and has become the standard. This choice of operation ensures that the σ always remain positive, changes are more likely small than large and have a mean of 1.

The constants τ_0 and τ are exogenous strategy parameters. τ_0 is the *global learning parameter* and τ the *coordinate learning parameter*. Schwefel recommends from theoretical and empirical evidence [Beyer02] that these should be:

$$\tau_0 = \frac{c}{\sqrt{2n}} \tag{5.3}$$

$$\tau = \frac{c}{\sqrt{2\sqrt{n}}} \tag{5.4}$$

The constant c is the *progress coefficient*, which depends on μ and λ. Bäck recommends setting $c = 1$ [Bäck96], and [Bäck97] contains a table with theoretically derived values for ES using panmictic intermediate recombination that range from about 0.3 to 2.3 for values of μ and λ up to 100.

The σ_i must be assigned a starting value when generating the initial population. Bäck [Bäck96] recommends setting it to 3, and in [Deb99] $(x^u - x^l)/\sqrt{12}$ was chosen for simulations (assuming that solutions are uniformly distributed in the range x^l to x^u).

5.2.4.2 Object parameter mutation operator

After mutating the strategy parameter, each object parameter v_i is mutated by adding to it a normally distributed random value $N(0, \sigma_i)$ with a mean of zero and a standard deviation of σ_i:

$$v_i' = v_i + N(0, \sigma_i') \tag{5.5}$$

The σ_i is part of the same ES gene that contains the object parameter v_i.

Should a mutation operation in gene G_i lead to the object parameter v_i being outside of its bounds $\mathrm{LB_i}$ or $\mathrm{UB_i}$, the v_i is set to the value of the closest exceeded limit.

Chapter 6

Experimental Results

6.1 Measuring runtimes

6.1.1 Objective of the experiment

When optimizing the parameters of MINISAT the objective function (which is to be minimized) will be the total run time required for solving a set of SAT problems. To measure the runtime of a program, the wall clock time between its start and end can be taken, but this will include the time spent on operating system jobs and other unrelated activities. Operating systems offer functions for measuring only the actual process time, which counts only the multitasking time slices used by the thread running the program. Under Linux, this *user time* can be retrieved with the getrusage() system function. MINISAT uses getrusage() to measure its own solving time and prints out the measured time after solving.

Unfortunately getrusage() is imprecise, meaning that even for the same program the returned user time will vary somewhat with each execution due to inherent imprecision of the time measurement. The EAs will have to be able to deal with these imprecisions, therefore this experiment will investigate what order of magnitude the variations of the measurements can reach.

6.1.2 Single user mode

The tests were first run under single-user mode to minimize interference by other running programs (for example graphical interface services). In single user mode the operating system provides a minimal environment and only runs essential processes. A number of SAT problems were chosen from the first qualification round of the 2006 SAT-Race [Sinz06] and each solved 1000 times with MINISAT on an Intel®Pentium®4@2.4GHz and an Intel®Core^{TM}2 4300@1.8GHz computer running Linux. A delay of 5 seconds was inserted between each solver start to allow the system time to do any cleanup operations after the previous solving attempt.

The recorded solving times that were printed out by MINISAT after every execution were sorted from lowest to highest, and distributed into 31 ($\sqrt{1000}$) equidistant bins between the minimum and maximum time, yielding the histograms in Figs. 6.1 and 6.2. The roughly bell-shaped histograms show that nearly all the measured run times are within 3 standard deviations σ of the mean μ, with a few outliers beyond. Figs. 6.3 and 6.4 show the standard deviation and relative error ($\frac{\sigma}{\mu}$ in percent) of each experiment as a function of the mean solving time.

In both experiments the standard deviation rises roughly linearly with the mean solving time, while the relative error varies between 0.48% to 1.17% for the slower computer (Intel®Pentium®4@2.4GHz) and 0.20% to 0.41% for the faster computer (Intel®Core$^{\text{TM}}$2 4300@1.8GHz). It can also be seen that there are always some rare outliers that are very strongly off from the average (for example the problem `hoons-vbmc-s04-07` on the Pentium®4 once runs 223 seconds when the average is only about 211 seconds).

6.1.3 Interferences due to heavy load

Next it was investigated what effect many concurrent, CPU-heavy processes have on run time measurements. This experiment was executed under the X-Windows graphical environment with various demanding programs running alongside the MINISAT benchmark program. Again a number of SAT problems were each solved 1000 times with MINISAT on the Intel®Core$^{\text{TM}}$2 4300@1.8GHz machine. The histograms are shown in Fig. 6.5.

It can be seen that in all cases but one there are actually two different, far apart peaks in the histograms. Due to this the relative error of the measurements is substantially higher than in the experiment under single-user mode, more than 10 times higher in fact.

6.1.4 MiniSAT with timeout implementation

The standard implementation of MINISAT has no built-in functionality for stopping after a timeout period is exceeded. Since timeouts are necessary for the optimization with EAs, it was implemented as a periodic call to getrusage(), more specifically a check if the alotted time is exceeded every time a new variable assignment decision is made. Another experiment was run to investigate if this added overhead has an influence on the imprecision of the time measurements. MINISAT with the timeout implementation was used to solve the problem `manol-pipe-c6n` 1000 times (with a high enough timeout so that the problem could be solved) on 6 PCs with different speeds.

Fig. 6.6 shows the relative error in percent charted over the average solving time on each of the 6 computers. It can be seen that the relative error stays under 1% for all but the slowest computer, which reaches 1.31%. These figures are close to the results for `manol-pipe-c6n` in standard MINISAT without timeout for this

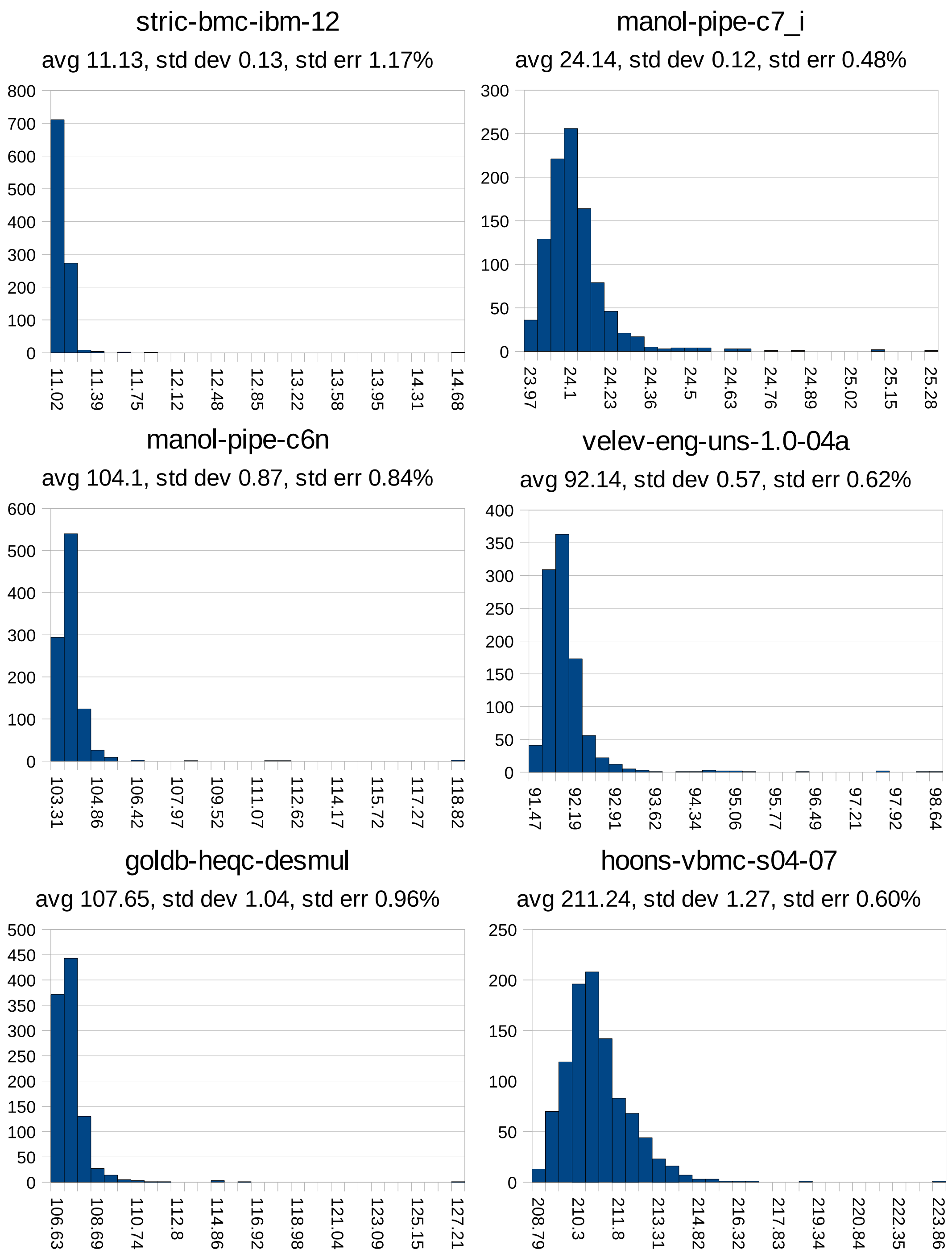


Figure 6.1: Solving time variation histograms on an Intel®Pentium®4@2.4GHz machine in single-user mode

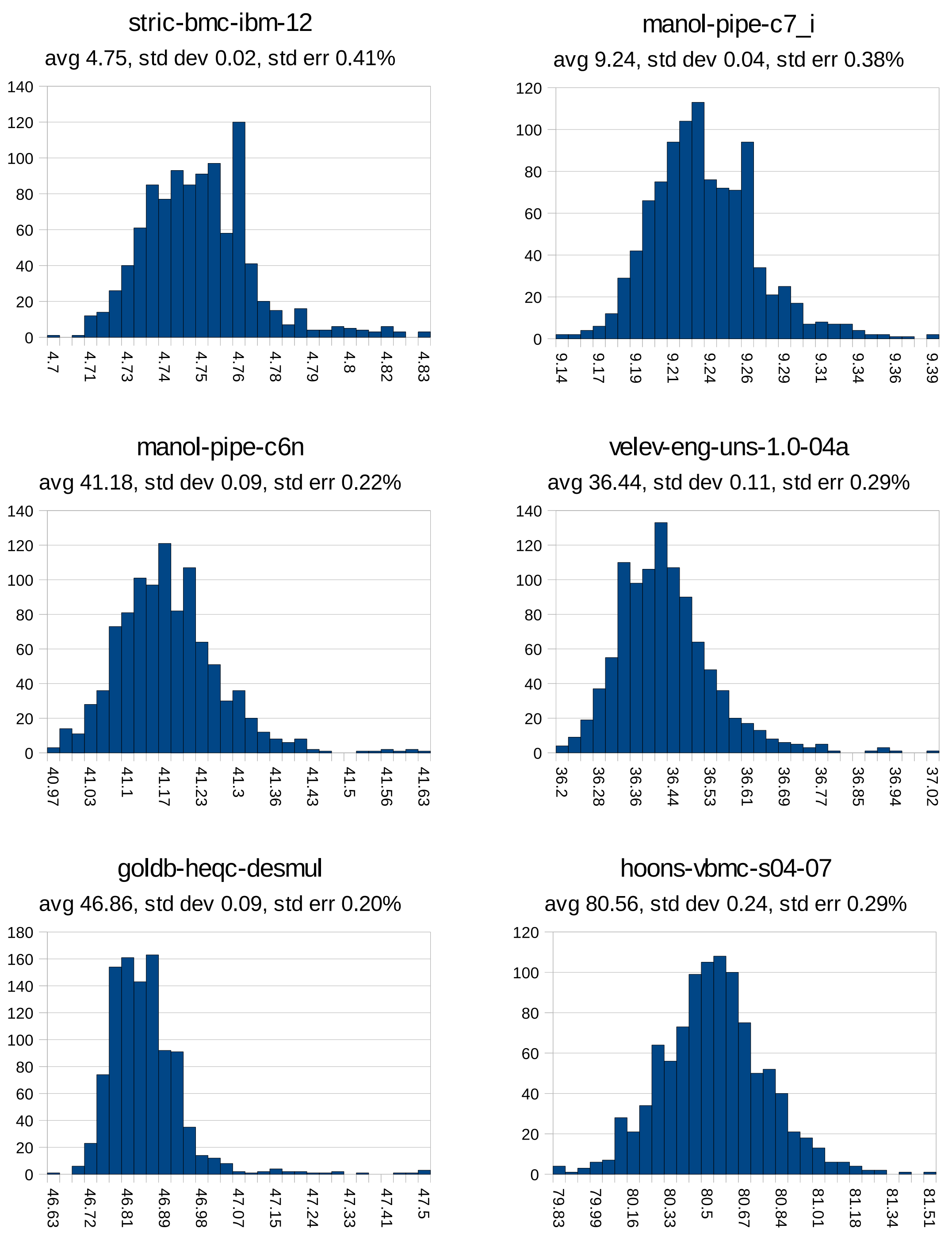


Figure 6.2: Solving time variation histograms on an Intel®Core^{TM}2 4300@1.8GHz machine in single-user mode

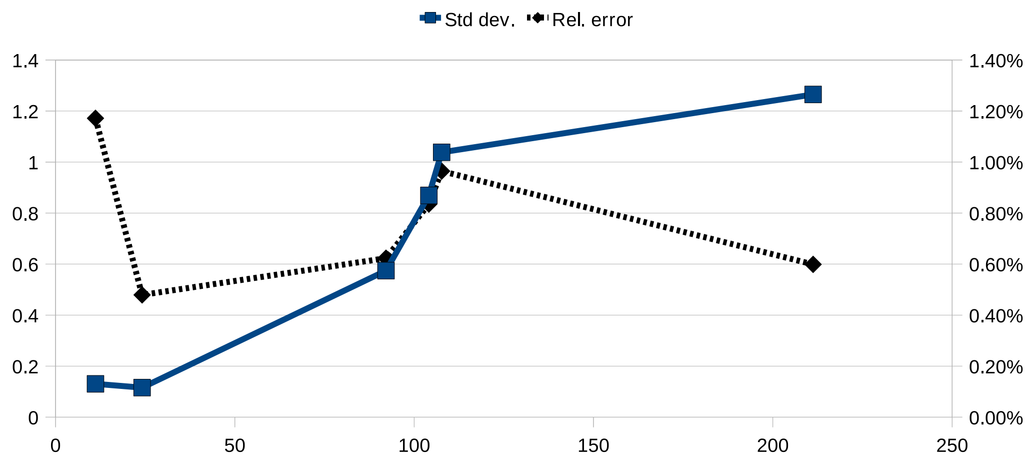


Figure 6.3: Standard deviation σ (left Y axis) and relative error $\frac{\sigma}{\mu}$ (right Y axis) as a function of mean solving time on an Intel®Pentium®4@2.4GHz machine in single-user mode

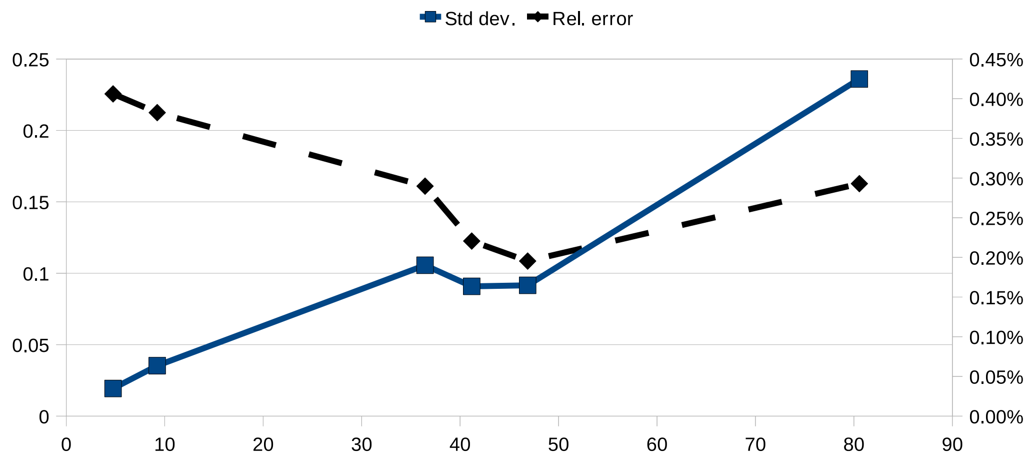


Figure 6.4: Standard deviation σ (left Y axis) and relative error $\frac{\sigma}{\mu}$ (right Y axis) as a function of mean solving time on an Intel®Core$^{\text{TM}}$2 4300@1.8GHz machine in single-user mode

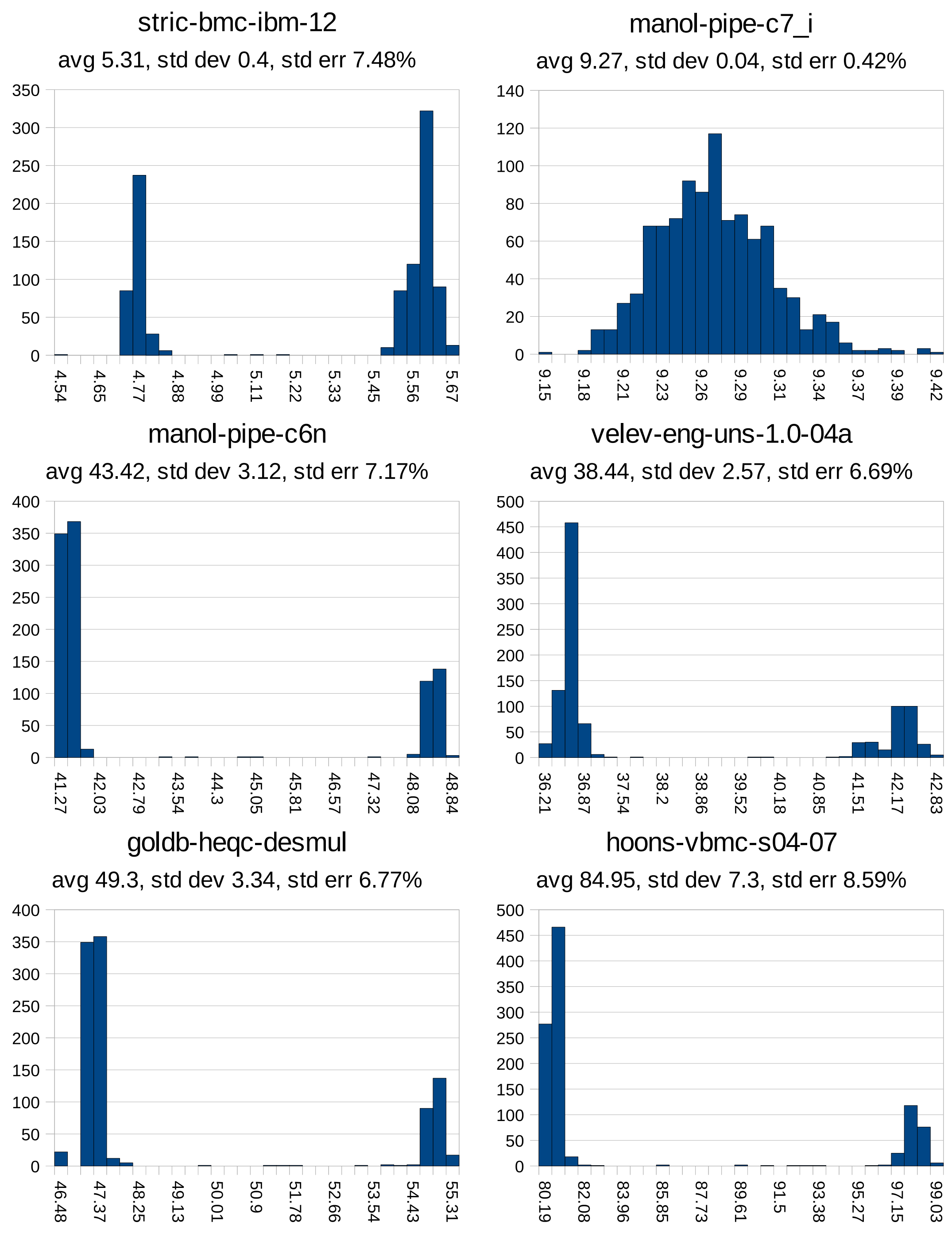


Figure 6.5: Solving time variation histograms on an Intel®Core^{TM}2 4300@1.8GHz machine under heavy load in multi-user mode under X-Windows

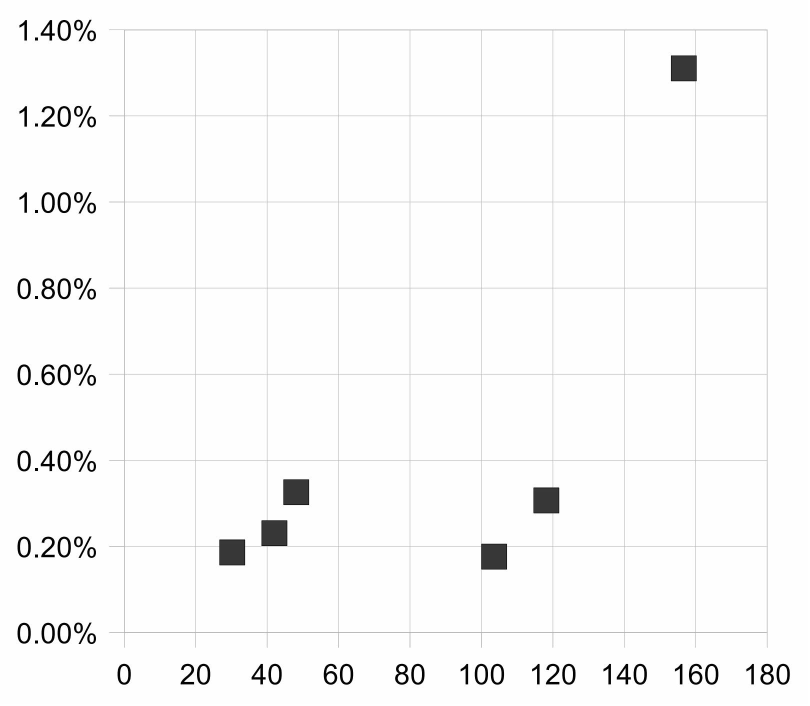


Figure 6.6: MiniSAT with built-in timeout implementation: standard deviation in percent of measured runtimes vs. average solving time for problem `manol-pipe-c6n` on 6 different computers

problem (0.84% and 0.22%). The overhead for the timeout checks was therefore considered negligible.

6.1.5 Conclusion

Even under circumstances with minimal interference by other processes the user time measurements using the getrusage() system function has some degree of imprecision. The errors are roughly normally distributed, and the relative error is higher the longer the computation takes. This means that any measured SAT solver run times used as objective values for an optimization algorithm will have to be able to cope with some amount of *noise*, since even the same solver configuration will have different objective values in two different evaluations. Additionally it should be ensured that any optimization programs are run with as few other concurrent processes as possible. Even then, there will be rare but unavoidable evaluations that give a much higher run time than would be expected on average. A relative error of about 1-2% should be expected when running under "ideal" conditions.

6.2 Testing the EAs

6.2.1 Objective of the experiment

When an optimization algorithm has been newly implemented, it must be ensured that it is working correctly. Programming errors or misconfigurations can cause the software to become unable to find optima. For this reason, it is useful to have a suite of *test problems* with known optima and fitness landscapes. The new algorithm can be applied on the test problems and the results can be compared to known, reliable implementations.

In this section, two test functions will be presented and then applied on the implementations of the GA and the ES that will be used for SAT solver optimization later. The test functions as well as the parameter settings that will be used for the GA and the ES were taken from [Bäck96], where it was also shown that the ES is generally the best choice for numerical optimization problems. If the GA and ES are correctly implemented, it should therefore be possible to show that both algorithms can find the optima of the test functions and that the ES has better performance than the GA.

6.2.2 Test functions

GA test suites were thoroughly explored in Kenneth DeJong's thesis [De Jong75], which introduced a set of 5 problems that became a standard suite. Hans-Paul Schwefel, one of the co-developers of Evolution Strategies, also presented a very large set of test functions in [Schwefel81] specifically for that algorithm. [Torn89] describes a number of popular test functions used in optimization. DeJong's functions were of low dimensionality and relatively simple, therefore Thomas Bäck expanded upon the idea in [Bäck96]. Bäck describes a number of criteria a test suite should meet to model the many difficulties an application can encounter.

- All functions should be scalable with regard to the number of unknowns to optimize (the number of dimensions of the search space).
- Include a single-peaked (*unimodal*) function to test convergence speed.
- Include step functions that provide no gradient information.
- Include multimodal functions (several peaks).

His suite of 5 functions are separated into two groups: two relatively easy functions for testing *convergence velocity* and three harder functions for testing *convergence reliability* ("*the capability to yield reasonably good solutions in case of highly multimodal topologies*" [Bäck96]). In this work, two functions (one each from the easy and the hard set) from Bäck's test suite were used: the *sphere model* and *Ackley's function*.

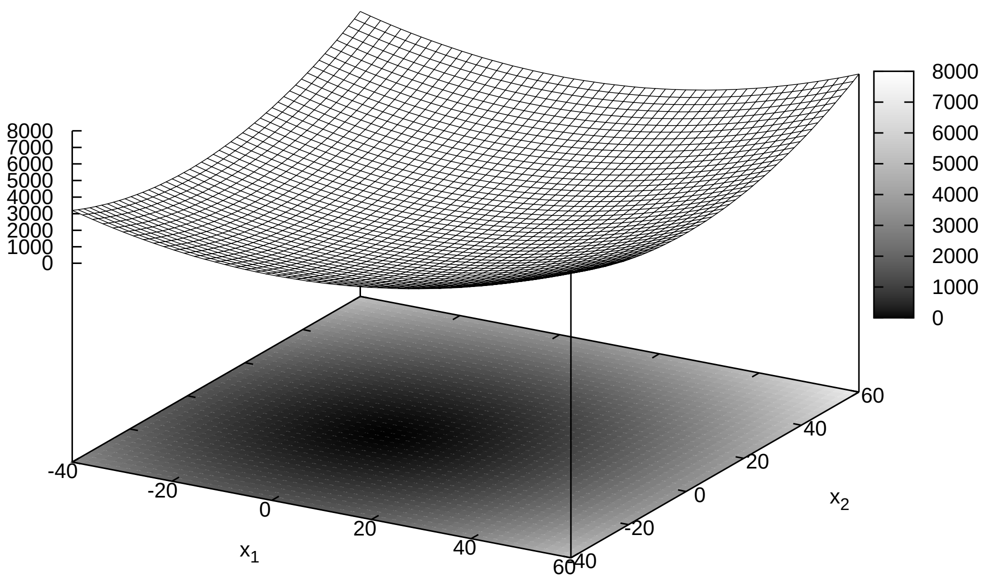


Figure 6.7: Sphere model for $n = 2$

6.2.2.1 Sphere model

The *sphere model* (Eq. 6.1) was DeJong's first test function, so called because it is a smooth (continuous) (hyper-)sphere with a single minimum (uni-modal) with value 0 at the origin.

$$\mathrm{SPHERE}(\vec{x}) = \sum_{i=1}^{n} x_i^2 \tag{6.1}$$

Bäck also uses SPHERE in his easy group of test functions. The number of dimensions can be adjusted with the paramater n. Since it is so simple in structure, performance on SPHERE is a measure of the convergence speed of an optimization algorithm.

Bäck [Bäck96] assumes $n = 30$ as the standard test case and a variable range of $-40 \leq x_i \leq 60$. The upper and lower limits are asymmetrical to avoid having the global optimum at the exact center, where for example a GA encoding might have a highly regular structure (for example all zeroes) that could make finding the global optimum easier than it would otherwise be. Fig. 6.7 shows the three-dimensional plot of the sphere model (for $n = 2, f(\vec{x}) = {x_1}^2 + {x_2}^2; x_1, x_2 \in [-40, 60]$).

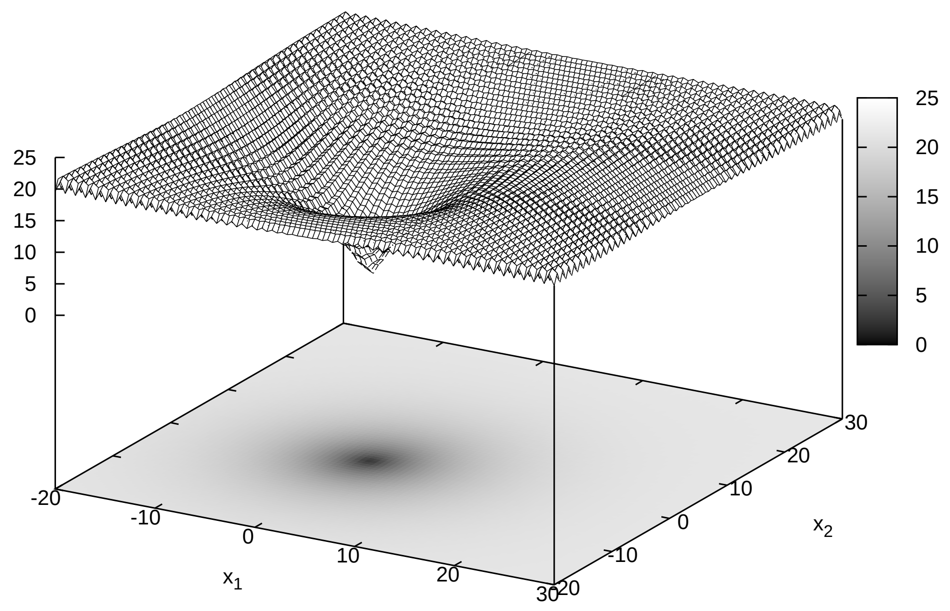


Figure 6.8: Ackley's function for $n = 2$ (entire search region)

6.2.2.2 Ackley's function

Ackley's function (Eq. 6.2) is continuous and multimodal (has many peaks).

$$\mathrm{ACKLEY}(\vec{x}) = -c_1 \cdot \exp\left(-c_2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n} cos(c_3 \cdot x_i)\right) + c_1 + e \tag{6.2}$$

It was presented in [Ackley87] for the two-dimensional case and generalized for the n-dimensional case in [Bäck96]. Bäck uses it in his harder group of test functions and sets the parameters to $c_1 = 20; c_2 = 0.2; c_3 = 2\pi$ so that the global minimum is at the origin at $\vec{x} = (0, \ldots, 0)$ with the function value 0, and assumes a search region of $-20 \le x_i \le 30$. Fig. 6.8 shows the plot of the entire search region for $n = 2; f(\vec{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{2}(x_1^2 + x_2^2)}\right) - \exp\left(\frac{1}{2}\left(\cos(2\pi x_1) + \cos(2\pi x_2)\right)\right) + 20 + e; x_1, x_2 \in [-20, 30]$ and Fig. 6.9 shows a close-up of the region near the global optimum. The outer regions are relatively flat, but there is a trough around the global minimum where the oscillations of the cosine wave become more pronounced. Ackley's function is considered moderately hard because simple hill climbing algorithms will get stuck in one of the many local minima.

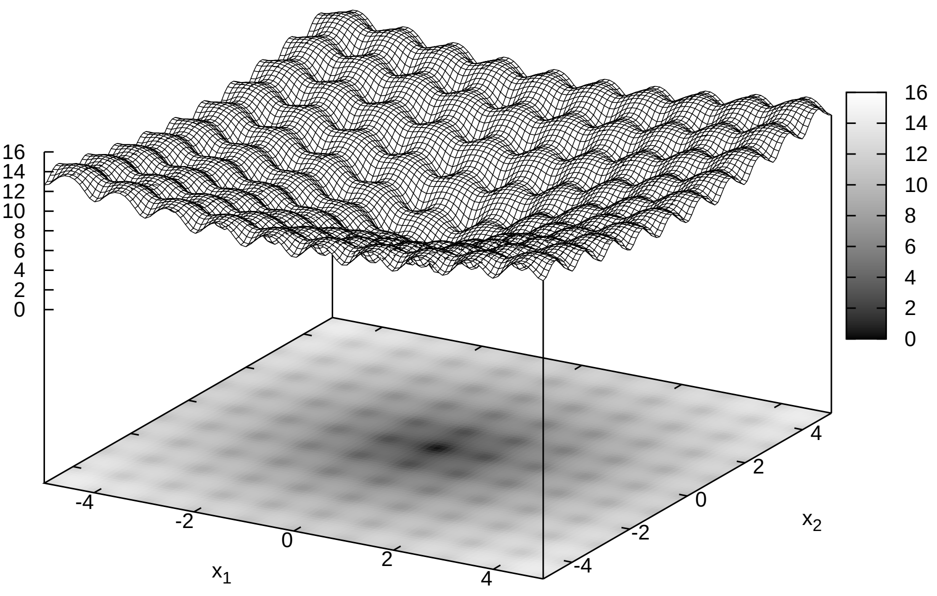


Figure 6.9: Ackley's function for $n = 2$ (close-up view of the area around the global optimum at the origin)

6.2.3 Standard GA and ES parameters

Before an EA can be applied on a problem, values for a set of exogenous parameters have to be decided. These include for example the population size to use, the types of operators and everything else that is not subject to evolution but rather controls the evolutionary process. For this work, the standard parameters given in [Bäck96] were used as a baseline:

- Length of each object variable: 30 bits.
- Gray code is used for the bit string encoding.
- For noisy objective functions all fitness values were recomputed in each generation.
- Population size: 50.
- Maximum number of generations: 50.
- Elitism was not used.
- Recombination operator: single point crossover with crossover probability 0.6.
- Mutation rate: $p_m = 0.001$ (flip mutation).
- Selection operator: Stochastic universal sampling (SUS) selection.
- For fitness scaling, the scaling window algorithm with window size 5 was used.

In addition to GAs, Bäck [Bäck96] also provides a set of standard parameters for ESs. To make the GA and ES comparable, his recommended offspring population size $\lambda = 100$ was halved for this work, and likewise the parent population size $\mu = 15$ was halved (and rounded up). This way both the GA and the ES are allowed to perform the same total number of objective funtion evaluations (50 times in each generation for 50 generations, so 2500 total). Otherwise Bäck's recommendation were accepted and the settings are:

- Selection type is comma selection ((μ, λ)-ES).
- n standard deviations for n object parameters.
- Parent population size: $\mu = 8$.
- Offspring population size: $\lambda = 50$.
- Maximum number of generations: 50.

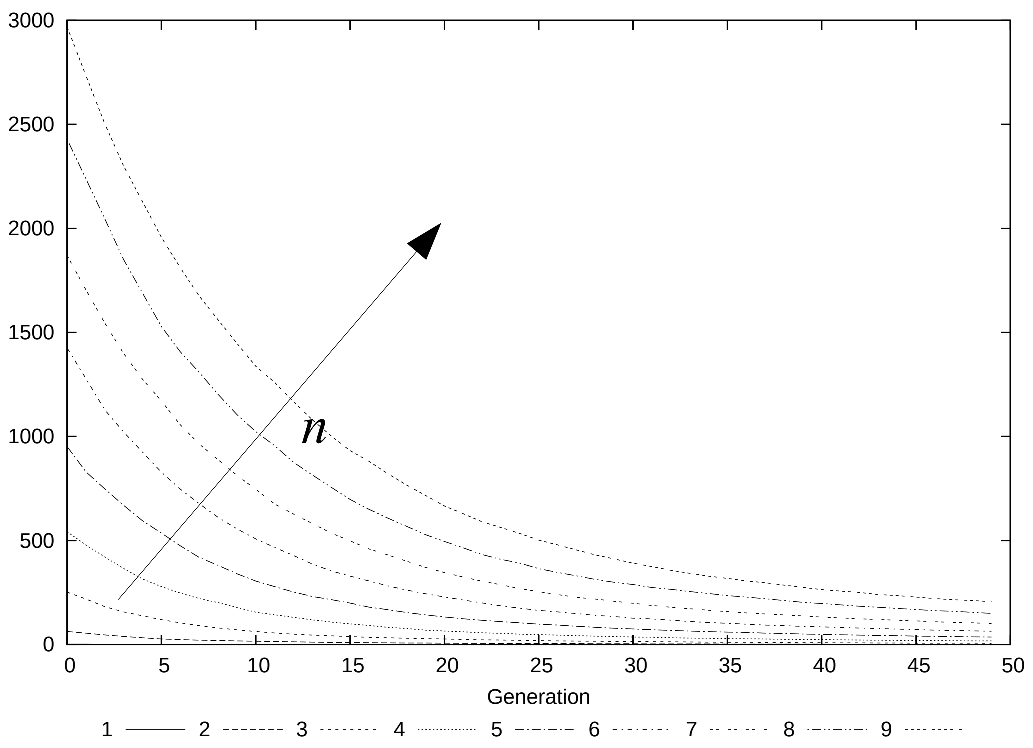
- Recombination is used: discrete sexual recombination for object parameters, intermediate panmictic recombination for strategy parameters.
- Mutation parameters: $\tau_0 = \frac{c}{\sqrt{2n}}$, $\tau = \frac{c}{\sqrt{2\sqrt{n}}}$, $c = 1$.
- Initial value of standard deviations σ: the smaller value of either $(x_i^u - x_i^l)/\sqrt{12}$ (assuming that the allowed range of object parameter x_i is x_i^l to x_i^u) or 3. Limiting the initial value is necessary because setting too high values can lead to the ES becoming unstable and incapable of converging on any optima.
- No lower bound for the σ.
- For fitness scaling, the scaling window algorithm with window size 5 was used.

6.2.4 Tests on the sphere model

The sphere model is a very simple test function on which any optimization procedure should be effective. Failing to find the optimum on this function is an indicator that there possibly is an error in the algorithm's implementation. If the algorithms are correctly implemented, tests on the sphere model allow to compare relative convergence speed.

Using the standard parameters, first the GA was used on the sphere model with varying numbers of dimensions (from $n = 1$ to $n = 9$). Because a single GA run can not be considered reliable evidence of performance due to the probabilistic nature of the algorithm, the GA was executed 1000 times so that the results could be averaged. The upper part of Fig. 6.10 shows the progression of the best (meaning lowest) objective value found in each generation for $n = 1$ to $n = 9$. It can be seen in the figure that the GA approaches the single global optimum with value zero gradually. The higher the dimension, the worse the level is at which the best values stagnate in the later generations.

The lower part of Fig. 6.10 shows the best solutions ever found during each of the 1000 runs for $n = 2$. The diagram is a top-down view for this three-dimensional case, with the X- and Y-coordinates of each point (represented as little crosses) signifying the two object parameters and the Z-coordinate being that point's objective value. The Z-coordinate is represented by drawing the crosses in varying shades of gray, with black being used for the lowest points and white for the highest ones (the scale on the right side of the diagram shows the respective objective values for the shades). While the solutions are generally clustered around the global optimum at the origin, the region in which each individual solution falls is fairly wide. The average distance of the points from the global optimum at the origin is 0.63, or 0.63% of the full range of each variable of 100 units (-60 to 40). The GA obviously has trouble pinpointing the best solution even at this low dimension and with a very easy test function.



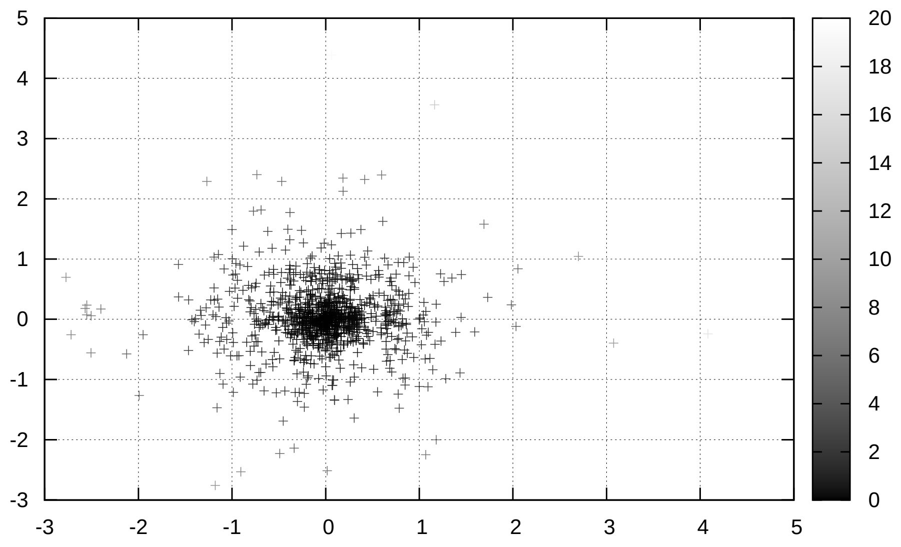


Figure 6.10: Progression of the average lowest objective value over 1000 runs in each generation when using a GA on the sphere model for $n = 1$ to $n = 9$, and best points found for $n = 2$

Next, the ES was applied to the sphere model using the standard settings (also 1000 times). The upper part of Fig. 6.11 shows the progression of the best (meaning lowest) objective value found in each generation for $n = 1$ to $n = 9$. Although the convergence becomes slower the higher the dimension is, just as in the case of the GA, the ES generally converges much faster for all dimensions. In contrast to the GA the ES also never stagnates at some non-optimal level but rather always reaches the best value of zero for all dimensions.

The lower part of Fig. 6.11 shows the best solutions ever found by the ES during each of the 1000 runs for $n = 2$. All solutions are very close to the global optimum at the origin, with the average distance being only $1.38 \cdot 10^{-8}$, many orders of magnitude better than the GA.

6.2.5 Tests on Ackley's function

Next, the tests were performed on Ackley's function using the GA and the ES with standard settings and the results averaged over 1000 runs each. The upper part of Fig. 6.12 shows the progression of the best (meaning lowest) objective value found in each generation for $n = 1$ to $n = 9$. Just as in the case of the sphere model, the GA approaches the global optimum at the origin with value zero gradually, but stagnates at some level which is worse the higher the dimension is.

The lower part of Fig. 6.12 shows the best solutions ever found by the GA during each of the 1000 runs for $n = 2$. The figure shows the best solutions clustered around the global optimum, but also around the local optima surrounding the global optimum. The average distance of the points is 0.44, or about 0.87% of the full range of 50 units for each variable.

The ES was then applied on Ackley's function 1000 times. The upper part of Fig. 6.13 shows the progression of the best (meaning lowest) objective value found in each generation for $n = 1$ to $n = 9$. The ES converges visibly faster than the GA and when it stagnates at some level in the higher dimensions, that level is much lower than the corresponding one for the GA.

The lower part of Fig. 6.13 shows the best solutions ever found by the ES during each of the 1000 runs for $n = 2$. All the solutions are very close to the global optimum, with an average distance of only $8.3 \cdot 10^{-9}$, again many orders of magnitude better than the GA as in the case of the sphere model. The ES can be said to have a much higher convergence reliability than the GA then.

6.2.6 Conclusion

Both algorithms were shown to be capable of finding the global optimum of the two test functions. For both functions the ES was found to be superior to the GA in convergence speed as well as convergence reliability. The implementations are

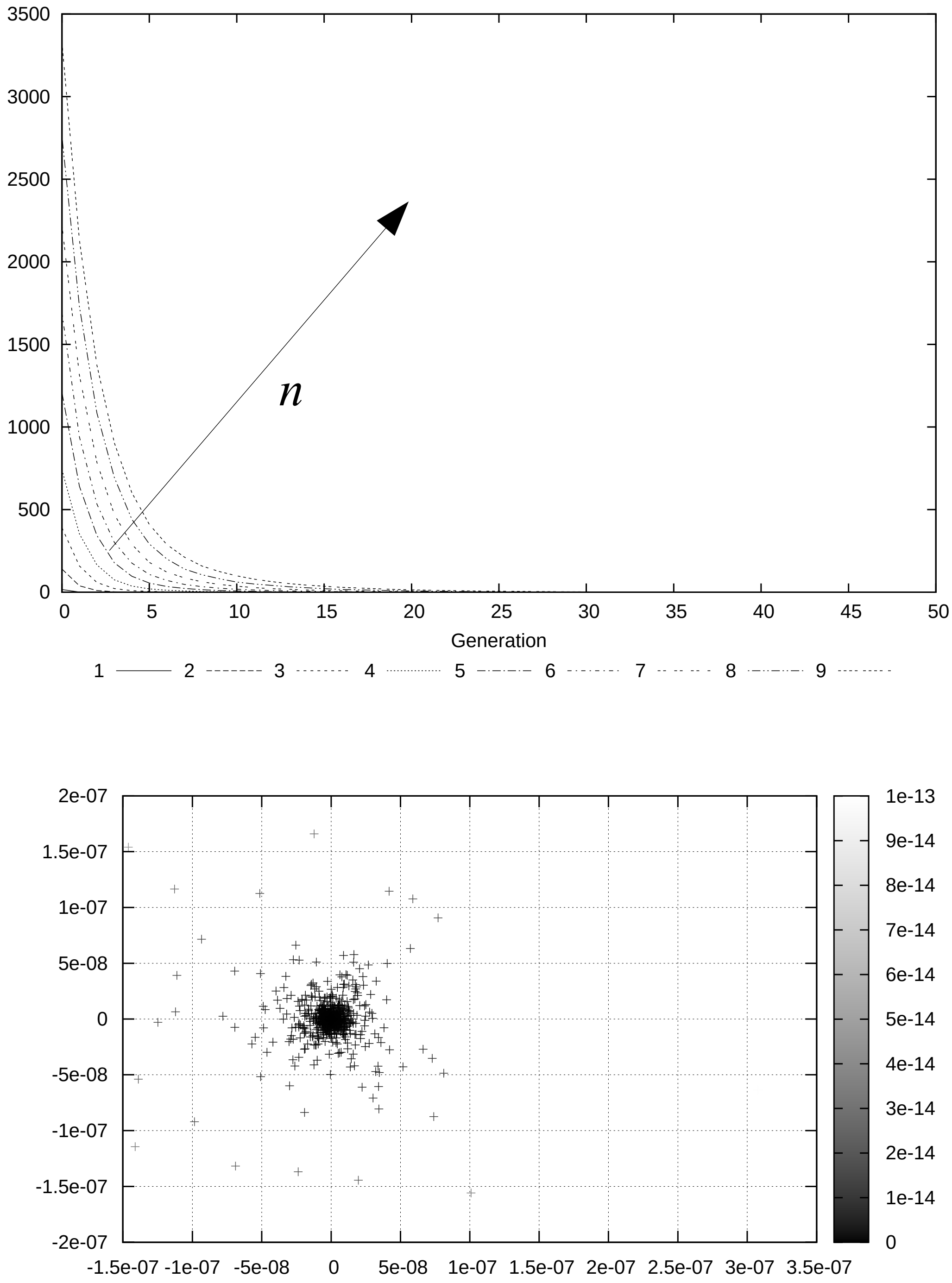
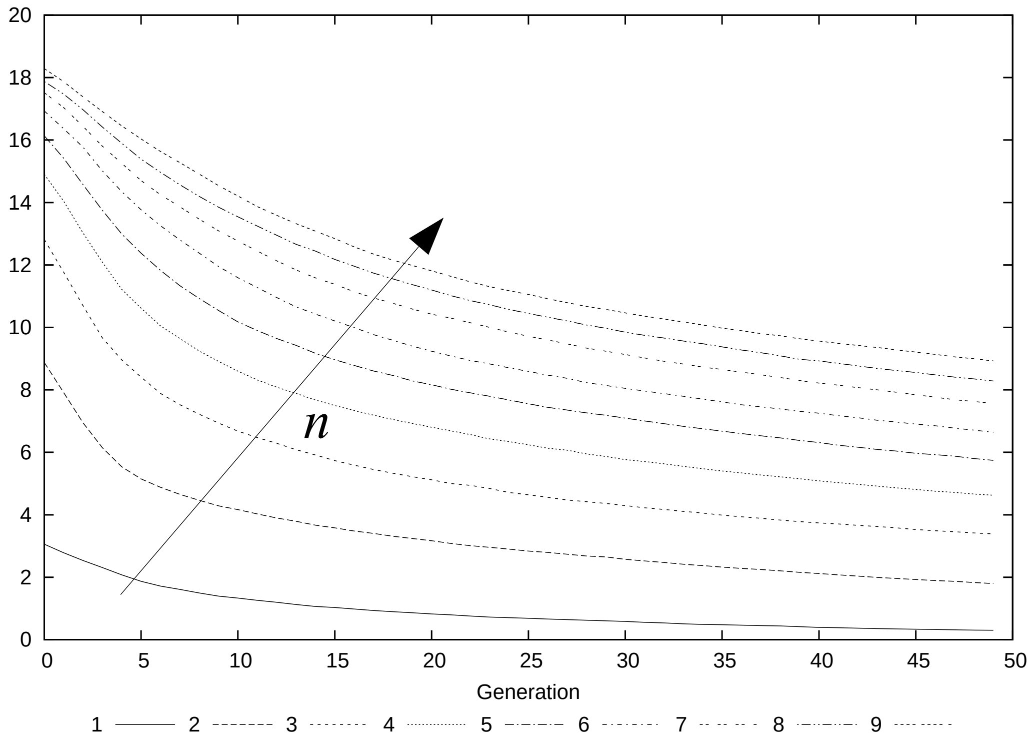


Figure 6.11: Progression of the average lowest objective value over 1000 runs in each generation when using an ES on the sphere model for $n = 1$ to $n = 9$, and best points for $n = 2$



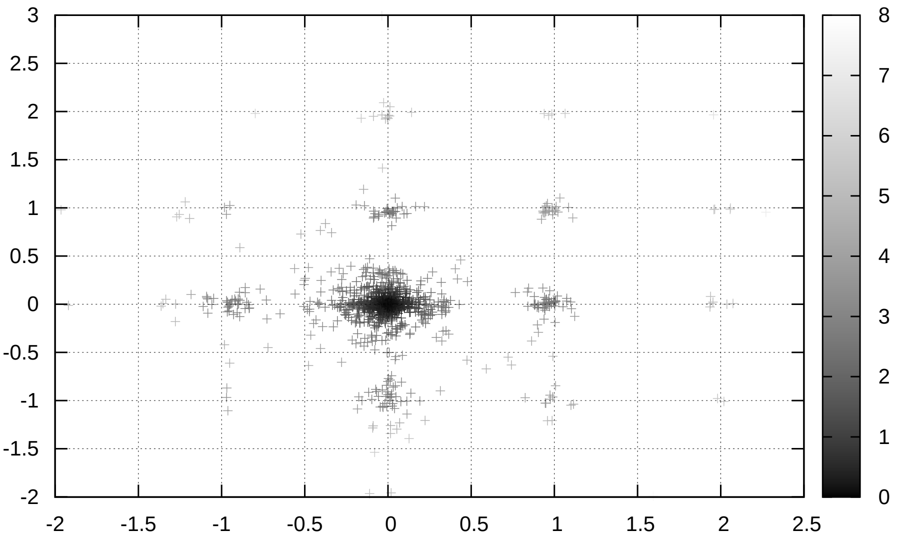
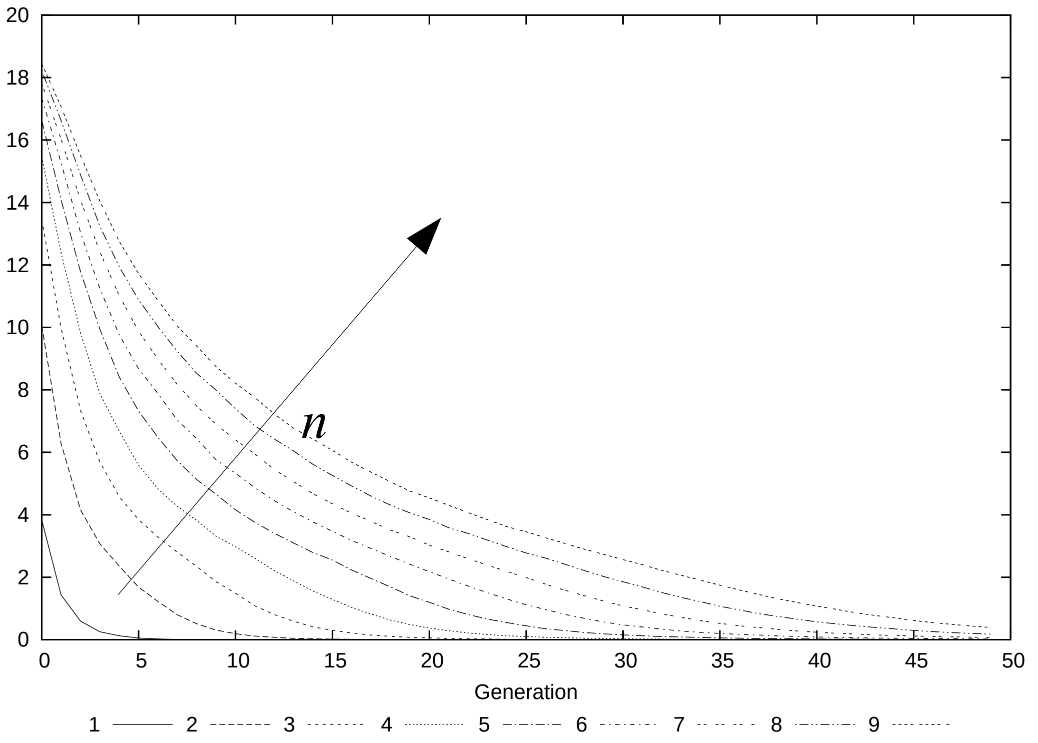


Figure 6.12: Progression of the average lowest objective value over 1000 runs in each generation when using a GA on Ackley's function for $n = 1$ to $n = 9$, and best points found for $n = 2$



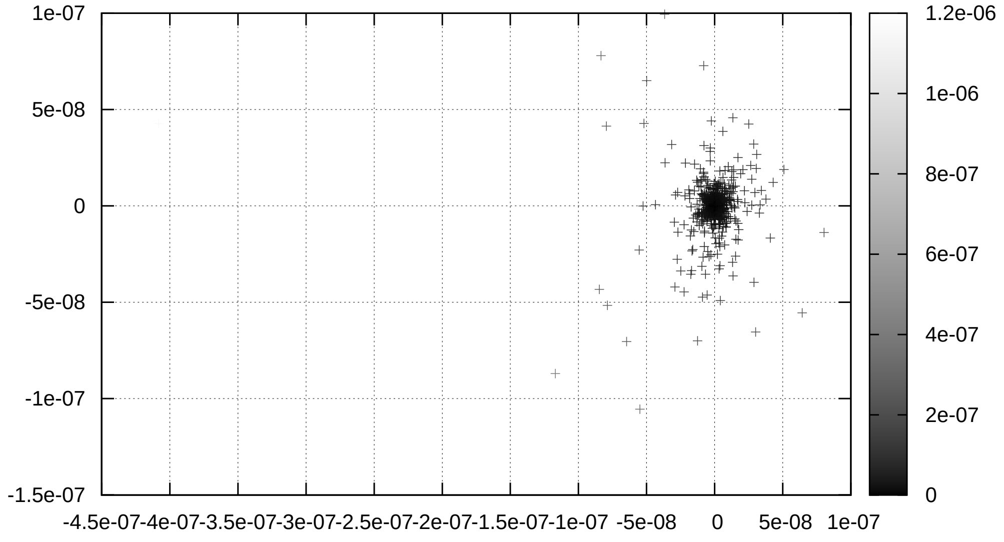


Figure 6.13: Progression of the average lowest objective value over 1000 runs in each generation when using an ES on Ackley's function for $n = 1$ to $n = 9$, and best points found for $n = 2$

therefore assumed to be correct, and Bäck's [Bäck96] findings that the ES is generally the best suited algorithm for numerical optimization tasks could be confirmed as well.

6.3 Effects of noise on the EAs

6.3.1 Objective of the experiment

Run-time measurements of programs are imprecise (see Sec. 6.1). When such a run-time is the objective function in an optimization problem, the optimization algorithm has to be able to deal with the random fluctuations without losing the ability to find an optimum. Experimental investigation [Beyer98] has shown:

> "Perhaps it may have appeared as a surprise, but the effects of fitness noise seem to be similar in GAs and ESs:
>
> (a) reduction of convergence velocity, and
>
> (b) deterioriation of the final optimum location quality ($R_\infty > 0$)
>
> Although this has been tested and quantified for the sphere model only, it should have been clear that the effects are of universal nature."

In [Nissen98] this was also tested empirically and confirmed:

> "The population-based GA and ES showed a remarkable robustness at all noise levels. In particular, for medium and high levels of noise, even a sample size of one frequently produced better results than a sample size of ten for the point-based techniques."

"Sample size" in the previous quote refers to the number of objective function evaluations that are performed for each individual. Increasing the sample size and averaging over the results provides a way to determine the actual objective value despite noise, but obviously this is very costly due to the increased computational effort. For a costly operation like a SAT solver run, the sample size should be kept as small as possible, ideally at 1.

In this experiment it will be attempted to confirm the results of [Nissen98] for the ES and GA implementation used in this work on the sphere model and on Ackley's function with a certain amount of noise. The noise in [Nissen98] was modeled by adding a normally distributed random value to the objective function value whose standard deviation was set to a percentage of the average expected value of the objective function. If $F(\vec{x})$ is the objective function, the noisy objective function is:

$$\Phi(\vec{x}, \sigma) = F(\vec{x}) + N(0, \sigma) \tag{6.3}$$

The standard deviation σ of the noise (assuming that the global minimum of $F(\vec{x})$ is 0) is defined as:

$$\sigma = \alpha \cdot \bar{m} \tag{6.4}$$

Dimension n	Sphere $\bar{m}$	Ackley $\bar{m}$
1	933.123	17.491
2	1867.01	19.3724
3	2798.75	19.945
4	3734.67	20.2002
5	4665.53	20.338
6	5601.14	20.4232
7	6532.93	20.4828
8	7462.97	20.5244
9	8400.39	20.557

Table 6.1: Average values of 2,500,000 random evaluations of the sphere model and Ackley's function

The parameter α determines the noise level. The values tested in [Nissen98] were $\alpha \in \{0.2\%, 0.5\%, 2\%, 5\%, 10\%\}$. The value $\bar{m}$ is the average objective value when evaluating a large number N of random solutions $\vec{x}$ from the search space S:

$$\bar{m} = \frac{1}{N} \sum_{j=1}^{N} F(\vec{x_j}); \vec{x_j} \in S, \vec{x_j} \text{ chosen randomly} \tag{6.5}$$

The test functions to be used will be the sphere model and Ackley's function. Tab. 6.1 shows the values of $\bar{m}$ that were determined for each function depending on the dimension n averaged over 2,500,000 random points (since the tests will be executed for 1000 runs, with 2500 objective function evaluations each).

The noise level chosen for the experiments was $\alpha = 5\%$ (meaning a value of $\alpha = 0.05$) since it was determined in the run-time variation experiments that in the expected environment (Linux in single user mode or under low load) the imprecision of run-time measurements stays under 5%.

6.3.2 Tests on the sphere model

The GA and ES were used to find the global minimum of the sphere model with noise added for 1000 runs, with the number of dimensions ranging from 1 to 9. The upper parts of Fig. 6.14 and Fig. 6.15 show the progression of the average best (lowest) objective value per generation over all runs for the GA and the ES respectively. The lowest objective values in this experiment reach negatives despite both test functions being non-negative everywhere because of the Gaussian noise, which can take on positive as well as negative values. The ES converges visibly faster than the GA.

The lower parts of Fig. 6.14 and Fig. 6.15 show the best points found in each run of the GA and ES for the case $n = 2$. The general area in which the result points are found is roughly the same for both algorithms inside the range $-5 \leq x_1 \leq 5$ and $-5 \leq x_2 \leq 5$. In this range the maximum possible objective function value is $x_1^2 + x_2^2 = 5^2 + 5^2 = 50$, and the standard deviation of the noise is $\sigma = \alpha \cdot \bar{m} = 0.05 \cdot 1867.01 = 93.3505$. Since the maximum objective function value in this area is only about half of the standard deviation of the noise it is not surprising that neither algorithm could narrow the optimum down any further.

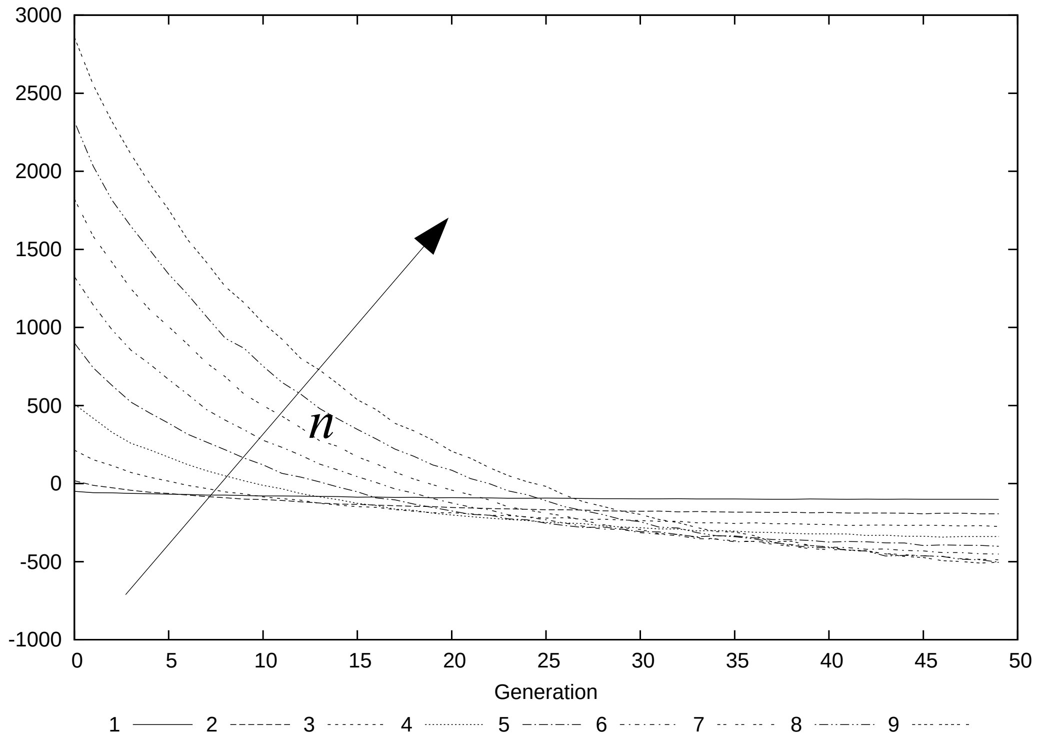
6.3.3 Tests on Ackley's function

The upper parts of Figs. 6.16 and 6.17 show the progression of the average best (lowest) objective value per generation over 1000 runs for the GA and the ES respectively. As in the case of the sphere model, the ES converges visibly faster than the GA for all dimensions. Additionally here the ES also finds better optima on average since the asymptotic values reached at the end of the runs are lower than those found by the GA.

The lower parts of Figs. 6.16 and 6.17 show the best points found in each run of the GA and ES for the case $n = 2$. While the GA found many points that are clustered around the global minimum, there are also many points which are clearly nearer to any of more than ten different local minima. The ES on the contrary found always found the global minimum and the points are very close to it.

6.3.4 Conclusion

As in the case of the tests without noise, both algorithms were shown to be capable of finding the global optimum of the two test functions. The ES was found to be superior to the GA in convergence speed as well as convergence reliability even when the objective function evaluation is noisy.



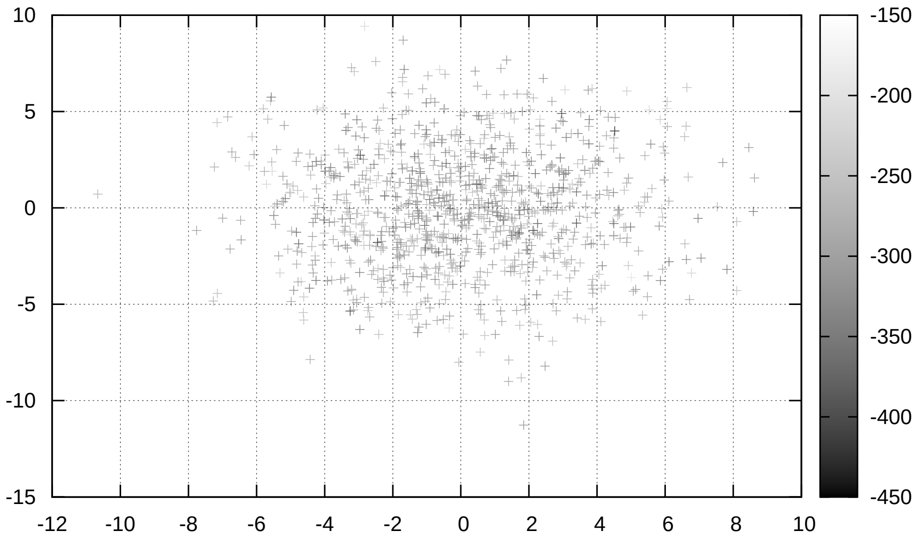
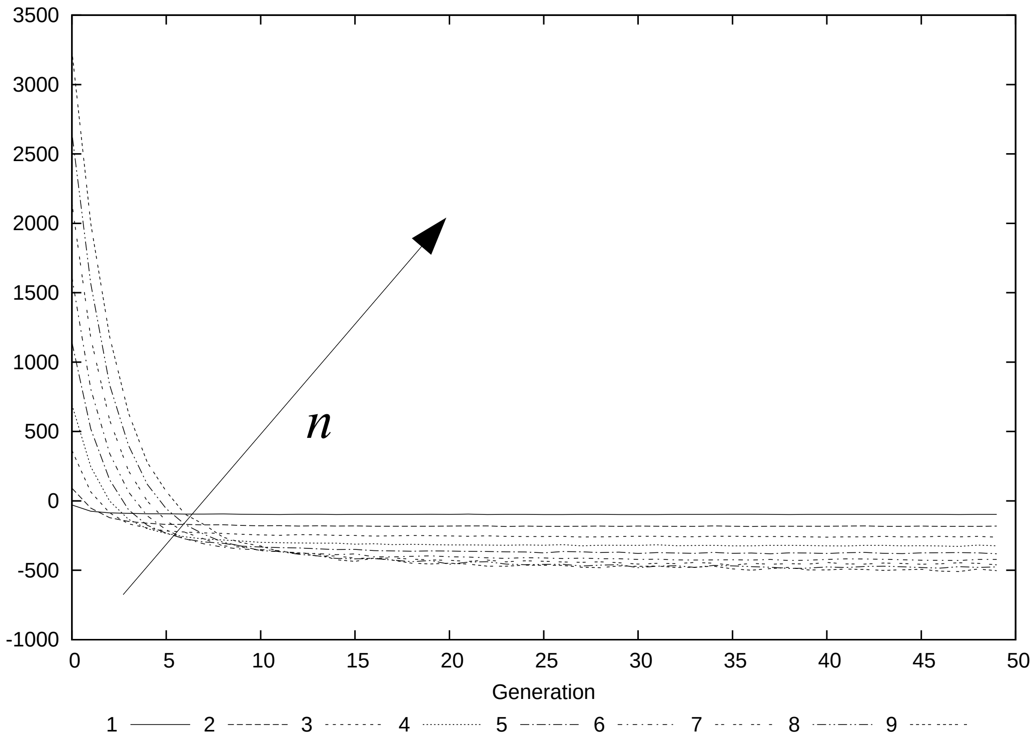


Figure 6.14: Progression of the average lowest objective value in each generation over 1000 runs when using a GA on the sphere model with 5% Gaussian noise added for $n = 1$ to $n = 9$, and best points found for $n = 2$



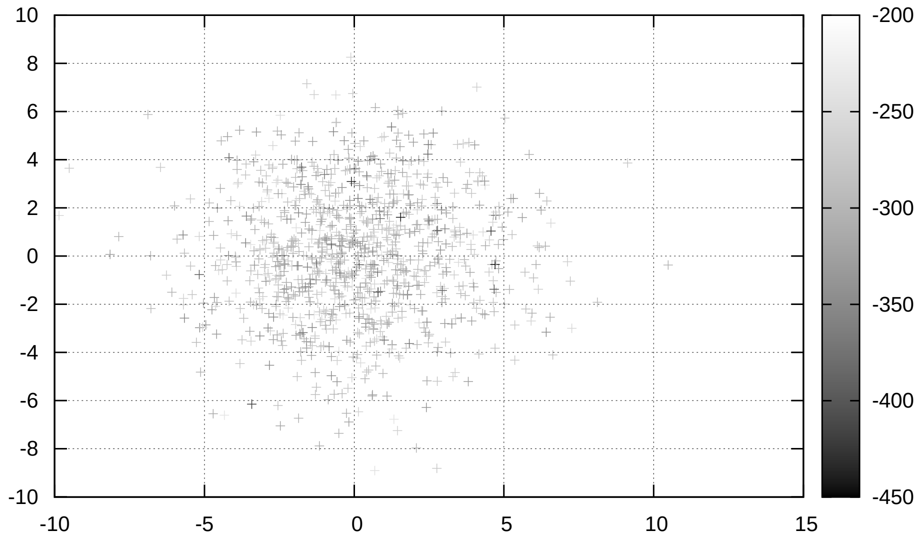
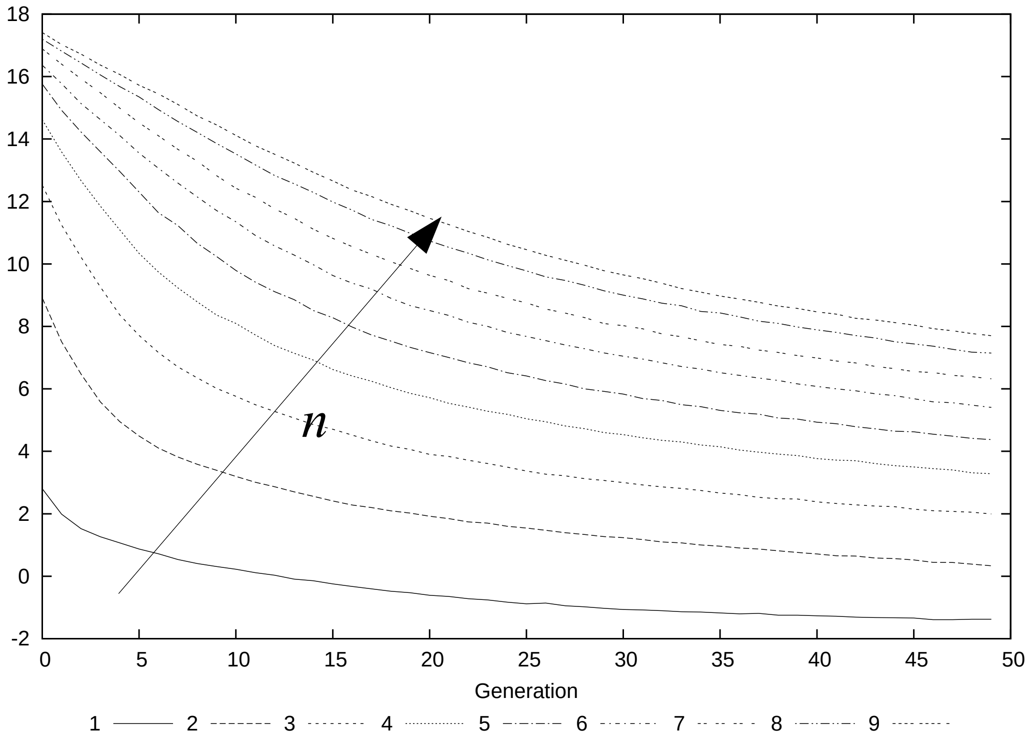


Figure 6.15: Progression of the average lowest objective value in each generation over 1000 runs when using an ES on the sphere model with 5% Gaussian noise added for $n = 1$ to $n = 9$, and best points found for $n = 2$



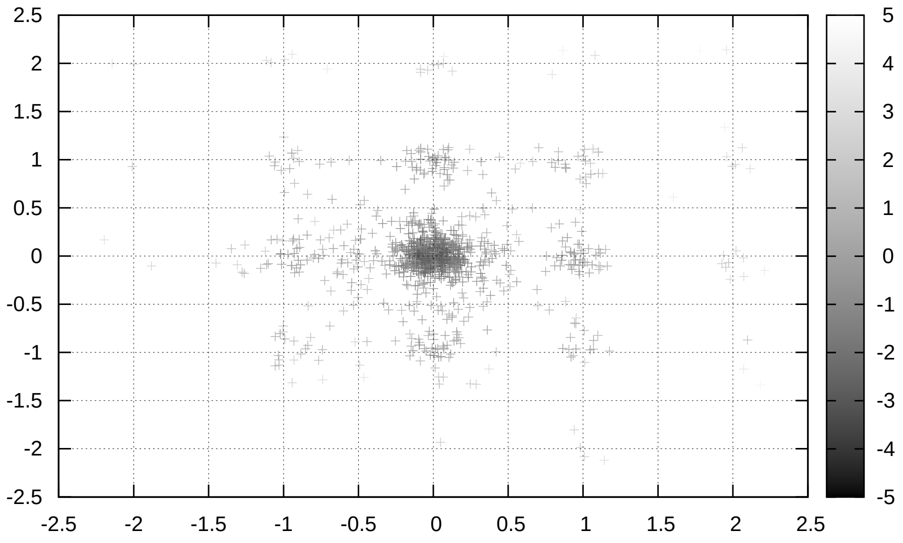
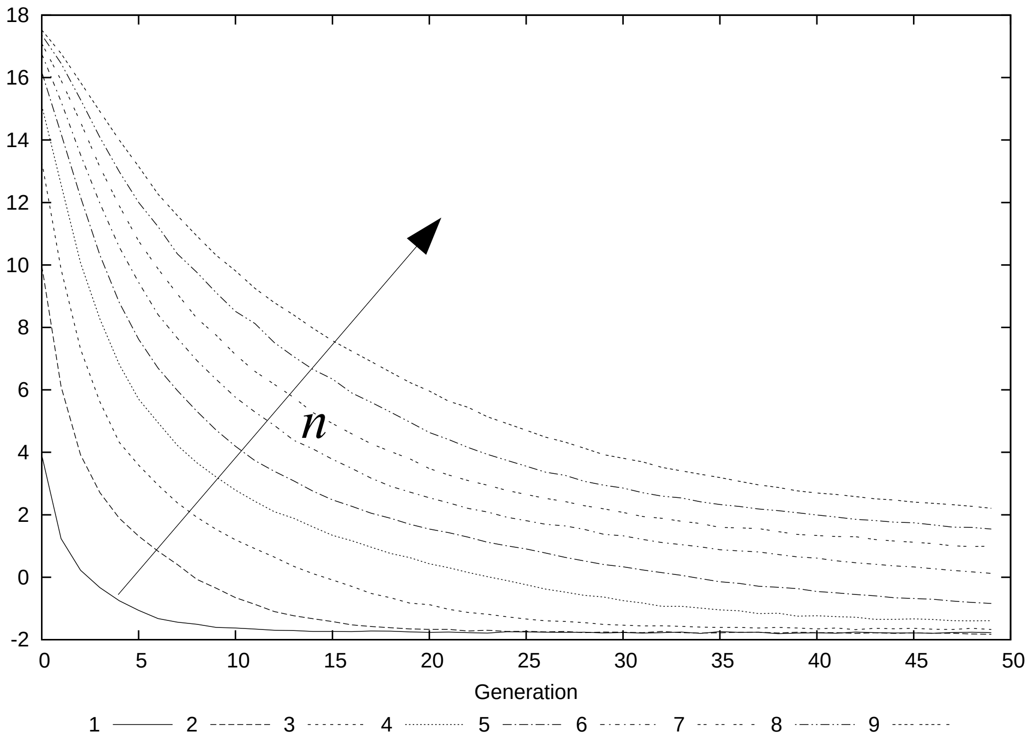


Figure 6.16: Progression of the average lowest objective value in each generation over 1000 runs when using a GA on Ackley's function with 5% Gaussian noise added for $n = 1$ to $n = 9$, and best points found for $n = 2$



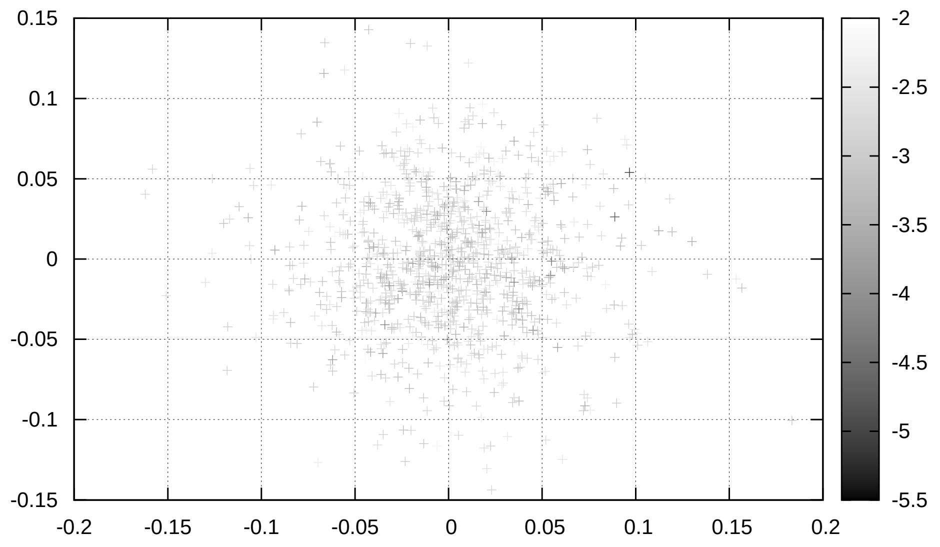


Figure 6.17: Progression of the average lowest objective value in each generation over 1000 runs when using an ES on Ackley's function with 5% Gaussian noise added for $n = 1$ to $n = 9$, and best points found for $n = 2$

6.4 Fitness landscapes

6.4.1 Objective of the experiment

A fitness landscape is a visualization of the search space of an optimization problem where optima are shown as peaks (maximization problem) or valleys (minimization problem) on a three-dimensional map (see for example Fig. 1.3 in Sec. 1.5). Such a map is useful to get a general idea of what kind of problems an optimization algorithm will encounter (for example, it shows how multimodal the search space is). A number of fitness landscapes for the SAT solver optimization problem using different parameter combinations were constructed using various SAT problems to try and gain some insight into what kind of behavior can be expected from the EAs.

6.4.2 Setup

To plot a fitness landscape two of the numerical parameters of the optimization problem have to be selected, as well as a region of interest for their values (meaning upper and lower limits for each parameter). One of the parameters forms the X axis of the fitness landscape and the other the Y axis, and the objective values of the optimization problem (in this case, the runtime of the SAT solver measured in seconds) form the Z axis of the landscape. The set of possible optimization parameter combinations forms a rectangle in the two-dimensional XY-plane, with the lower and upper limits of each axis variable being the sides. Inside this rectangle, some number of data points can be sampled. To get a "landscape" rather than just a collection of points, some type of interpolation can be used to approximate the area between the points. The more data points are computed, the more detailed the landscape will be.

In this work the fitness landscapes were created by putting a regular grid of $N \times N$ points over the rectangular search space, then running MINISAT on a SAT problem with the parameters described by the grid points (all other parameters were held at their default values). The measured run times for each parameter combination are the Z components of data points in the fitness landscape. If a problem can not be solved within a predetermined time-out period, then the respective data point is assigned a solving time of twice the time-out period (equivalent to how the objective function was computed in such cases for the EAs later). A number of (unsatisfiable) industrial SAT problems from the benchmark set of the 2006 SAT Race first qualification round [Sinz06] were chosen as the basis for the fitness landscape experiments.

The open-source graph software GNUPLOT[1] was used to generate three-dimensional images from the gathered data points. The landscapes are represented as "wiremesh" surfaces where each data point is at the meeting point of the wires (linear interpolation between the data points). The wiremesh diagrams are of course only an

[1]Homepage: http://www.gnuplot.info/

approximation of what the fitness landscape “really” looks like, since time measurements are noisy, and additionally the area between data points is represented here by simple planes where in reality it is likely much less smooth. Higher grid resolutions give a more detailed image, but since the number of run times to measure rises with the square of the number of subdivisions per axis, at most about 100 points per axis (yielding about 10000 data points per landscape) were used.

For each fitness landscape the resolution of the grid used (subdivisions per axis), the time-out and the name of the machine the data points were computed on (see the following section) is given in the title. In the upper left corner of all fitness landscapes “Min.: $(X/Y/Z)$, Max.: M” states the X- and Y-coordinates of the point with the lowest computed run time Z, and M is the highest run time encountered (which is the “timed out” value in some cases). Additionally the minimum point is marked by a filled black circle in the landscape. The point in the landscape defined by the default values of each parameter in MINISAT (with a Z value of 0) is marked by a filled black triangle; the default values are additionally also noted in the upper left corner of the figures as “Default (*default X*/*default Y*)”.

6.4.2.1 Runtime normalization

Generating a fitness landscapes can take a large amount of time depending on the grid resolution and time out period. For example, assuming a grid of 100×100 points (10000 total) and an average run time per data point of 5 minutes, the entire landscape would take about 34 days to compute. The landscapes in this work were usually completed in shorter times, but still took too long to repeat them arbitrarily often. To save time the experiments were conducted on several different computers concurrently (and generally out of order, whenever machines became free), though of course all run times for a single fitness landscape were taken on the same machine. Tab. 6.2 shows the hardware configurations of the 6 different types of machine that were used. **`Pc0`** was the slowest machine, **`Pc1`** was faster than **`Pc0`** and so on. The designation of the machine used for each landscape is indicated in the title of the diagram.

The available computers had different architectures and speeds, which of course leads to different solving times being measured for the same problem and solver configuration. The usefulness of the landscapes would be questionable if they looked entirely different for the same problem on different computers, but fortunately it appeares that solving times are more or less a linear function of a computer’s “power”. Tab. 6.3 shows the measured run times when solving the problems used for the fitness landscapes with MINISAT using standard parameters. These times were measured while running in single-user mode under Linux to minimize the imprecision. All fitness landscapes were computed with no concurrent non-essential tasks running as well.

The computer **`Pc1`** was the first dedicated machine that was available for experi-

Designation	Hardware
Pc0	Intel®Pentium®4@2.4GHz / 512KB cache
Pc1	Intel®Pentium®4@3.2GHz / 2MB cache
Pc2	Intel®Core™2 4300@1.8GHz / 2MB cache
Pc3	Intel®Core™2 6320@1.86GHz / 4MB cache
Pc4	Intel®Core™2 Duo E6550@2.33GHz / 4MB cache
Pc5	Intel®Core™2 Duo E8400@3GHz / 6MB cache

Table 6.2: Hardware configurations of the computers used

ments and was therefore chosen as the "reference" machine. If t_{ref} is the solving time for some problem on the reference machine and t_{other} the solving time on a different computer, the *normalization factor* c_{norm} is:

$$c_{\text{norm}} = \frac{t_{\text{other}}}{t_{\text{ref}}} \tag{6.6}$$

Tab. 6.3 shows the median, average and standard deviation (in % of the average) of all the normalization factors; it can be seen that the factors c_{norm} remain at fairly similar values for each machine across the problems. It is therefore assumed that landscapes generated on different machines are essentially equivalent apart from a scaling factor for the "heights" of the data points and some amount of noise due to imprecise time measurements and other factors such as different CPU architectures, cache sizes and memory speeds whose effects can not be quantified easily.

The knowledge of the relative speeds of different computers used for the experiments was also used in this work to allow roughly the same amount of "searching" per evaluation with the DPLL algorithm in the optimization experiments with EAs: the timeouts were adjusted up or down depending on if a computer was slower or faster compared to a reference machine. Compared to the timeout period T_{ref} on the reference machine the timeout T_{other} was adjusted to be:

$$T_{\text{other}} = c_{\text{norm}} \times T_{\text{ref}} \tag{6.7}$$

Problem	**Pc1**	**Pc0**	c_{norm}	**Pc2**	c_{norm}	**Pc3**	c_{norm}	**Pc4**	c_{norm}	**Pc5**	c_{norm}
`manol-pipe-c7_i`	10.7	23.7	2.21	9.2	0.85	7.5	0.69	6.1	0.57	3.9	0.36
`velev-npe-1.0-02`	15.9	33.4	2.09	13.6	0.85	11.4	0.71	9.3	0.58	6.4	0.4
`velev-eng-uns-1.0-04a`	42.5	92.1	2.17	36.2	0.85	28.7	0.68	24.0	0.56	14.6	0.34
`manol-pipe-c6n`	47.3	104.0	2.2	41.2	0.87	32.7	0.69	27.4	0.58	14.5	0.31
`goldb-heqc-desmul`	50.0	107.6	2.15	47.4	0.95	35.6	0.71	30.2	0.6	20.2	0.4
		med.	2.17		0.85		0.69		0.58		0.36
		avg.	2.17		0.88		0.7		0.58		0.36
		stdev.%	2.15%		4.66%		2.27%		2.63%		11.01%

Table 6.3: Runtimes for fitness landscape problems using standard MINISAT on different machines

6.4.2.2 Optimization parameter set

In the SAT solver optimization problem in this work there are 7 parameters to choose from for the fitness landscapes. To optimize a real-valued variable with an EA an allowed range has to be decided into which it may fall, which in the current case may only contain values which make sense as settings for the SAT solver. Tab. 6.4 shows the 7 MINISAT parameters to be optimized, including their default values as well as the lower and upper limit of values between (and including) which the EAs will search. Following list summarizes the function of the parameters:

1. $\mathrm{VARDECAY}$ (see Sec. 3.3.8): *inverse* of the multiplication factor with which the variable activity "bumping" amount is multiplied periodically to simulate an aging of the activities. The multiplication factor must be greater than 1 (or in the borderline case where there is no aging equal to 1) for the aging mechanism to work: $\frac{1}{\mathrm{VARDECAY}} \geq 1$. Therefore $\mathrm{VARDECAY}$ has to be positive, not zero, and less than or equal to 1, making its possible range $0 < \mathrm{VARDECAY} \leq 1$. The closer to 0 $\mathrm{VARDECAY}$ is, the stronger the aging effect becomes. Conversely, the closer to 1 the parameter is, the weaker the aging effect is, with no aging at all when it is equal to 1. The lower limit was arbitrarily set to 0.01, which gives a maximum increase factor of up to 100, which is much higher than the default aging factor and should give a more than sufficient range to explore.

2. $\mathrm{CLAUSEDECAY}$ (see Sec. 3.6.1): works in a very similar way as $\mathrm{VARDECAY}$, but it controls the aging of learned clauses. The possible range is $0 < \mathrm{CLAUSEDECAY} \leq 1$, and 0.01 was chosen as the lower limit for this parameter.

3. $\mathrm{NOFCONFLICTSINC}$ (see Sec. 3.6): multiplication factor for MINISAT's geometric series restart strategy. Must be ≥ 1, where value 1 means that restarts are periodically triggered after a constant number of conflicts. The upper limit was chosen as 2, which was considered high enough due to the very quick (exponential) rise of the number of conflicts and relatively rare use of the parameter.

4. $\mathrm{NOFCONFLICTSBASE}$ (see Sec. 3.6): base number of conflicts for MINISAT's geometric series restart strategy. Must be any positive number; 10 and 1000 were arbitrarily chosen as limits.

5. $\mathrm{NOFLEARNTSINC}$ (see Sec. 3.6.1): after every restart the limit of learned clauses that are allowed for the current search run is multiplied with this parameter. Must be ≥ 1, upper limit was chosen as 2.

	Variable	Lower limit	Upper limit	Default value
1	VARDECAY	0.01	1.0	0.95
2	CLAUSEDECAY	0.01	1.0	0.999
3	NOFCONFLICTSINC	1.0	2.0	1.5
4	NOFCONFLICTSBASE	10.0	1000.0	100
5	NOFLEARNTSINC	1.0	2.0	1.1
6	NOFLEARNTSDIVISOR	0.1	10.0	3.0
7	RANDOMVARFREQ	0.0	0.5	0.02

Table 6.4: The MINISAT parameters to optimize, with their chosen ranges and default values

6. NOFLEARNTSDIVISOR (see Sec. 3.6.1): the initial number of allowed learned clauses is the number of original CNF clauses divided by this parameter; 0.1 and 10 were arbitrarily chosen as limits.

7. RANDOMVARFREQ (see Sec. 3.3.8): with a probability defined by this parameter MINISAT sometimes chooses a random variable. The value 0 means no random decisions at all and is the lower limit. The upper limit was chosen as 0.5, meaning half the decisions are random; higher values were considered not useful.

Of the 7 values, 6 were paired as fitness landscape axis variables due to the similarity of their function or because they affect the same part of the solver. VARDECAY and CLAUSEDECAY affect different parts of the solver (decision heuristic and learned clause deletion), but work in a similar way (both are “decay” constants). They also have very similar possible ranges.

NOFCONFLICTSINC and NOFCONFLICTSBASE both affect the restart policy of the solver and are a natural choice for pairing as axis variables. Here the ranges are very different.

At last, NOFLEARNTSINC and NOFLEARNTSDIVISOR control the solver’s learned clause deletion mechanism (along with CLAUSEDECAY) and were considered a good choice to pair. Their ranges differ by one order of magnitude.

RANDOMVARFREQ was charted alone by itself.

6.4.3 VARDECAY / CLAUSEDECAY

For the parameter pair $\mathrm{VARDECAY}$ and $\mathrm{CLAUSEDECAY}$ in addition to the full range fitness landscape ($0.01 \leq \mathrm{VARDECAY} \leq 1$ and $0.01 \leq \mathrm{CLAUSEDECAY} \leq 1$) a close-up of the region $0.9 \leq \mathrm{VARDECAY} \leq 1$ and $0.9 \leq \mathrm{CLAUSEDECAY} \leq 1$ that contains the default values was generated.

Additionally, full range fitness landscapes of 9 shuffled versions of each SAT problem were computed, though with a lesser number of grid points to keep computation times acceptable. For the shuffled problems in this experiment, the seeds were 100001 to 100009. Fig. 6.19 to Fig. 6.28 show the computed fitness landscapes. Each double page contains the landscapes for one of the test problems.

6.4.3.1 Observations

1. It can be seen that all the fitness landscapes are highly multimodal: the surfaces are rough, with many peaks and valleys.

2. Despite the roughness there is a visible downward trend of the landscapes in the directions of $\mathrm{VARDECAY} = 1$ as well as $\mathrm{CLAUSEDECAY} = 1$. Also noticeable is that the solving time jumps up noticably or the solver even times out at exactly $\mathrm{VARDECAY} = 1$ in some cases.

3. The landscapes contain ridge-shaped structures of long valleys extending along the $\mathrm{CLAUSEDECAY}$ axis. The ridges are not completely level but rather they slowly slope downwards in the direction of $\mathrm{CLAUSEDECAY} = 1$.

4. The lower-resolution landscapes of the shuffled problems are fairly similar to each other and the high resolution landscape of the original problem. The minimum points are always near $\mathrm{VARDECAY} = 1$ and $\mathrm{CLAUSEDECAY} = 1$.

6.4.3.2 Interpretation

1. The roughness of the fitness landscapes is not unexpected, due to the complexities of the interplay between the various elements of the solver. A single different decision can lead the solver down a very different area of the search space, leading to potentially very different solving times for two similar parameter configurations.

2. The solving times seem to get better the closer to the value 1 both parameters $\mathrm{CLAUSEDECAY}$ and $\mathrm{VARDECAY}$ are, though $\mathrm{VARDECAY}$ should not be exactly 1. The decay parameters control the aging of variable and clause activities. The more quickly variables age, the more focused on variables that occured in recent conflict clauses the search becomes. The more quickly learned

clauses age, the more likely they become to be deleted early. Very high decay strengths (parameter values near 0) are apparently not useful. In the case of clause decay, very high values would mean that a lot of learned clauses get deleted quite frequently, leading to a high loss of information and a therefore a less efficient search. The drawback of very high variable decay is less clear, but the need to more frequently having to rescale all activities once a maximum is exceeded may be one factor. Turning off the variable decay completely (equal to 1) leads to high solving times or timeout in some cases, indicating that the presence of this feature is important for the solver performance.

3. The ridge structures indicate that the solving time varies less strongly with changes of $\mathrm{CLAUSEDECAY}$ than of $\mathrm{VARDECAY}$. Presumably the solving time reacting more strongly to changes in $\mathrm{VARDECAY}$ is due to the variable activities coming into play every time a decision is needed (which is very often) while the clause activities are only relevant when the learned clause database is cleaned up (which happens much less frequently).

4. For shuffled problems the solving time can be different from the original problem using some set of parameters, but generally the best choice of parameter is still somewhere near $\mathrm{VARDECAY} = 1$ and $\mathrm{CLAUSEDECAY} = 1$.

Fig. 6.18 shows a summary of all minimum points founds in the landscapes along with their averages, medians, minimum and maximum value found and standard deviations in percent. The figure also visualizes the best points in a XY-diagram. It can be seen that according to the fitness landscapes values that are very near to the default values used in MINISAT ($\mathrm{VARDECAY} = 0.95$ and $\mathrm{CLAUSEDECAY} = 0.999$) are optimal.

Best points found in the full range in original problem and 9 shuffled versions:

	VARDECAY	CLAUSEDECAY
Min.:	0.736	0.934
Max::	1.000	1.000
Med.:	0.934	1.000
Avg.:	0.910	0.987
Stddev. %:	5.74%	2.70%

Best points found in close-ups:

	VARDECAY	CLAUSEDECAY
Min.:	0.902	0.970
Max::	0.980	1.000
Med.:	0.934	0.992
Avg.:	0.940	0.988
Stddev. %:	3.17%	1.26%

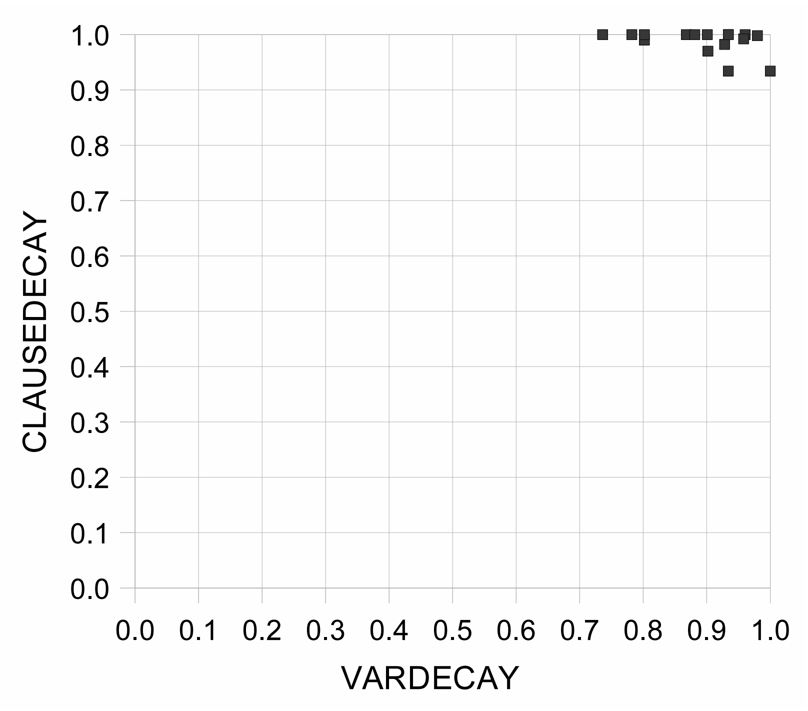
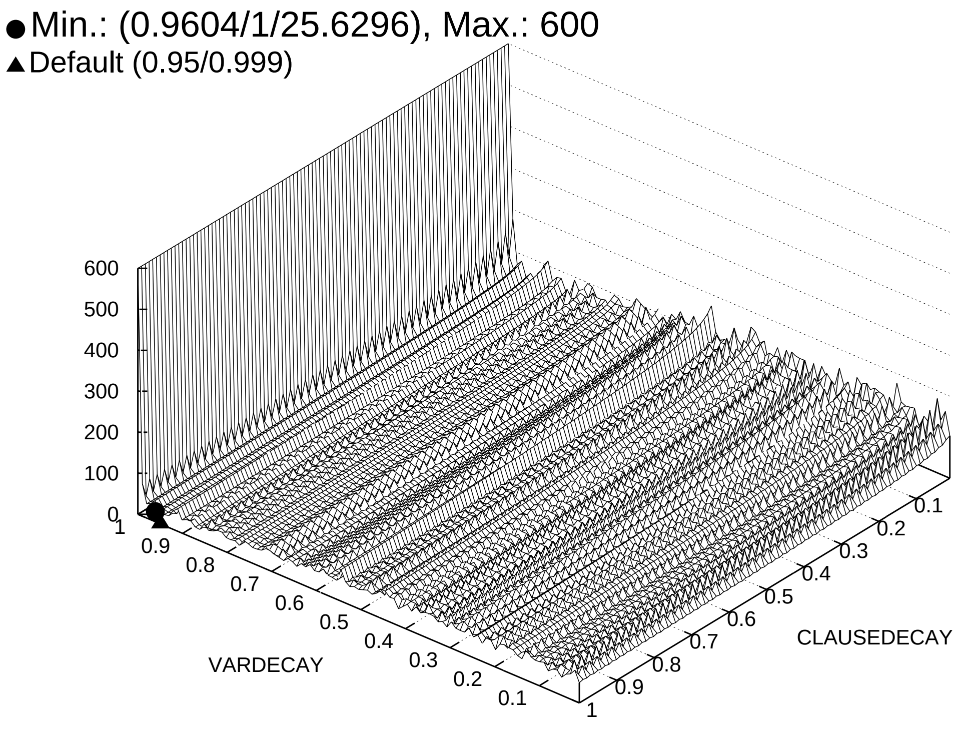


Figure 6.18: Analysis and visualization of all best points found in fitness landscapes for VARDECAY / CLAUSEDECAY



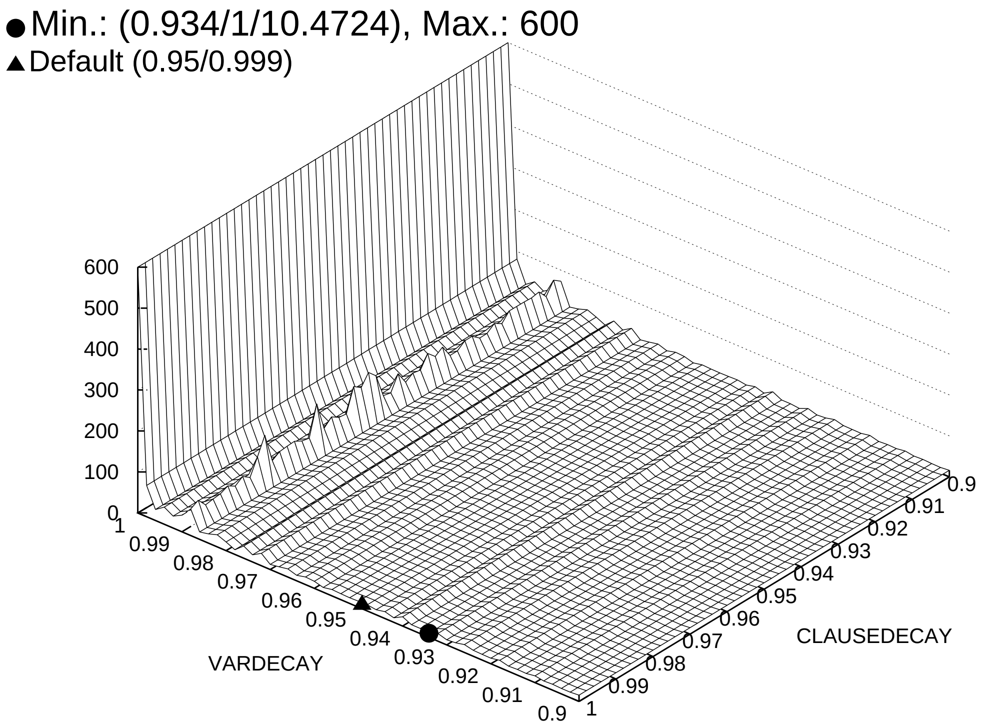


Figure 6.19: Fitness landscapes of problem `manol-pipe-c6n` for VARDECAY / CLAUSEDECAY using the full range (top, 101×101 grid, timeout 300s, **Pc2**) and a close-up (bottom, 51×51 grid, timeout 300s, **Pc5**)

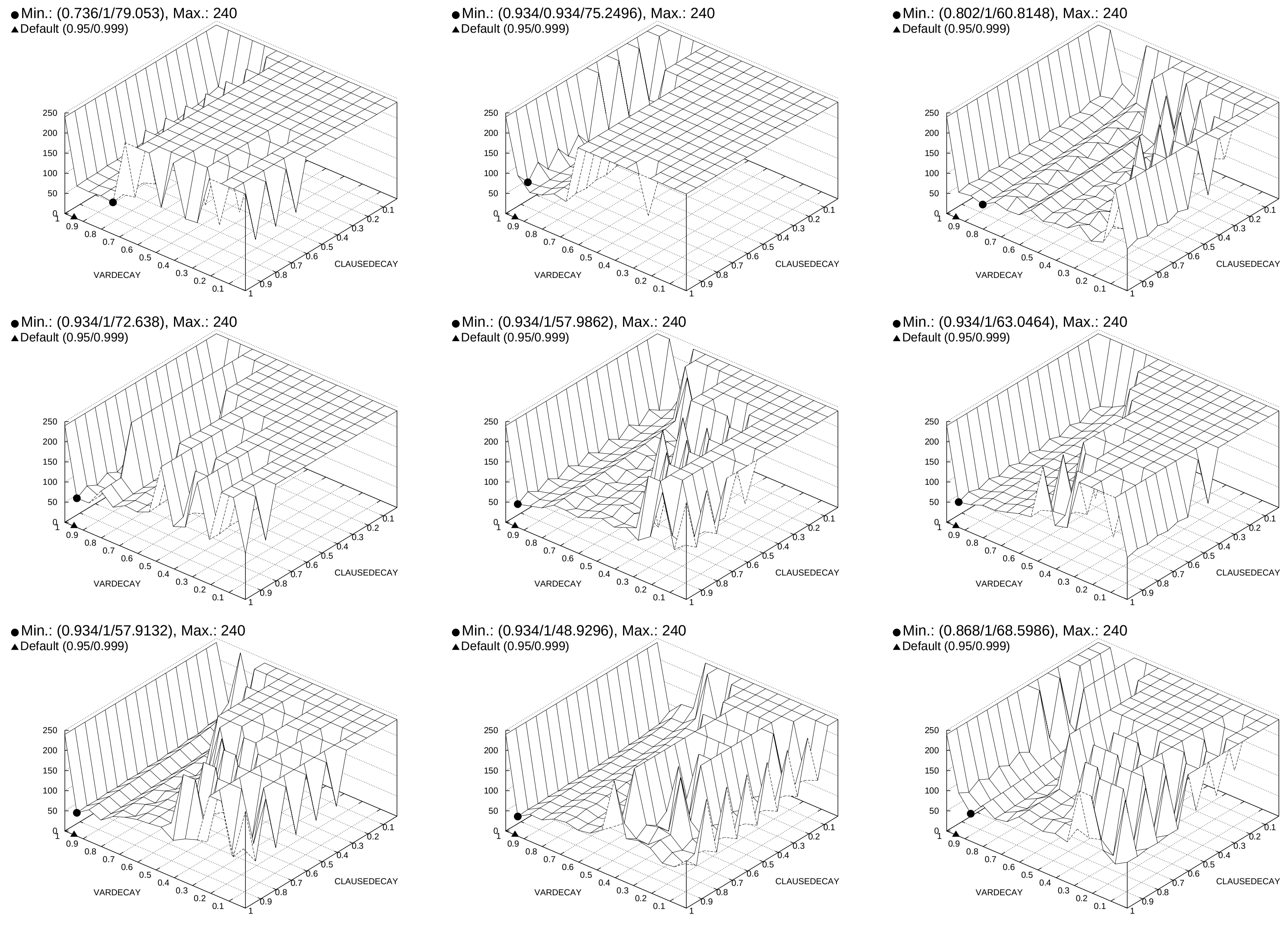
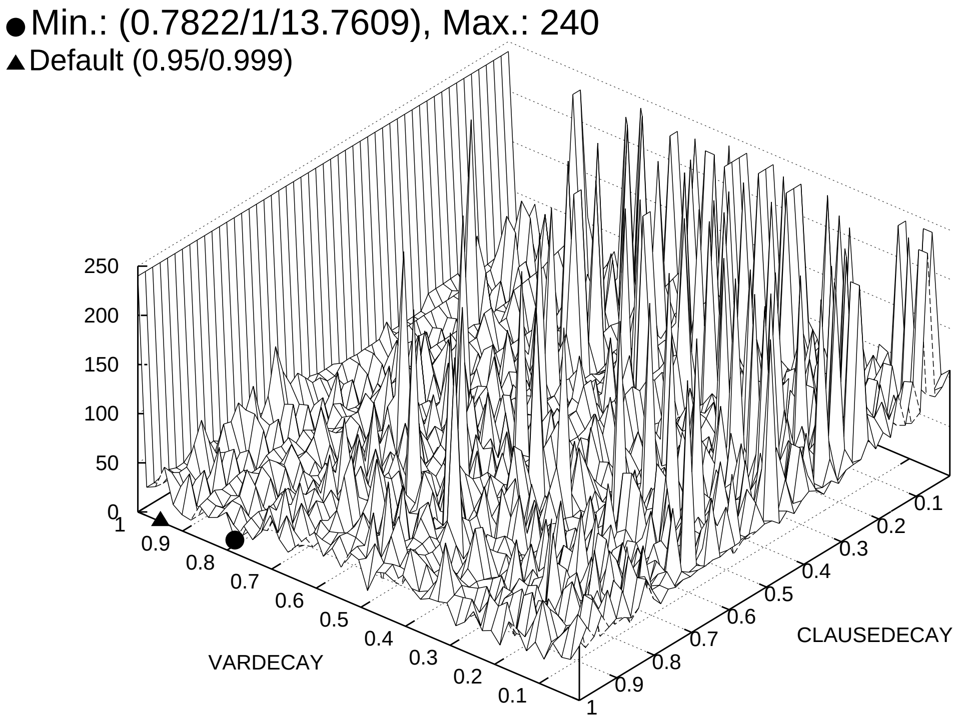


Figure 6.20: Fitness landscapes of 9 shuffled versions of problem `manol-pipe-c6n` for VARDECAY / CLAUSEDECAY (16 × 16 grid, timeout 120s, **`Pc1`**)



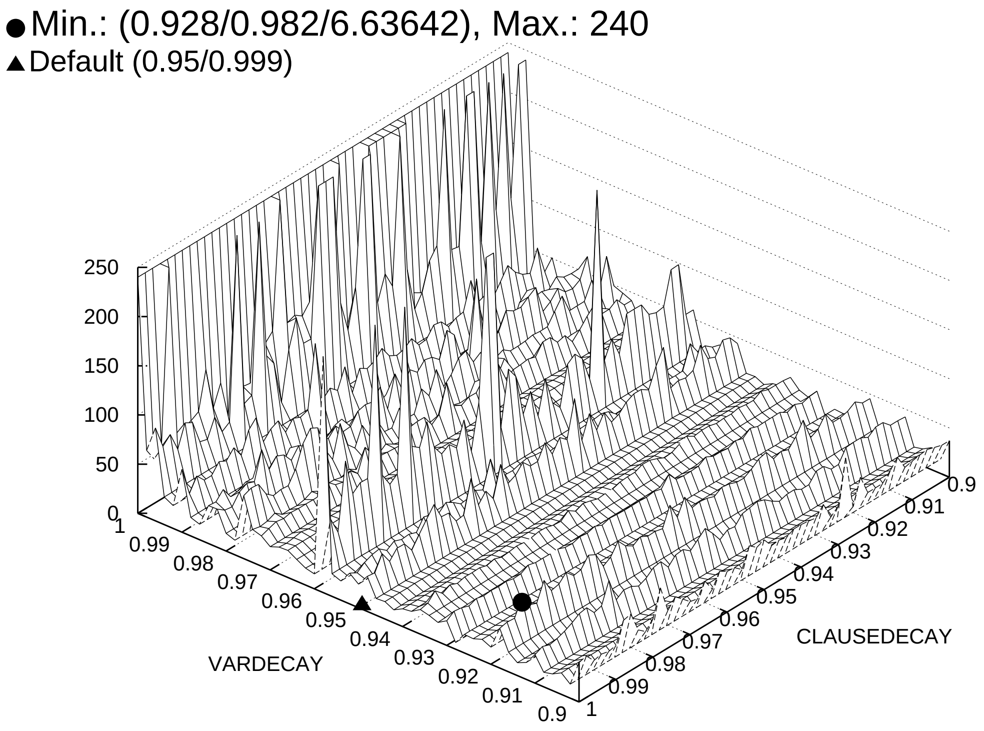


Figure 6.21: Fitness landscapes of problem `manol-pipe-c7_i` for VARDECAY / CLAUSEDECAY using the full range (top, 51×51 grid, timeout 120s, **`Pc0`**) and a close-up (bottom, 51×51 grid, timeout 120s, **`Pc0`**)

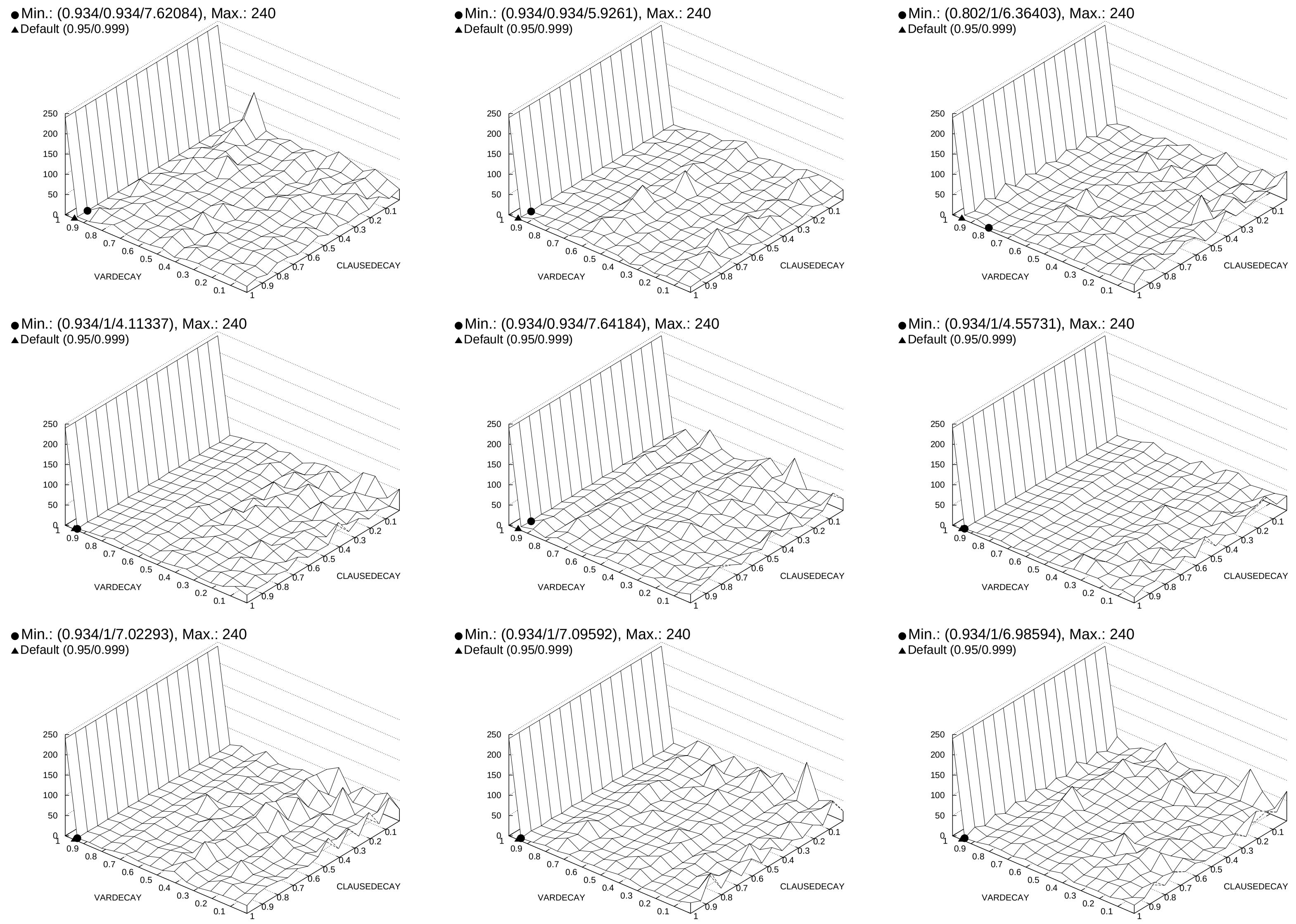
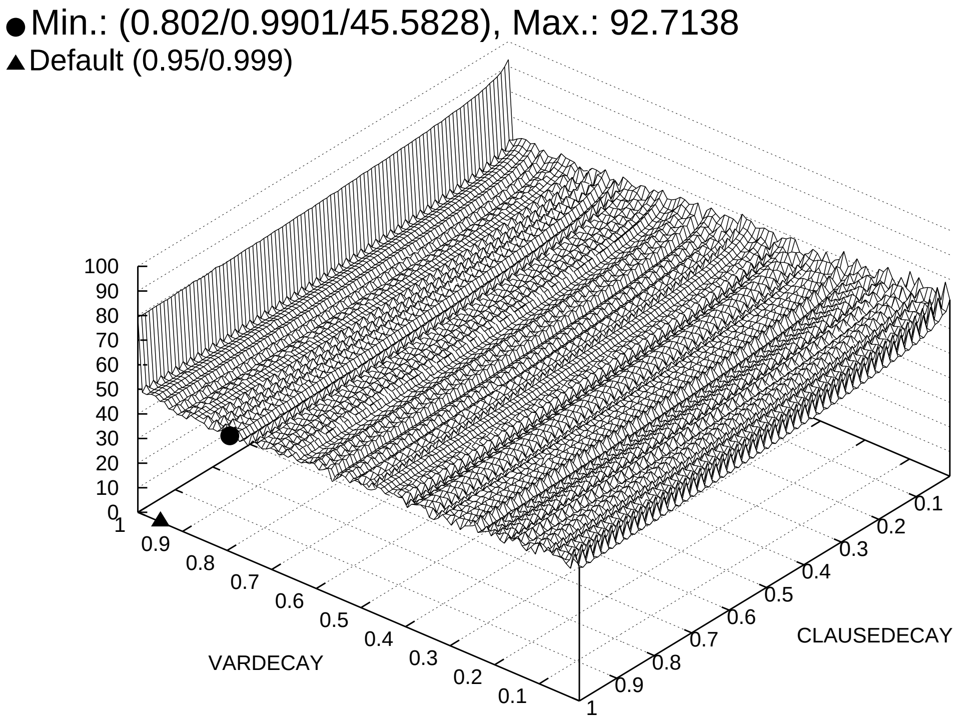


Figure 6.22: Fitness landscapes of 9 shuffled versions of problem `manol-pipe-c7_i` for VARDECAY / CLAUSEDECAY using the full range (16×16 grid, timeout 120s, **`Pc2`**)



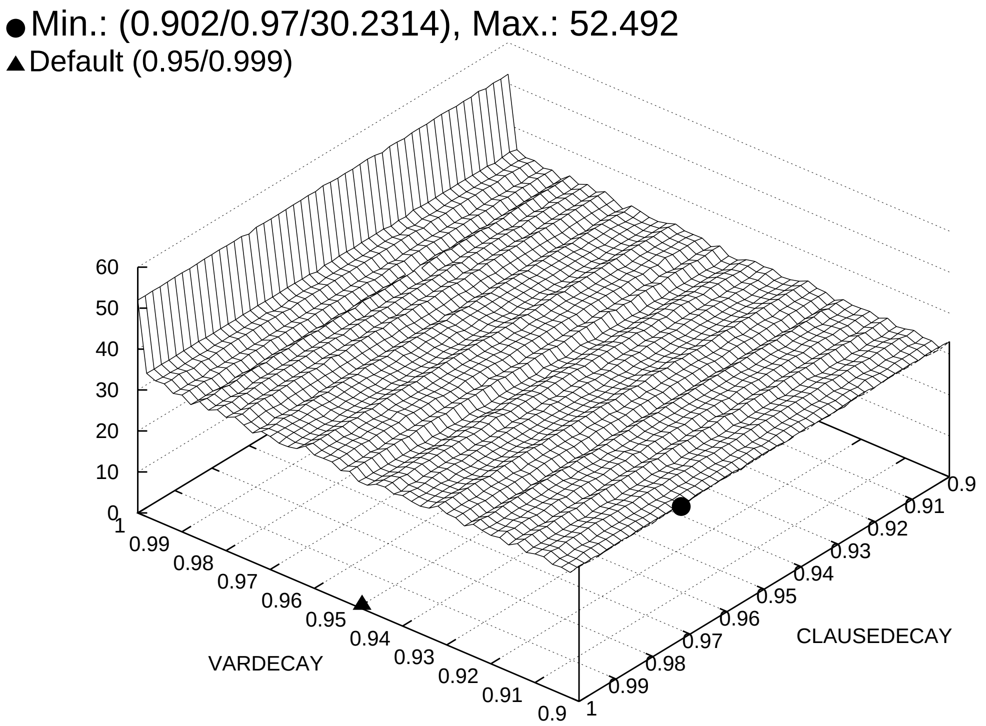


Figure 6.23: Fitness landscapes of problem `goldb-heqc-desmul` for VARDECAY / CLAUSEDECAY using the full range (top, 101×101 grid, timeout 300s, **`Pc2`**) and a close-up (bottom, 51×51 grid, timeout 300s, **`Pc4`**)

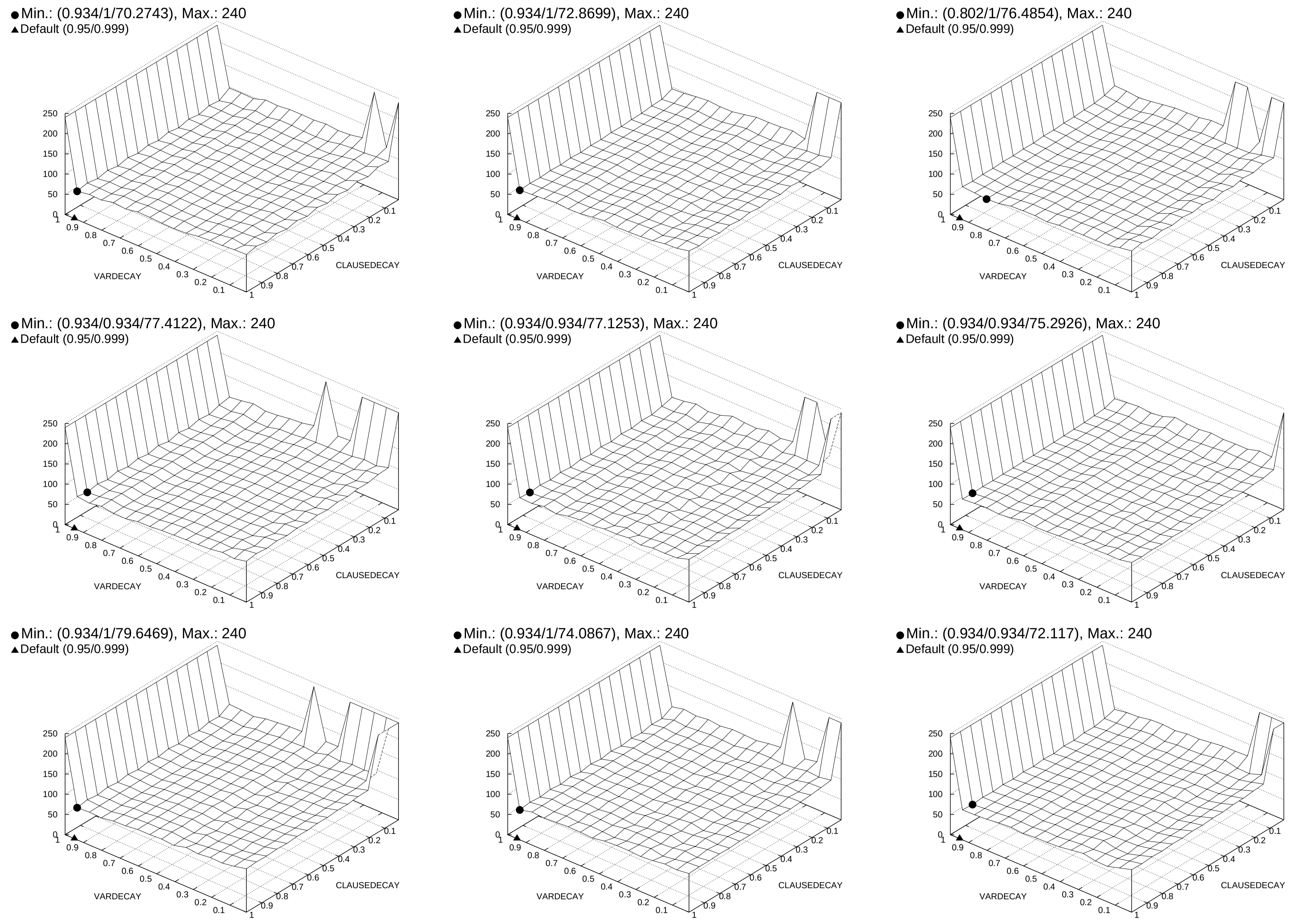
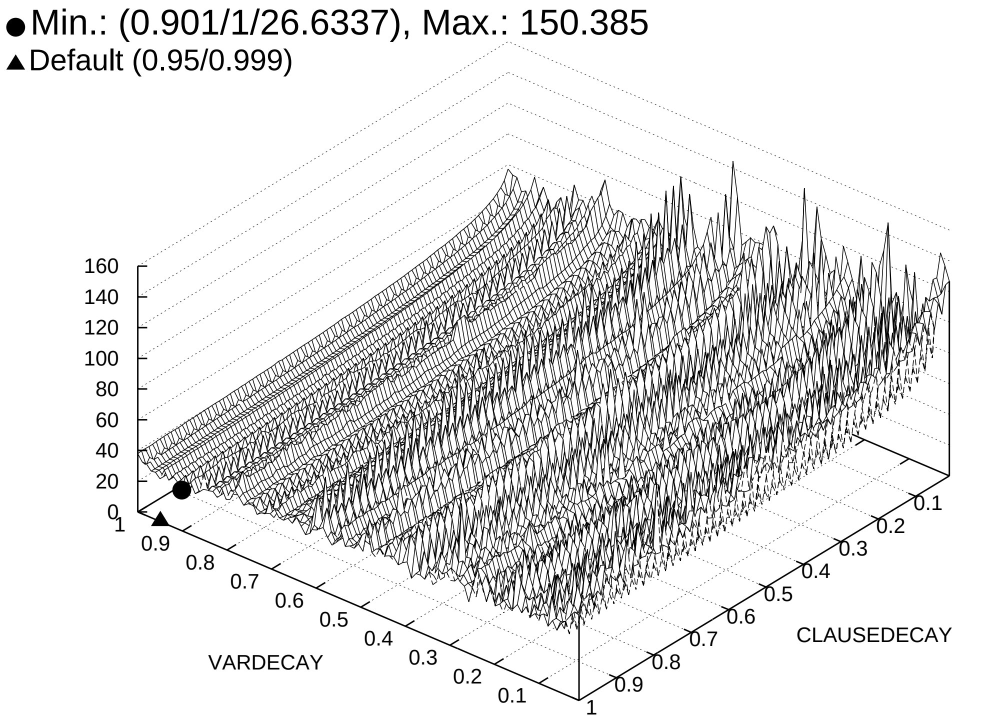


Figure 6.24: Fitness landscapes of 9 shuffled versions of problem `goldb-heqc-desmul` for VARDECAY / CLAUSEDECAY using the full range (16×16 grid, timeout 120s, **`Pc1`**)



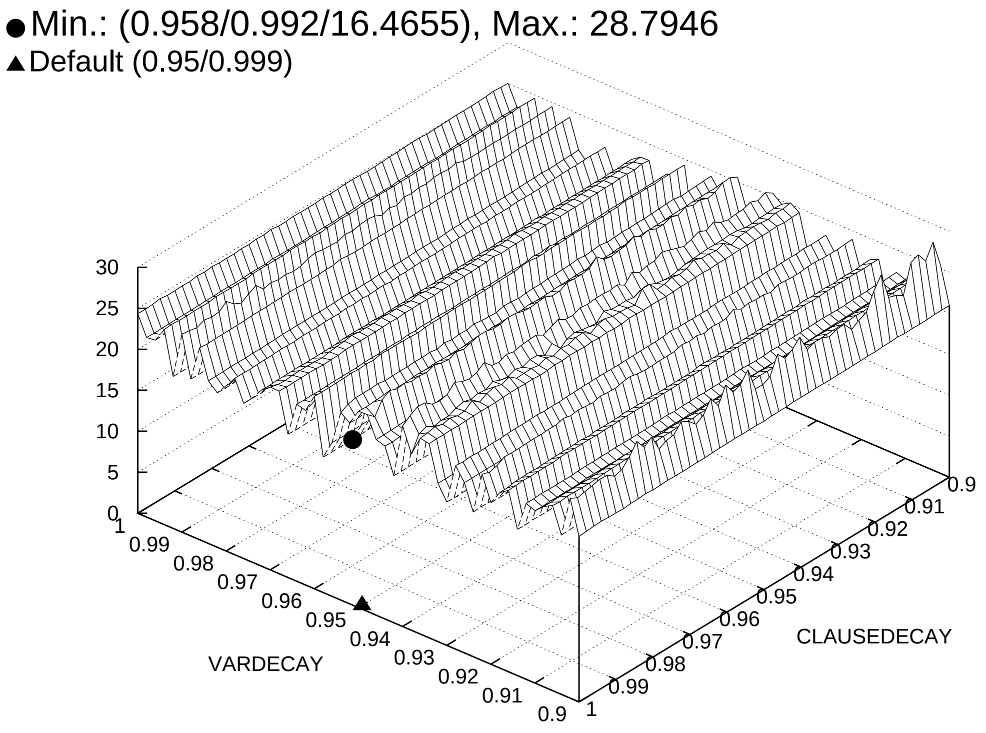


Figure 6.25: Fitness landscapes of problem `velev-eng-uns-1.0-04a` for VARDECAY / CLAUSEDECAY using the full range (top, 101×101 grid, timeout 300s, **`Pc2`**) and a close-up (bottom, 51×51 grid, timeout 300s, **`Pc4`**)

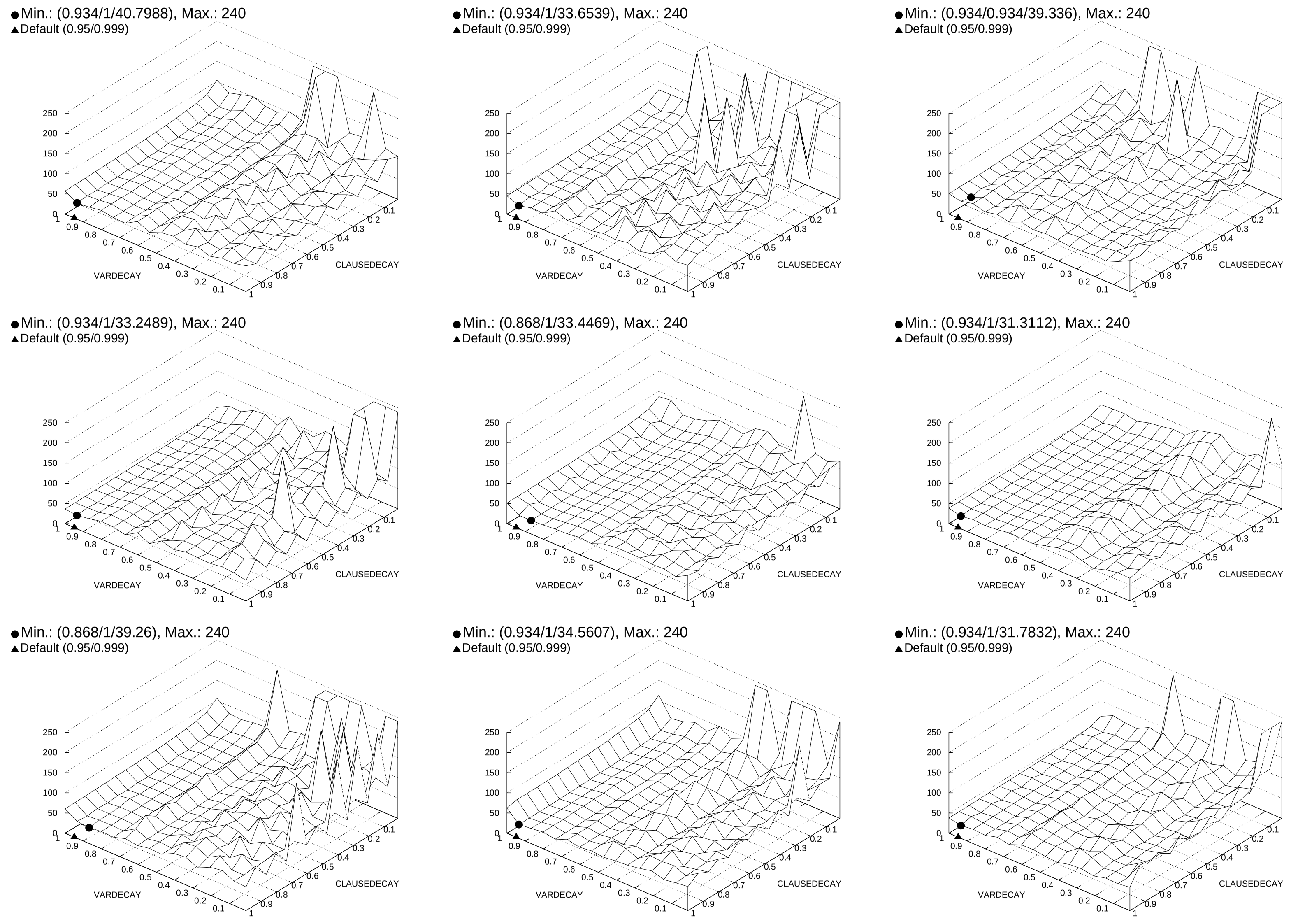
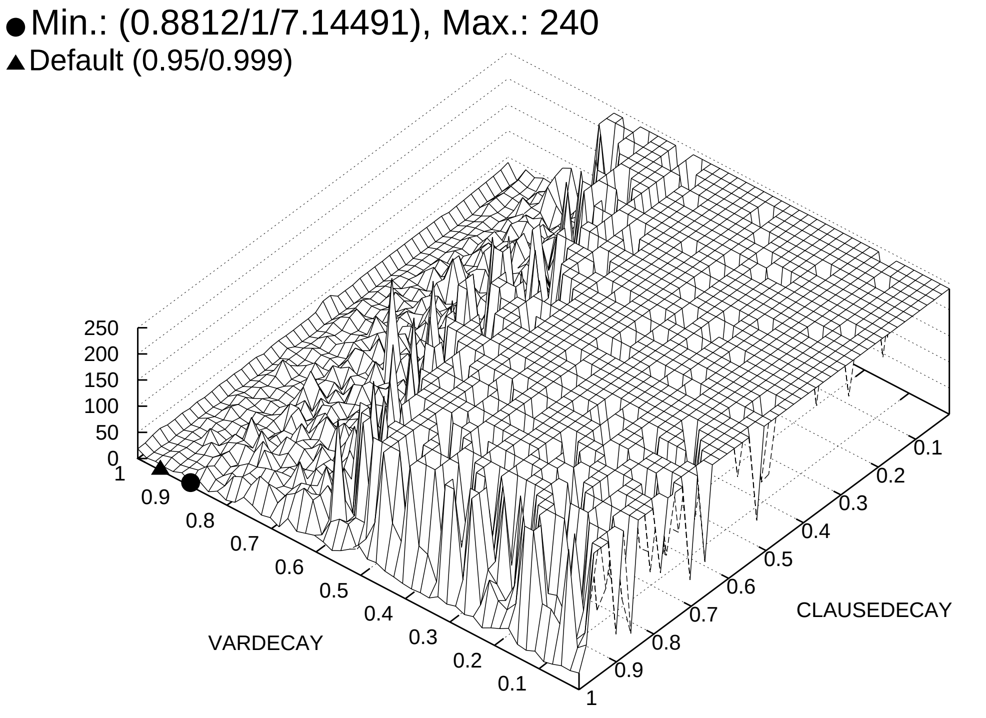


Figure 6.26: Fitness landscapes of 9 shuffled versions of problem `velev-eng-uns-1.0-04a` for VARDECAY / CLAUSEDECAY using the full range (16×16 grid, timeout 120s, **`Pc1`**)



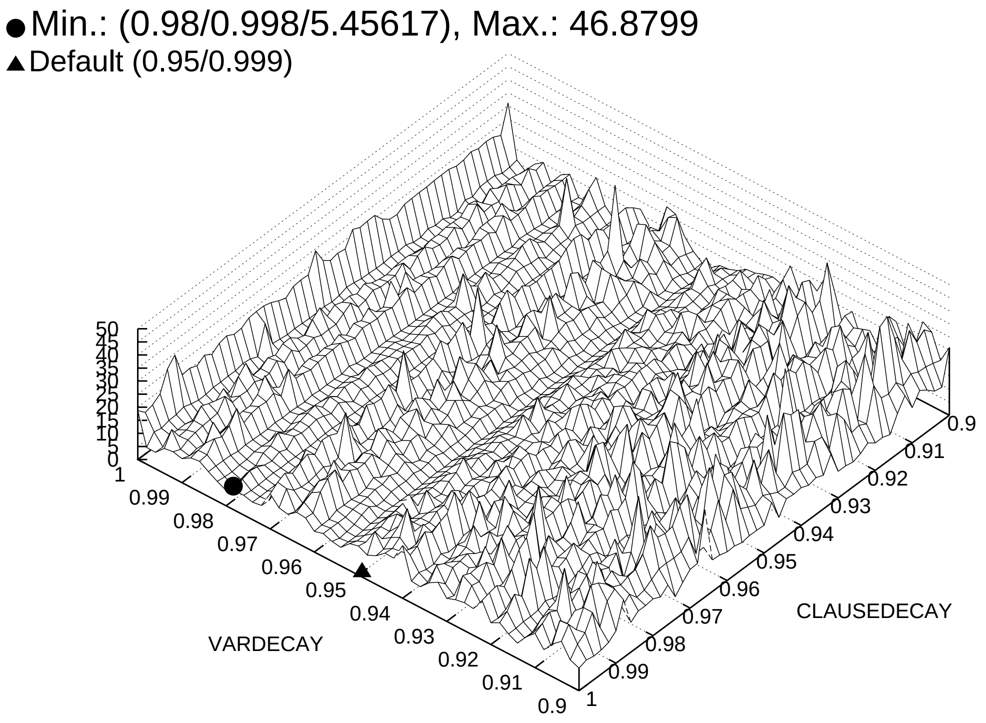


Figure 6.27: Fitness landscapes of problem `velev-npe-1.0-02` for VARDECAY / CLAUSEDECAY using the full range (top, 51×51 grid, timeout 120s, **Pc3**) and a close-up (bottom, 51×51 grid, timeout 120s, **Pc3**)

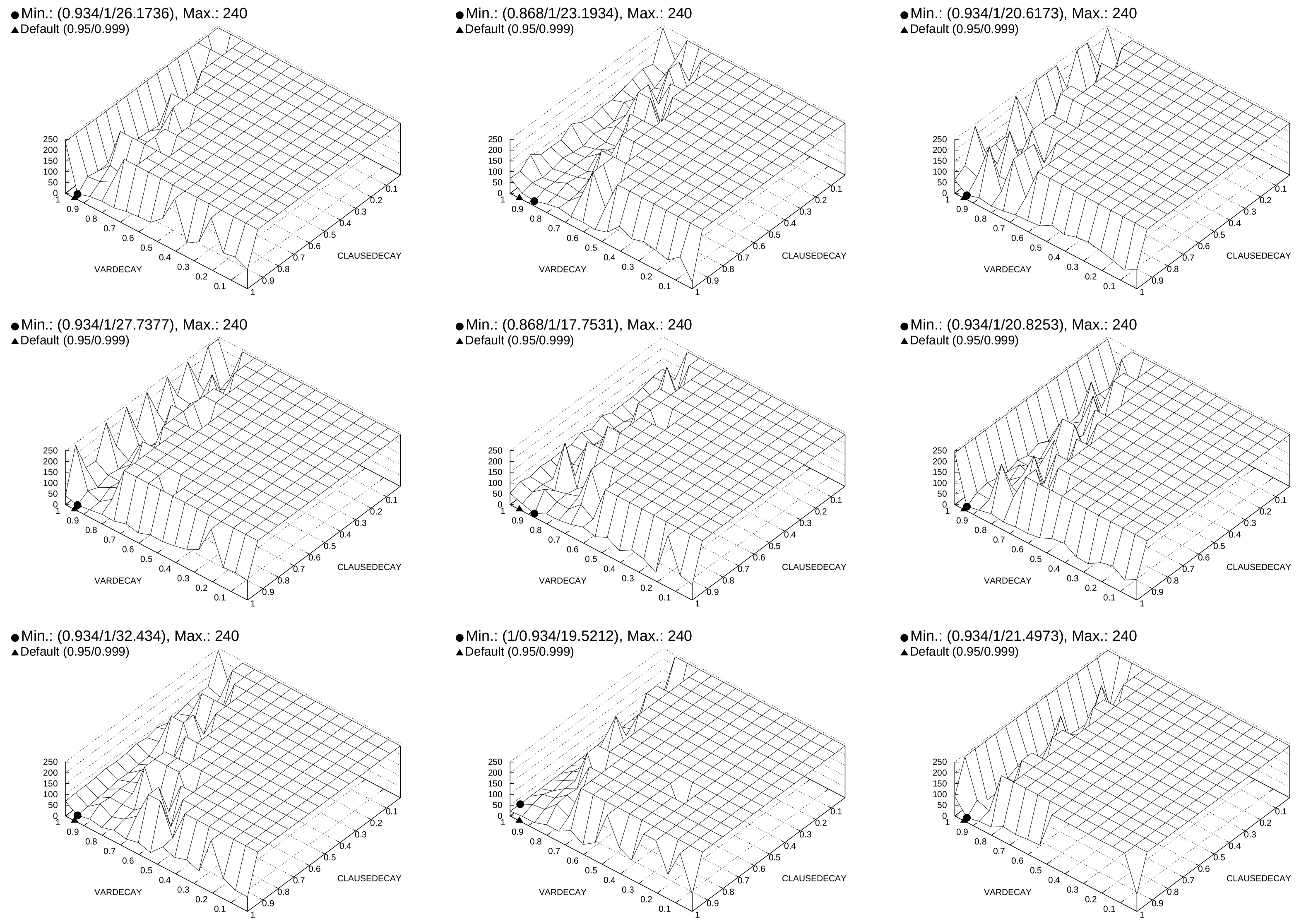


Figure 6.28: Fitness landscapes of 9 shuffled versions of problem `velev-npe-1.0-02` for VARDECAY / CLAUSEDECAY using the full range (16×16 grid, timeout 120s, **`Pc2`**)

6.4.4 NOFCONFLICTSINC / NOFCONFLICTSBASE

For the parameter pair $\mathrm{NOFCONFLICTSINC}$ and $\mathrm{NOFCONFLICTSBASE}$ in addition to the full range fitness landscape ($1 \leq \mathrm{NOFCONFLICTSINC} \leq 2$ and $10 \leq \mathrm{NOFCONFLICTSBASE} \leq 1000$) two close-ups of the area near the default values were made. Fig. 6.30 to Fig. 6.39 show the fitness landscapes. Each double page contains the landscapes for one of the test problems.

6.4.4.1 Observations

For this parameter pair it became apparent early on that the fitness landscapes are generally extremely noisy and show little structure, so that generating landscapes for shuffled problems (which would have to have much less resolution as well) did not seem promising. Instead, in addition to a “low resolution” close-up of the area around the default values ($1.4 \leq \mathrm{NOFCONFLICTSINC} \leq 1.6$ and $10 \leq \mathrm{NOFCONFLICTSBASE} \leq 200$) another, even narrower close-up ($1.49 \leq \mathrm{NOFCONFLICTSINC} \leq 1.51$ and $90 \leq \mathrm{NOFCONFLICTSBASE} \leq 110$) with a high resolution grid (101×101 points instead of 51×51) was computed.

1. None of the full range fitness landscapes show any discernible structure in the vast majority of the examined area; the run times seem to be very noisy. The landscapes contain a large number of seemingly randomly distributed, very narrow peaks and valleys without any apparent order. The landscapes remind of the surface of sandpaper. For problem `goldb-heqc-desmul` the landscape barely even shows these surface features: it is very flat all over, though still noisy, but to a lesser degree than the other landscapes.

2. The landscapes of problems `manol-pipe-c6n`, `goldb-heqc-desmul` and `velev-eng-uns-1.0-04a` have a visible increase of the solving time in the corner $\mathrm{NOFCONFLICTSINC} = 1$ / $\mathrm{NOFCONFLICTSBASE} = 10$ (the lower limits for both parameters).

3. The minimum points of the full range landscapes are found more toward the upper end of the parameter ranges ($1.46 - 2$ for $\mathrm{NOFCONFLICTSINC}$ and $406 - 980$ for $\mathrm{NOFCONFLICTSBASE}$). The minima in the close-up landscapes seem to have no system to them at all.

4. The first close-up landscapes are nearly indistinguishable from the full range landscapes: they seem to show a comparable amount of randomness as the full range ones, despite showing an area only about 1/25th in size but with the same grid resolution. No order is obvious in the landscapes.

5. The extreme close-up landscapes still show a lot of randomness despite the very small range for the parameters ($\mathrm{NOFCONFLICTSINC}$ only

varies in a range of 0.02 and NOFCONFLICTSBASE in 20, which is 2% of the full range for both axes). For clarity, Fig. 6.29 shows the graphs of NOFCONFLICTSINC and NOFCONFLICTSBASE as they pass through the default point $(1.5, 100)$ in the extreme close-up landscape for problem `manol-pipe-c6n` (Fig. 6.31). It can be seen that varying NOFCONFLICTSINC even by a tiny amount (the distance between points on this axis is only 0.0002) can drastically change the run time of the solver. The graph for NOFCONFLICTSBASE has "steps" about 5 grid points wide where the run time does not change. Small changes to this parameter also can change the run time by a large amount.

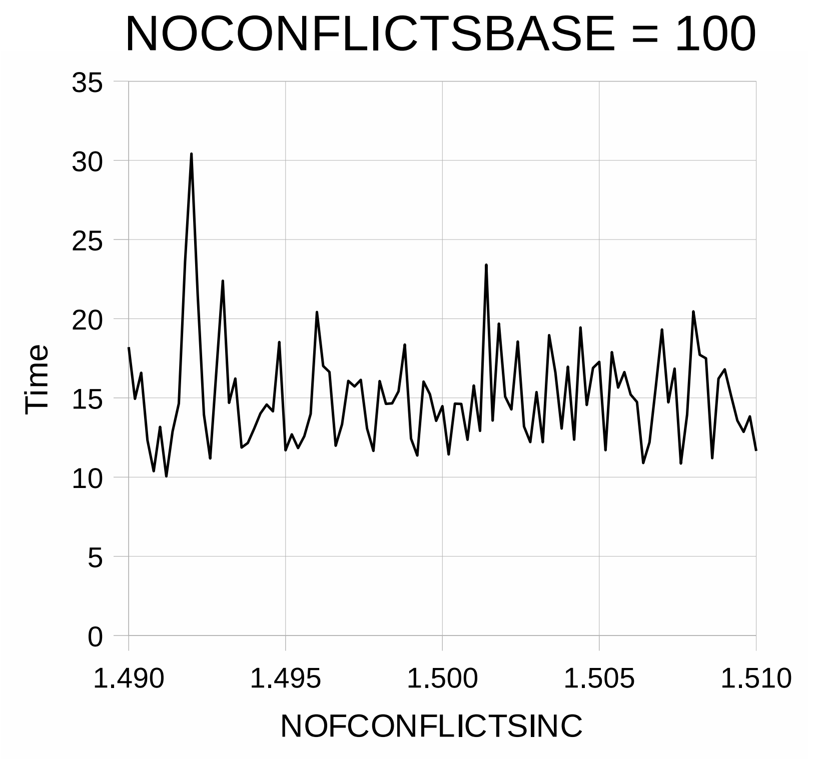
6.4.4.2 Interpretation

The parameters NOFCONFLICTSINC and NOFCONFLICTSBASE control the solver's restart strategy. Only integer values of NOFCONFLICTSBASE are sensible (since it defines the number of conflicts after which a restart is performed), and any positions after the decimal point for this parameter will be cut off when given to the solver. This leads to the many 5 grid points wide edges in the extreme close-up landscapes along which the run time is always the same (the decimal places of the parameter are cut off there), because the distance between grid points along the NOFCONFLICTSBASE axis is only 0.2.

The increase of run times in the NOFCONFLICTSINC $= 1$ / NOFCONFLICTSBASE $= 10$ corner of problems `manol-pipe-c6n`, `goldb-heqc-desmul` and `velev-eng-uns-1.0-04a` is most likely due to these values signifying very frequent restarts after relatively few conflicts, and a slowly (in the extreme case linearly) rising number of conflicts. The frequent restarts cause a lot of overhead, which increases the run time. Why this effect is visible only in three of the five test problems (and undetectable in the other two) is unknown.

Very small changes in either of the two parameters lead to wildly changing run times, as can be seen in the extreme close-up landscapes. In the full range and wider close-up landscapes this means that every grid point has a totally random run time, so that these landscapes have a "sandpaper" appearance. Why problem `goldb-heqc-desmul` is apparently less susceptible to changes of the parameters is unknown. In light of this randomness it seems doubtful that any weight can be given to the locations of the minimum points detected in the landscapes.

Restarts (see Sec. 3.6) are considered a vital part of SAT solving strategy to stabilize solving times and to keep the solver from being bogged down in unpromising regions of the search space. In this experiment restarts are never "turned off" completely, only the parameters of the geometric series that sets the limits after which a restart is performed are modified. At the "frequent restart" extreme,



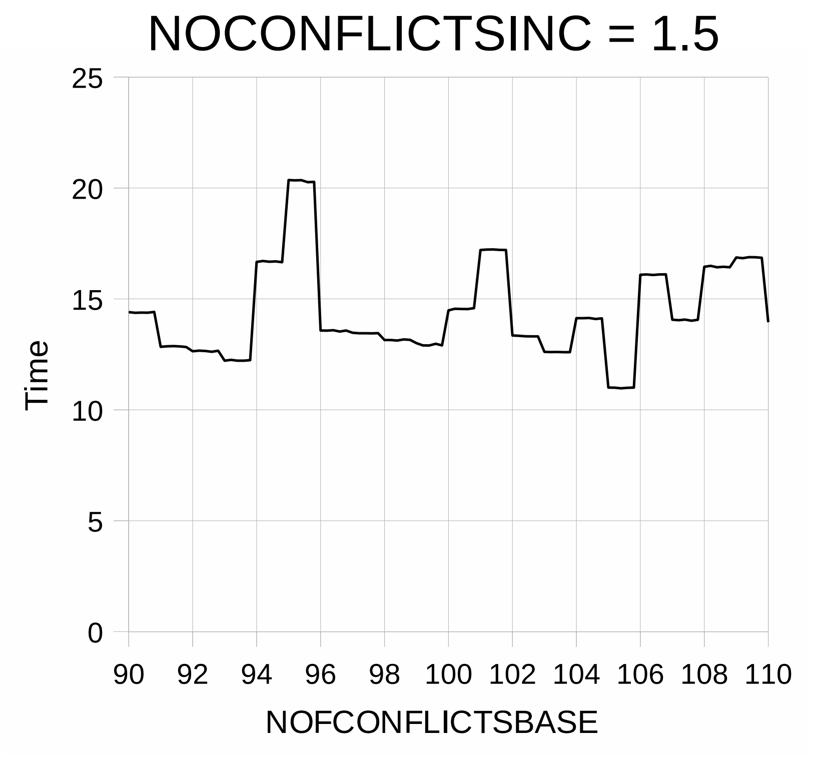
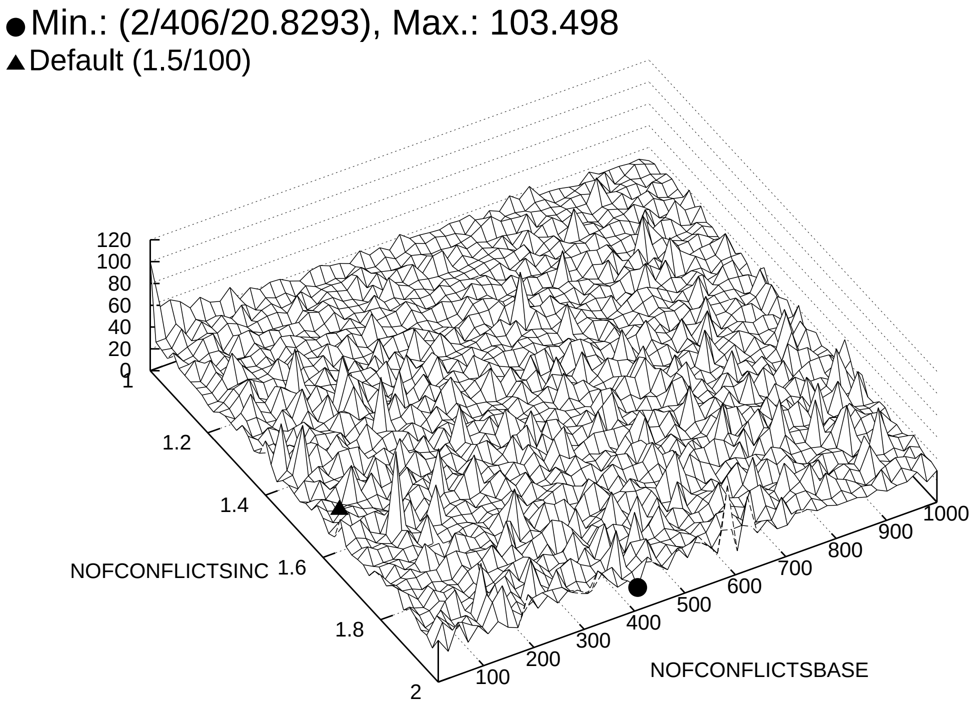


Figure 6.29: Separate graphs for run time versus NOFCONFLICTSINC and NOFCONFLICTSBASE passing through the default value point taken from the extreme closeup of the fitness landscape of problem `manol-pipe-c6n` (101 × 101 grid, timeout 120s, **`Pc5`**)

restarts are performed after every 10 conflicts ($\mathrm{NOFCONFLICTSINC} = 1$ and $\mathrm{NOFCONFLICTSBASE} = 10$); at the other extreme ($\mathrm{NOFCONFLICTSINC} = 2$ and $\mathrm{NOFCONFLICTSBASE} = 1000$) they are performed after 1000 conflicts, then 2000 more, then 4000, 8000, 16000, ... etc. The only conclusion that could be drawn from the landscapes seems to be that restarts should not be extremely frequent, as evidenced by the "raised corners" in three of the five problems, though even then some problems are not affected at all. Apart from this any choice of parameter inside the used limits seems viable, or at least not obviously preferable to another. There is no indication that the default values for the parameters are particularly good except that they do not lead to very frequent restarts.



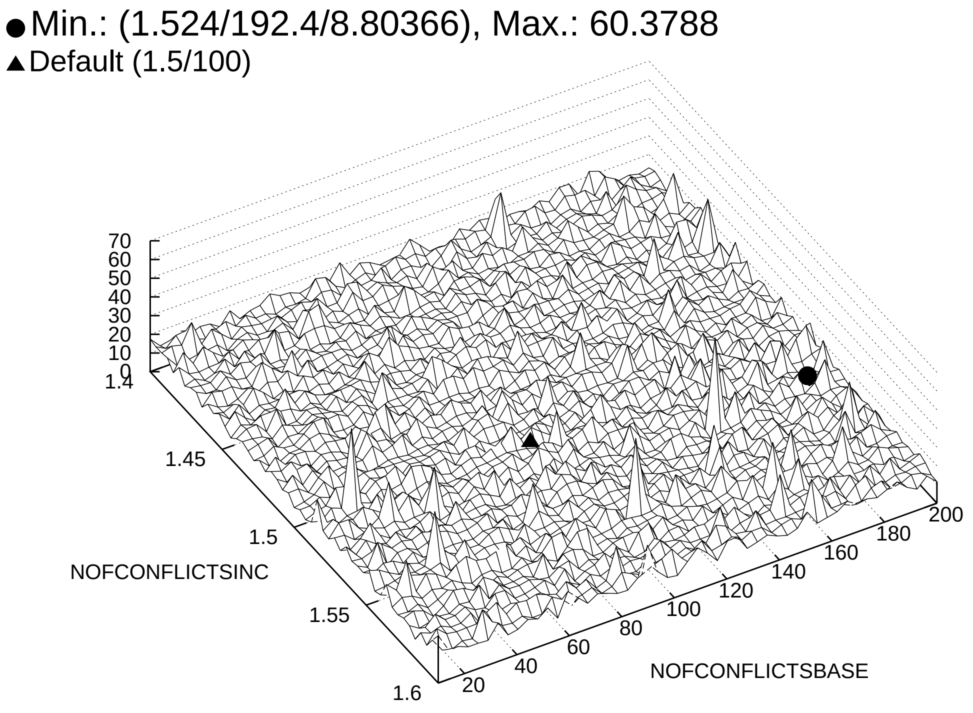


Figure 6.30: Fitness landscapes of problem `manol-pipe-c6n` for NOFCONFLICTSINC / NOFCONFLICTSBASE using the full range (top, 51×51 grid, timeout 300s, **`Pc2`**) and a close-up (bottom, 51×51 grid, timeout 300s, **`Pc5`**)

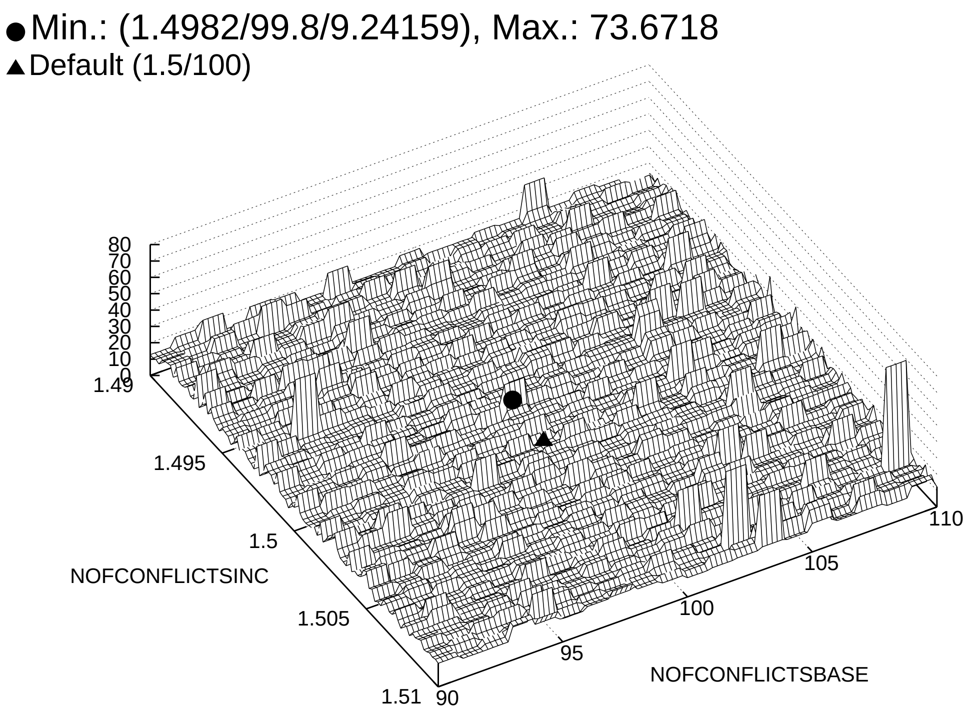
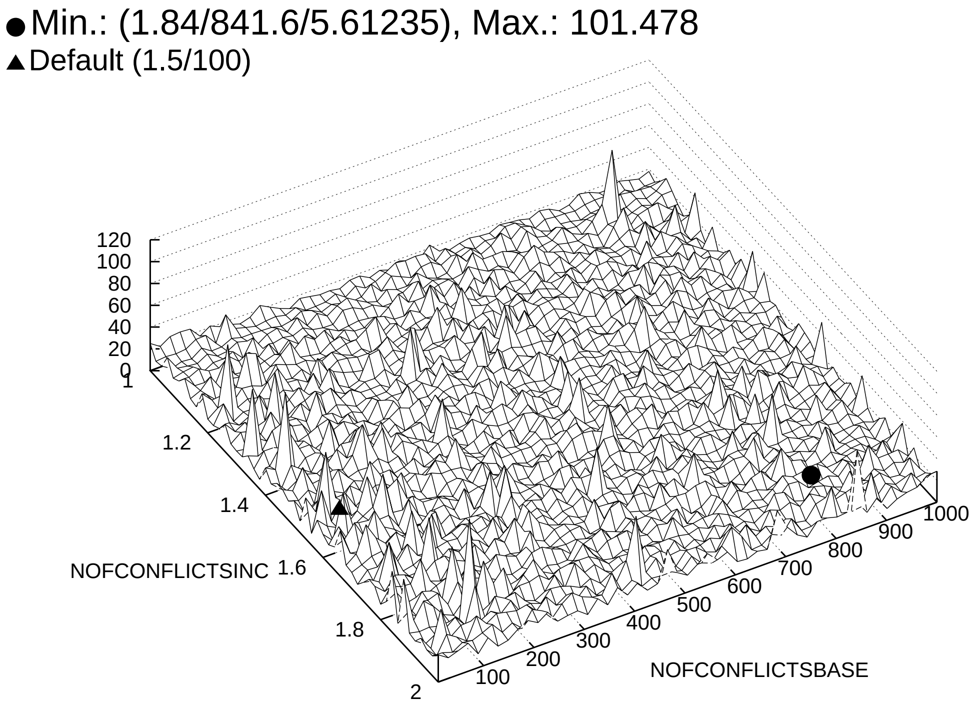


Figure 6.31: Extreme closeup of the fitness landscape of problem `manol-pipe-c6n` for NOFCONFLICTSINC / NOFCONFLICTSBASE (101×101 grid, timeout 120s, **`Pc5`**)



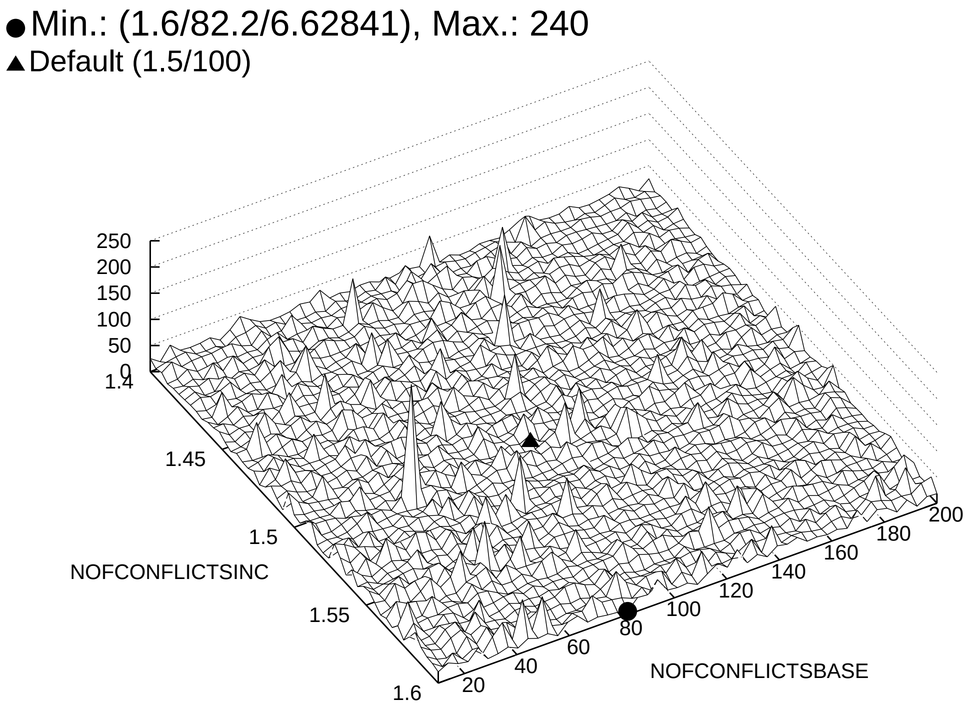


Figure 6.32: Fitness landscapes of problem `manol-pipe-c7_i` for NOFCONFLICTSINC / NOFCONFLICTSBASE using the full range (top, 51×51 grid, timeout 120s, **`Pc0`**) and a close-up (bottom, 51×51 grid, timeout 120s, **`Pc0`**)

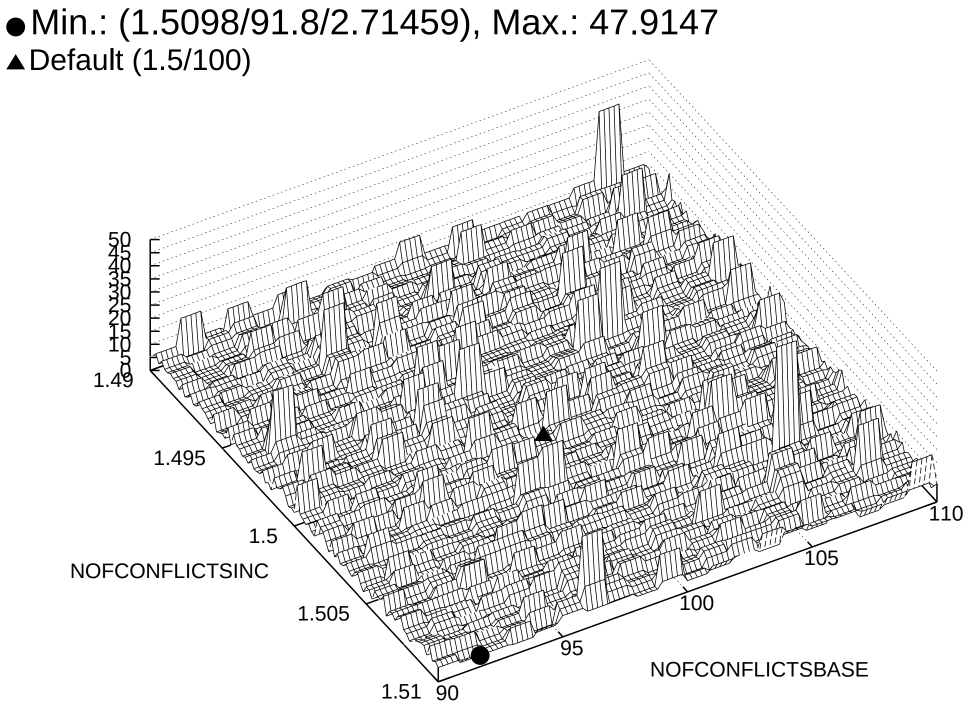
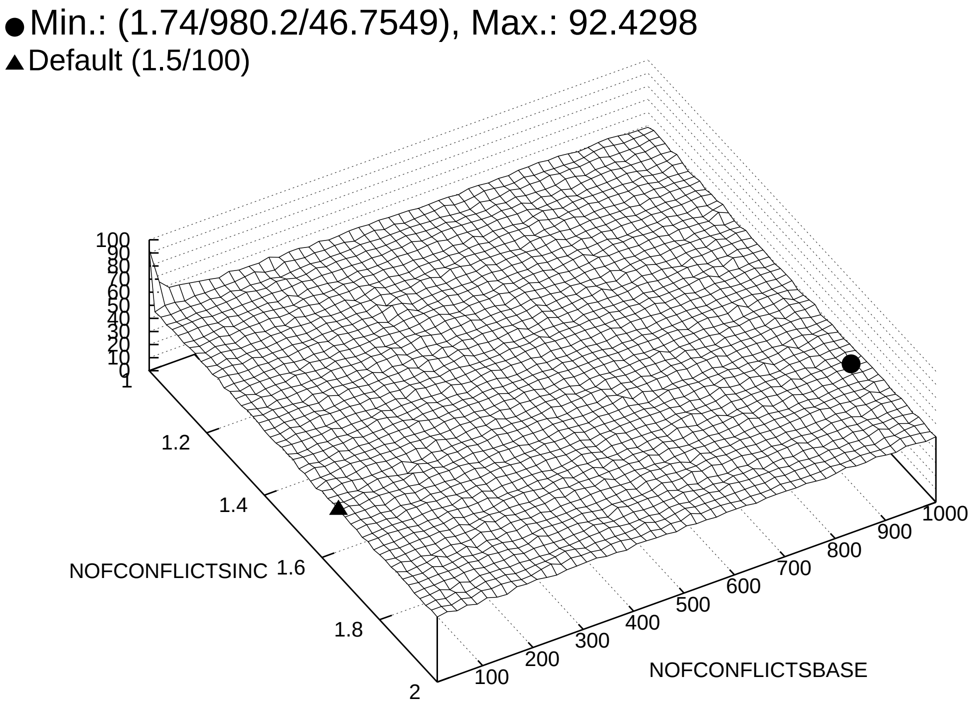


Figure 6.33: Extreme closeup of the fitness landscape of problem `manol-pipe-c7_i` for NOFCONFLICTSINC / NOFCONFLICTSBASE (101 × 101 grid, timeout 120s, **`Pc3`**)



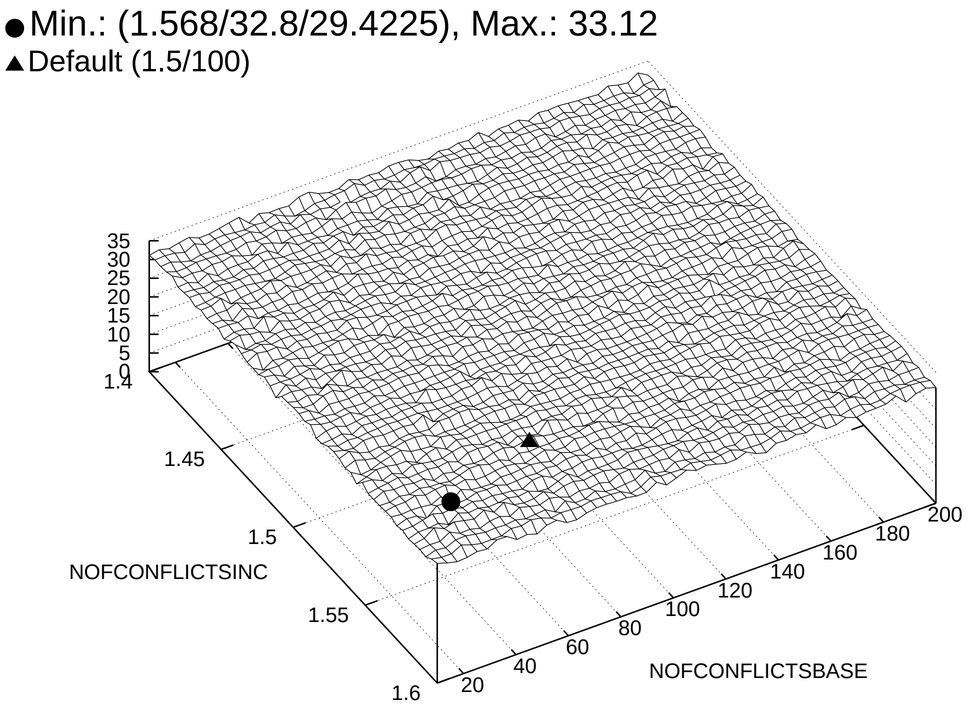


Figure 6.34: Fitness landscapes of problem `goldb-heqc-desmul` for NOFCONFLICTSINC / NOFCONFLICTSBASE using the full range (top, 51×51 grid, timeout 300s, **Pc2**) and a close-up (bottom, 51×51 grid, timeout 300s, **Pc4**)

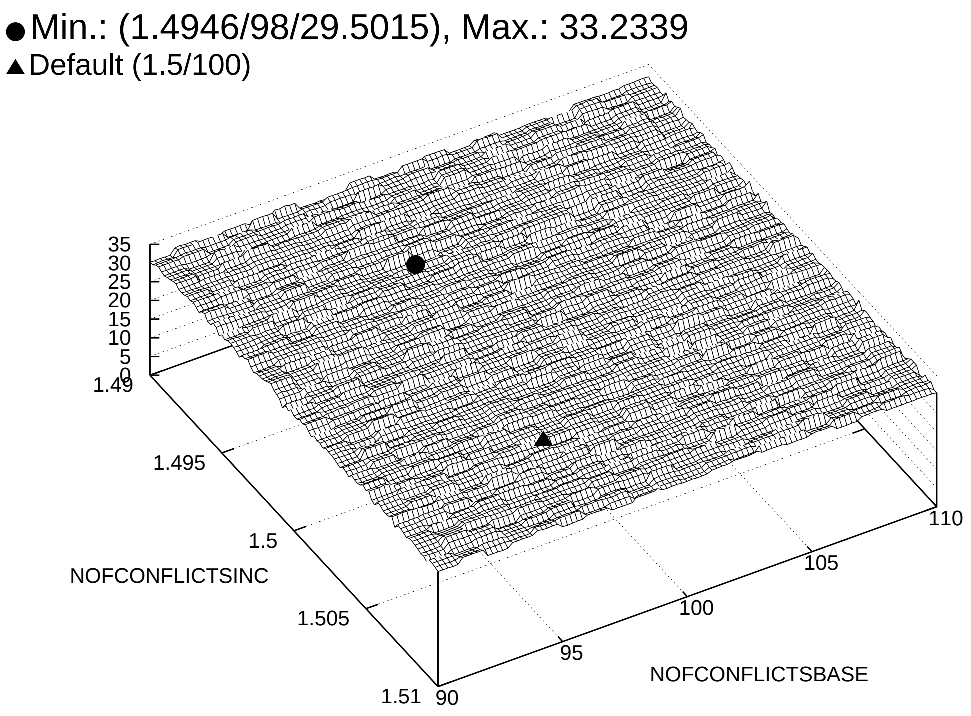
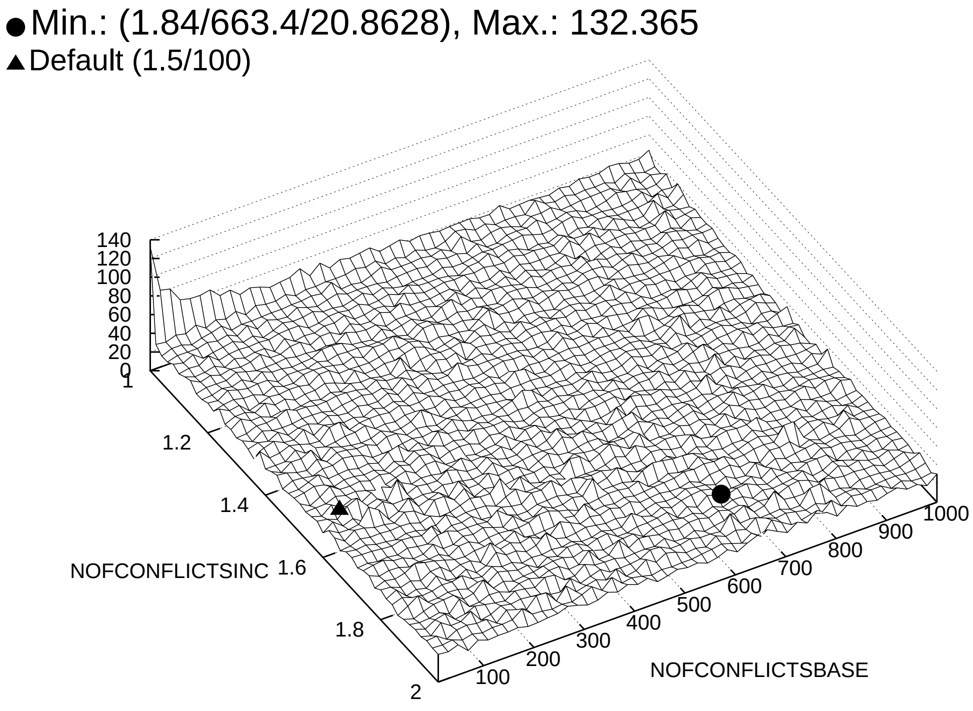


Figure 6.35: Extreme closeup of the fitness landscape of problem `goldb-heqc-desmul` for NOFCONFLICTSINC / NOFCONFLICTSBASE (101×101 grid, timeout 120s, **`Pc4`**)



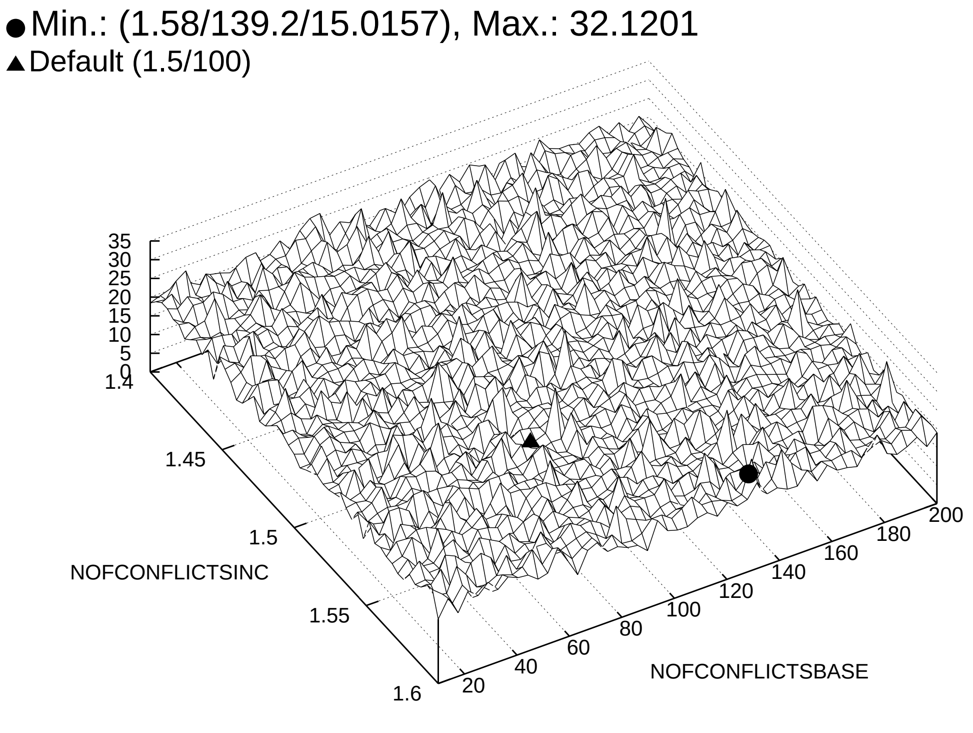


Figure 6.36: Fitness landscapes of problem `velev-eng-uns-1.0-04a` for NOFCONFLICTSINC / NOFCONFLICTSBASE using the full range (top, 51×51 grid, timeout 300s, **`Pc2`**) and a close-up (bottom, 51×51 grid, timeout 300s, **`Pc4`**)

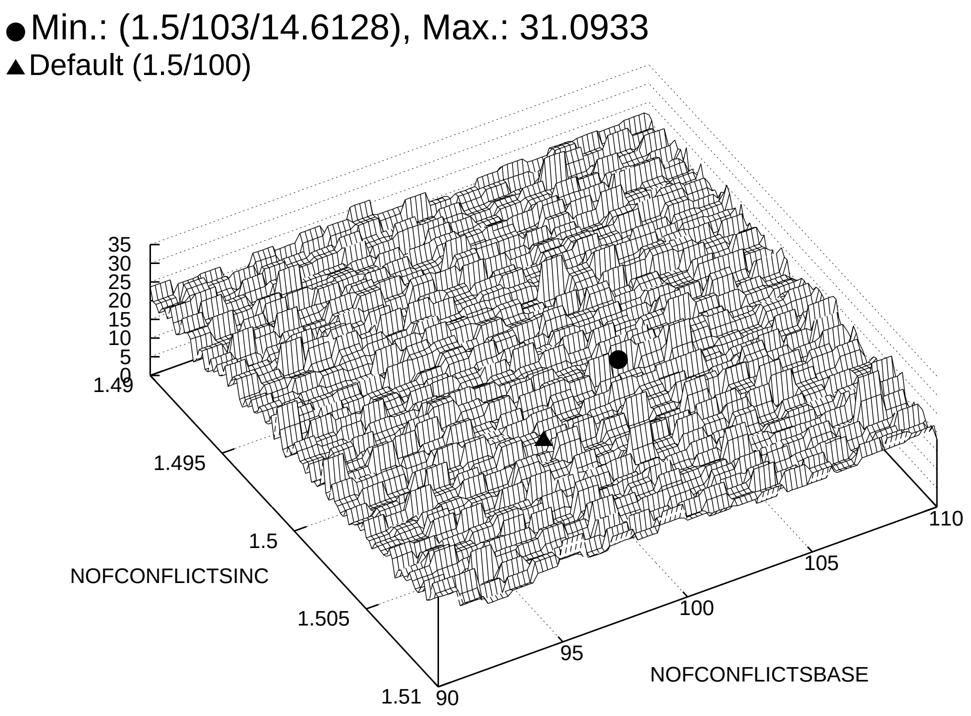
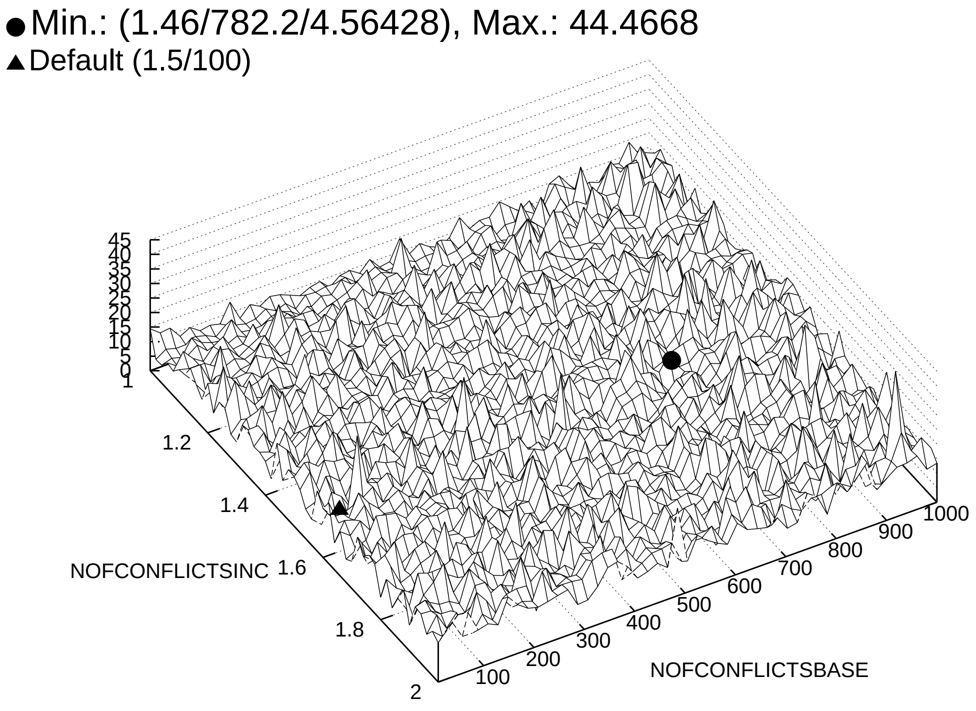


Figure 6.37: Extreme closeup of the fitness landscape of problem `velev-eng-uns-1.0-04a` for NOFCONFLICTSINC / NOFCONFLICTSBASE (101 $\times$ 101 grid, timeout 120s, **`Pc4`**)



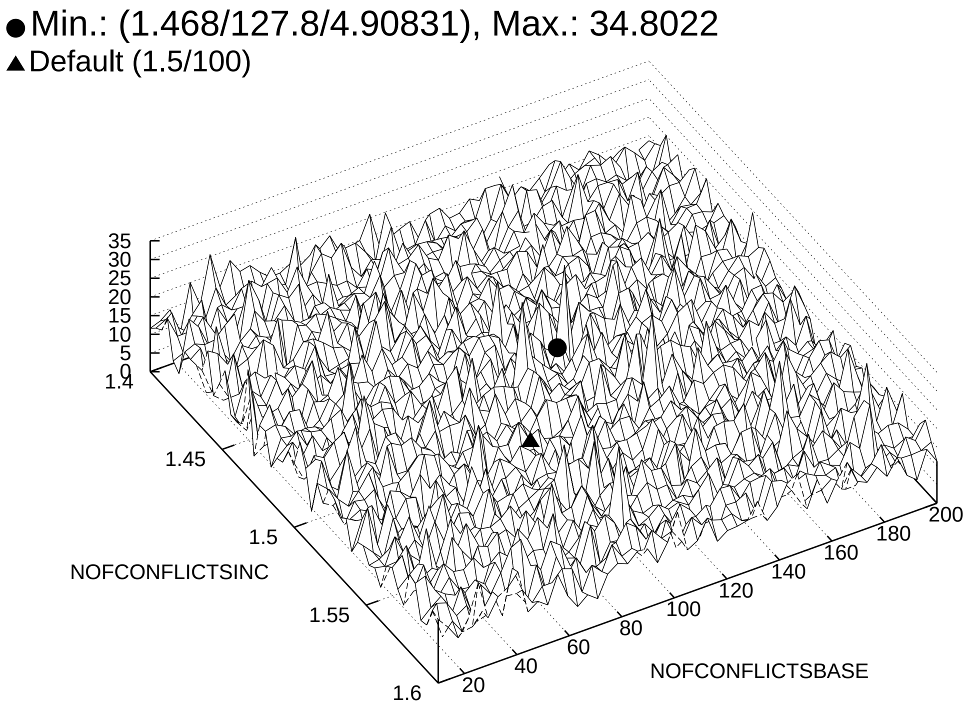


Figure 6.38: Fitness landscapes of problem `velev-npe-1.0-02` for NOFCONFLICTSINC / NOFCONFLICTSBASE using the full range (top, 51×51 grid, timeout 120s, **`Pc2`**) and a close-up (bottom, 51×51 grid, timeout 120s, **`Pc2`**)

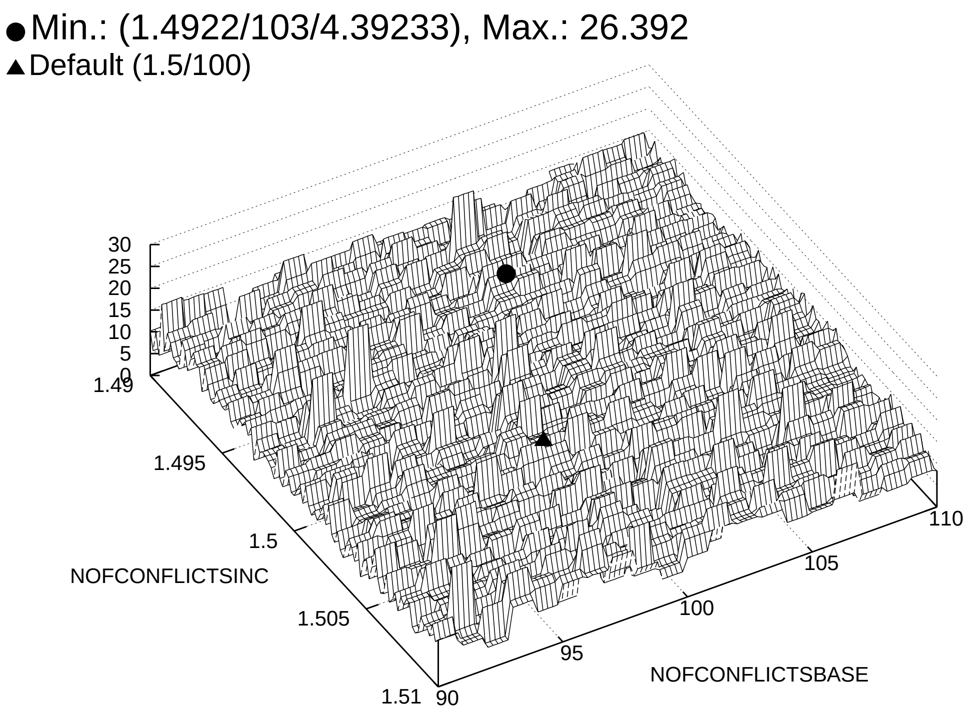


Figure 6.39: Extreme closeup of the fitness landscape of problem `velev-npe-1.0-02` for NOFCONFLICTSINC / NOFCONFLICTSBASE (101×101 grid, timeout 300s, **`Pc3`**)

6.4.5 NOFLEARNTSINC / NOFLEARNTSDIVISOR

For the parameter pair $\mathrm{NOFLEARNTSINC}$ and $\mathrm{NOFLEARNTSDIVISOR}$ in addition to the full range fitness landscape ($1 \leq \mathrm{NOFLEARNTSINC} \leq 2$ and $0.1 \leq \mathrm{NOFLEARNTSDIVISOR} \leq 10$) a close-up of the region $1 \leq \mathrm{NOFLEARNTSINC} \leq 1.2$ and $2 \leq \mathrm{NOFLEARNTSDIVISOR} \leq 4$ that contains the default values was generated. Additionally, full range fitness landscapes of 9 *shuffled* versions of each SAT problem were computed. For the shuffled problems in this experiment, the seeds were 100001 to 100009. Fig. 6.40 to Fig. 6.49 show the computed fitness landscapes. Each double page contains the landscapes for one of the test problems.

6.4.5.1 Observations

1. All full range landscapes are noticeably separated into two distinct "zones": the "mountainous" zone is a roughly triangular area of random peaks and valleys, and the "flat" zone is the rest of the landscape which is completely flat. For problem `goldb-heqc-desmul` the mountainous zone is much smaller than for all the other problems. The corners of the triangular mountainous zones are at the points $(1, 0.1)$, $(1, 10)$ and $(x, 10)$, where x is some value of $\mathrm{NOFLEARNTSINC}$ that is at the highest about 1.2. The peaks in this zone can reach over the level of the flat zone, and the valleys can reach levels under the flat zone.

2. The close-up landscapes show the transition between the mountainous and the flat zone. Even in magnification the flat zone appears completely flat.

3. The low-resolution landscapes of the shuffled problems indicate that these two consist of the two zones. The mountainous zone is located in a similar region as in the original problems.

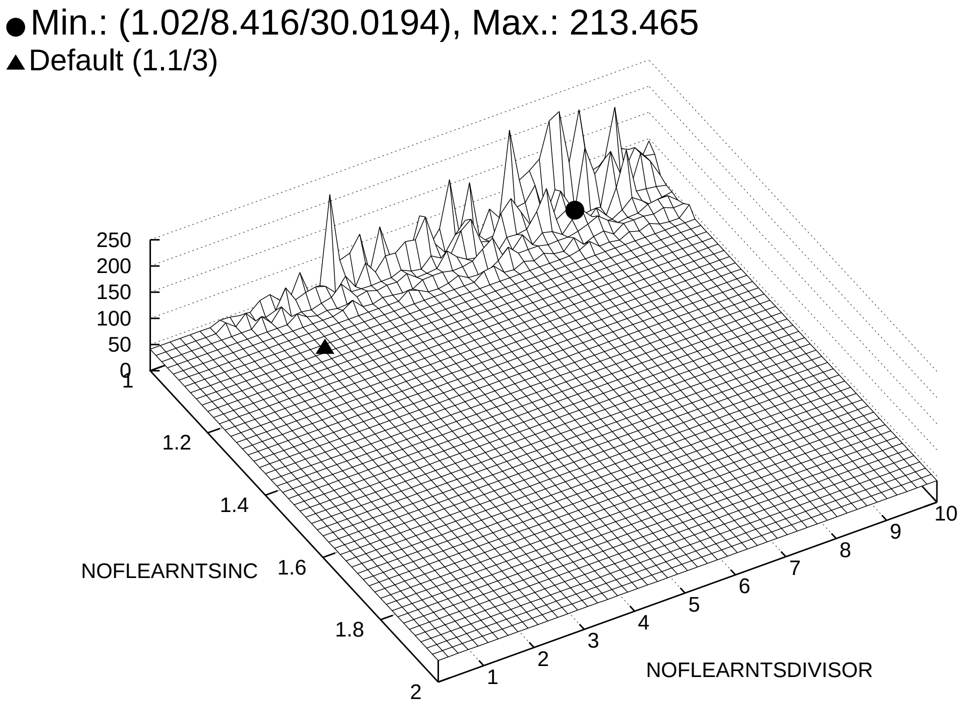
4. The minimum points in the full range landscapes are all located inside the mountainous zone, except for problem `goldb-heqc-desmul` where this zone is a narrow strip of peaks and the entire rest of the landscape is flat. In the landscapes of shuffled problems the minimum is sometimes in the mountainous and sometimes in the flat zone, but due to the low grid resolution used for these it is unknown if the "true" minimum points were simply missed.

6.4.5.2 Interpretation

The parameters $\mathrm{NOFLEARNTSINC}$ and $\mathrm{NOFLEARNTSDIVISOR}$ control the deletion of learned clauses. The initially allowed number of learned clauses is the total clauses divided by $\mathrm{NOFLEARNTSDIVISOR}$; this means the higher this value is the fewer learned clauses are allowed (at the lowest only 10% of the initial clauses since 10 is the highest allowed value). After every restart the allowed number of learned clauses is multiplied with $\mathrm{NOFLEARNTSINC}$.

The triangular mountainous zone marks the region where the number of learned clauses is kept fairly low, while beyond it in the flat zone a large number of learned clauses is allowed. The worst case of the lowest initial amount of learned clauses being allowed and the smallest increase is at the point $\mathrm{NOFLEARNTSINC} = 1$ and $\mathrm{NOFLEARNTSDIVISOR} = 10$, which is the corner furthest in the back in the diagrams. Learning a lot of learned clauses is generally considered problematic since it slows down the BCP engine considerably (this is the reason why learned clauses are deleted periodically in the first place). Deleting learned clauses can also have the undesirable effect of destroying vital information that may be needed to prove the satisfiability of the problem, therefore tuning this mechanism is a balance act between throwing away "useless" clauses and keeping as many "useful" ones as possible without slowing down BCP too much.

The landscapes indicate that after a certain point the allowed number of learned clauses becomes large enough that further increases have no effect; this is the solver operating in the flat zone. If the allowed number is kept low, the effect is that the run time can increase but also decrease. Presumably if it increases, the solver deletes useful clauses too often (the deletion cycle also costs time) and has to retrace its steps continually to regenerate the necessary clauses until it can solve the problem. If the run time decreases then the solver frequently deleted clauses, which keeps the clause database small and BCP fast but useful clauses are not deleted, which leads to an early success. Unfortunately no pattern can be seen in the landscapes for which parameter setting is beneficial, making the use of very fast clause deletion settings a gamble. The default parameters used in MINISAT are well within the flat zone.



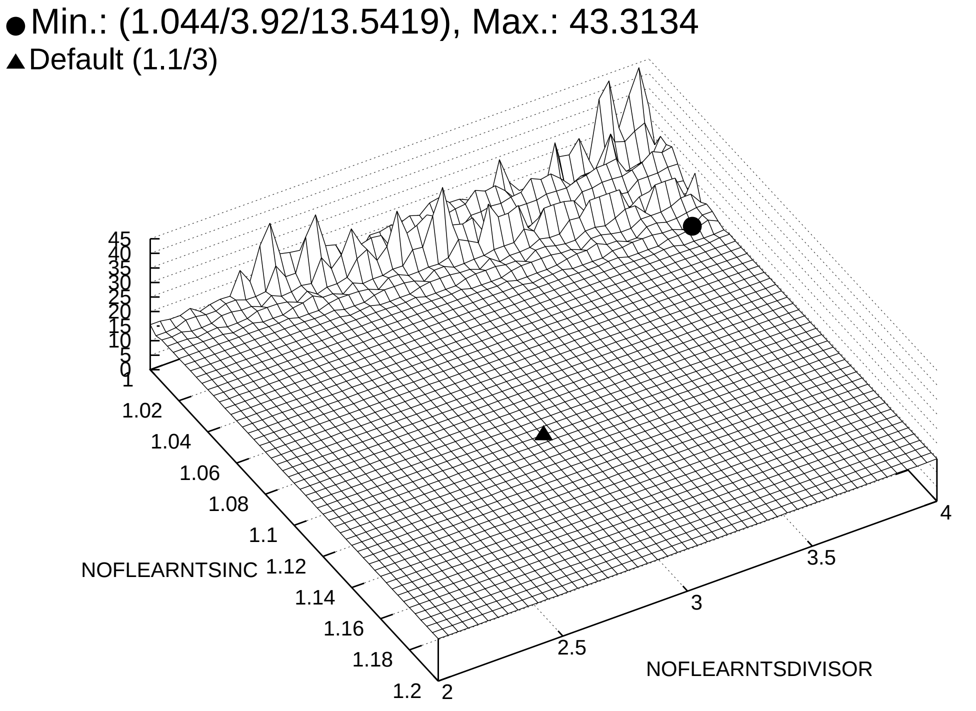


Figure 6.40: Fitness landscapes of problem `manol-pipe-c6n` for NOFLEARNTSINC / NOFLEARNTSDIVISOR using the full range (top, 51×51 grid, timeout 300s, **`Pc2`**) and a close-up (bottom, 51×51 grid, timeout 300s, **`Pc5`**)

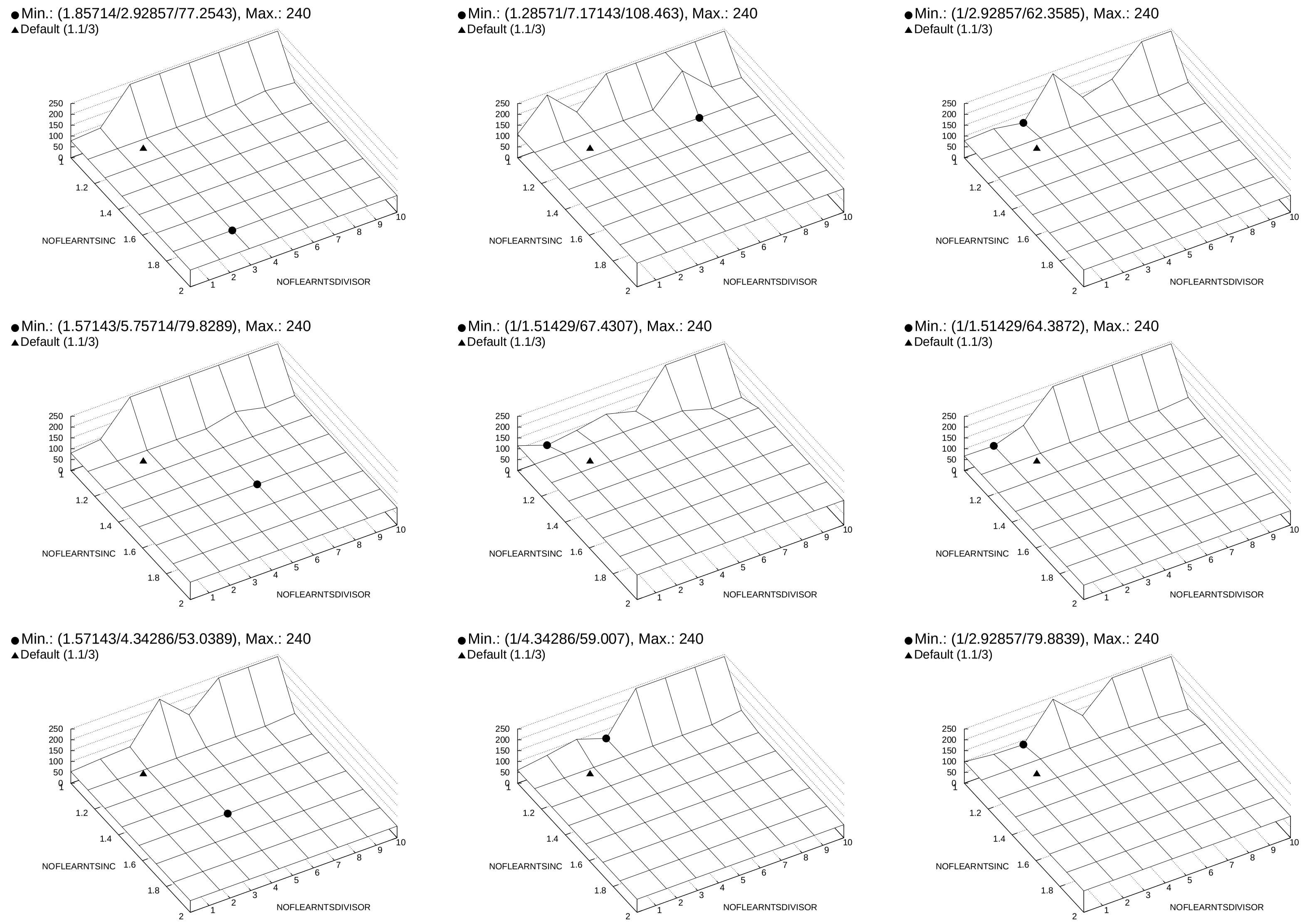
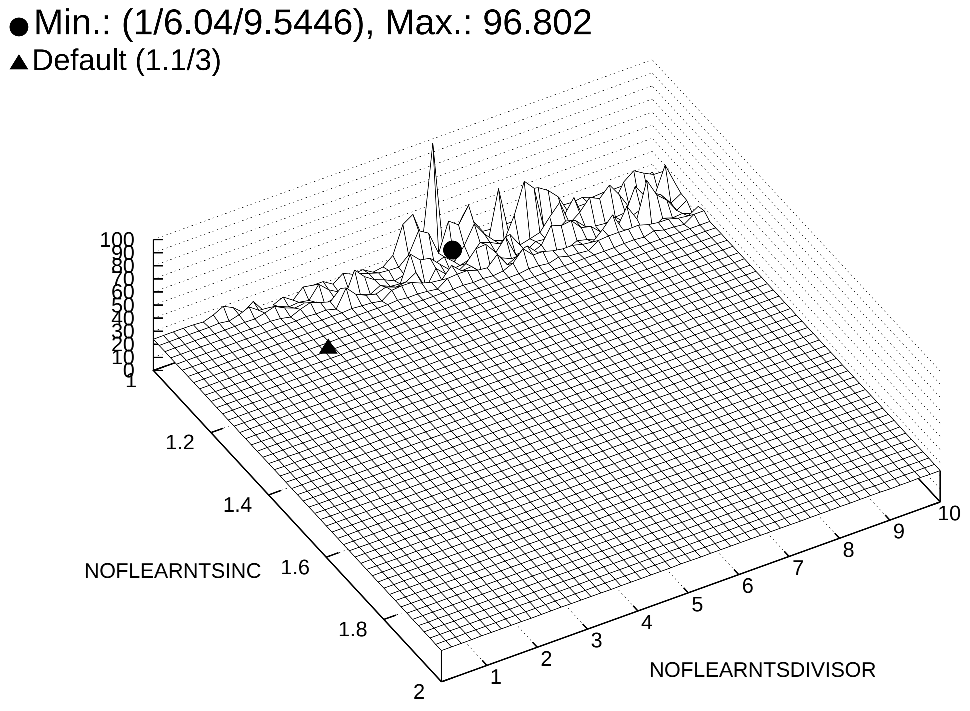


Figure 6.41: Fitness landscapes of 9 shuffled versions of problem `manol-pipe-c6n` for NOFLEARNTSINC / NOFLEARNTSDIVISOR (8×8 grid, timeout 120s, **`Pc1`**)



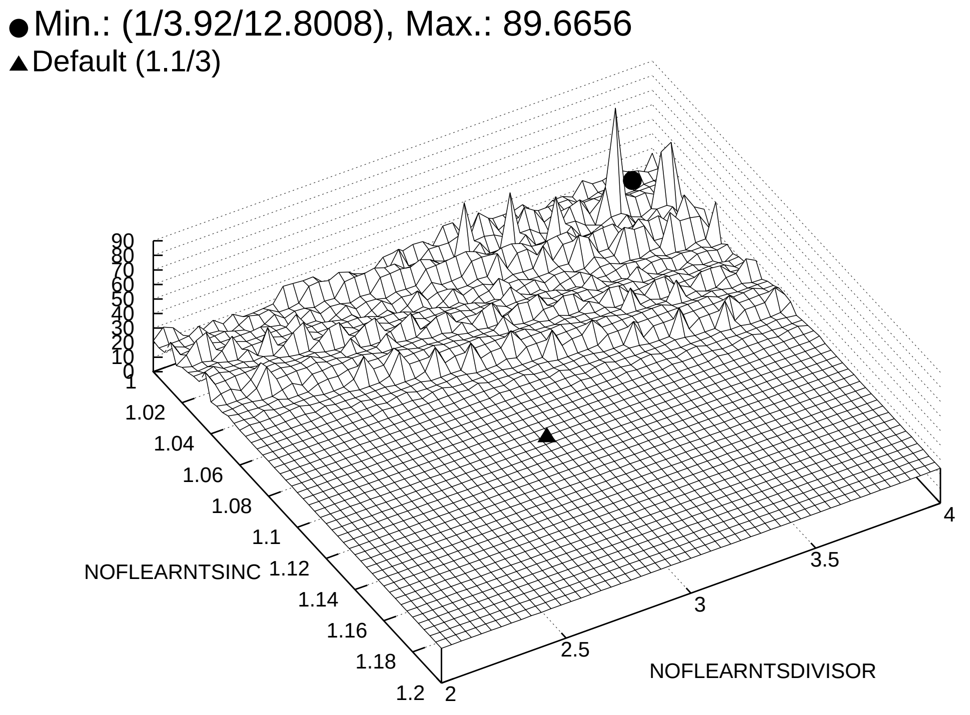


Figure 6.42: Fitness landscapes of problem `manol-pipe-c7_i` for NOFLEARNTSINC / NOFLEARNTSDIVISOR using the full range (top, 51×51 grid, timeout 120s, **Pc0**) and a close-up (bottom, 51×51 grid, timeout 120s, **Pc0**)

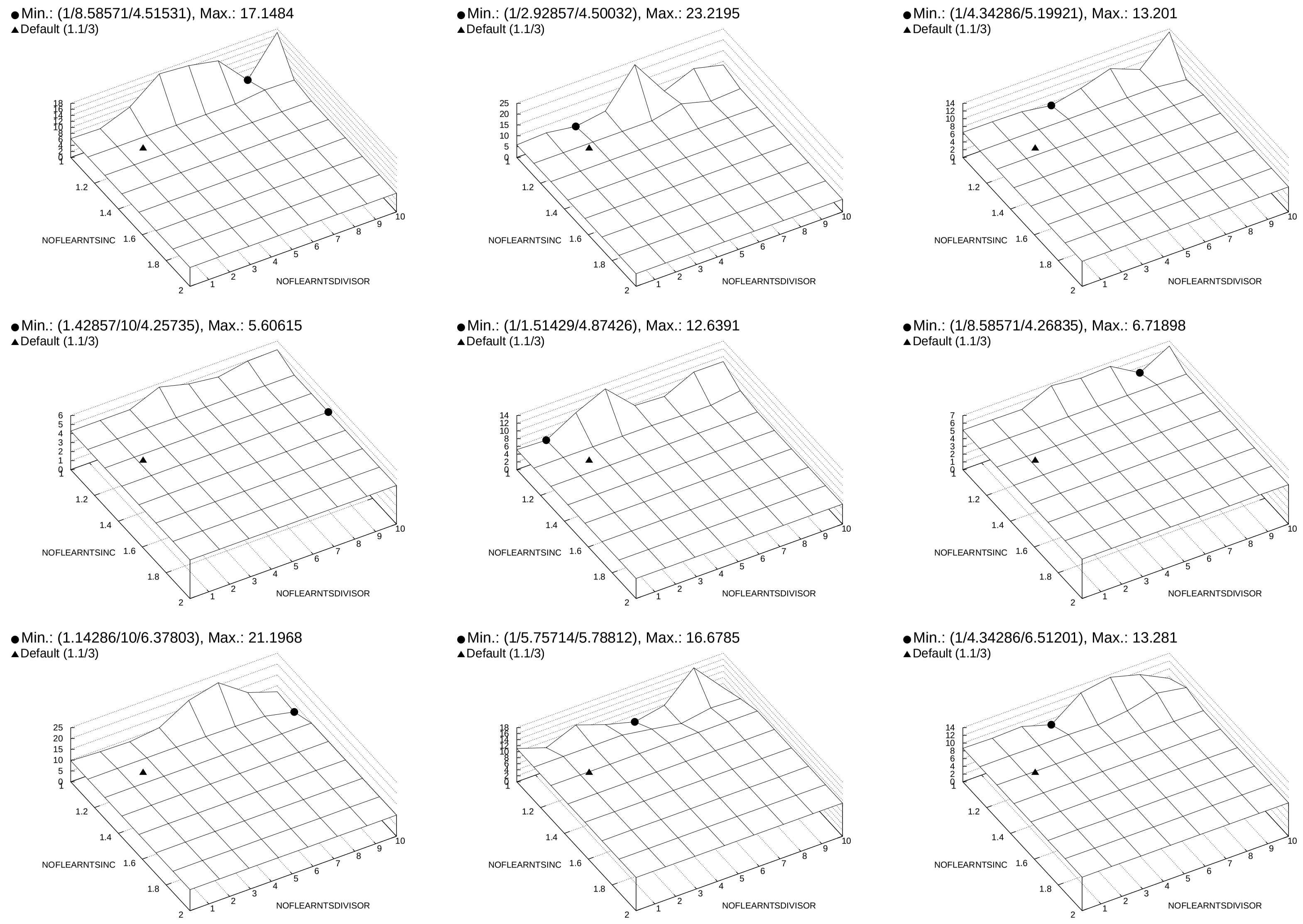
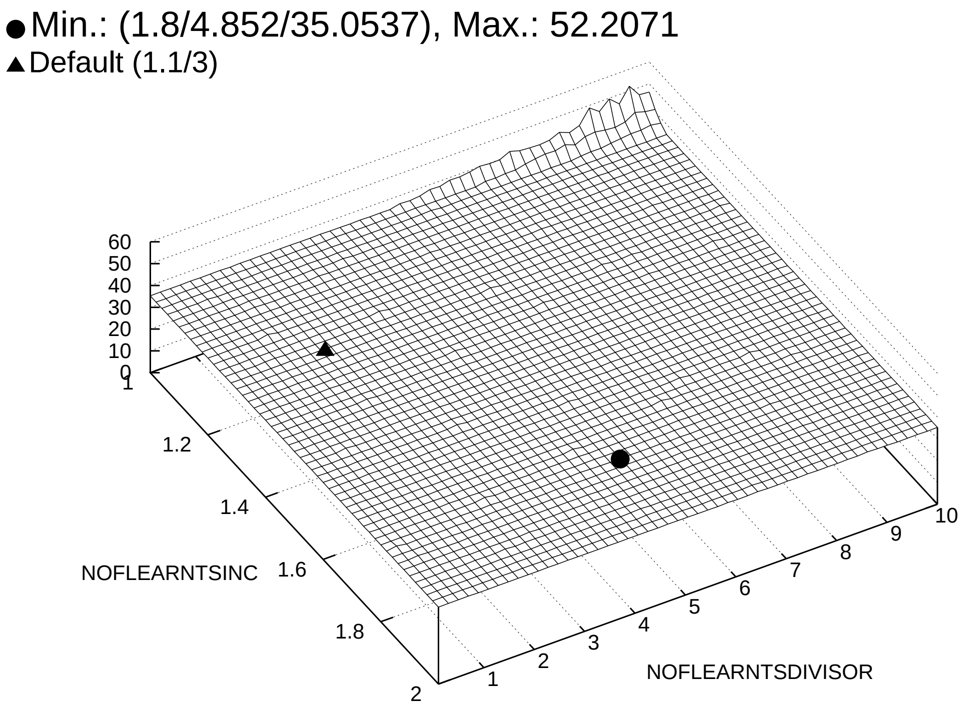


Figure 6.43: Fitness landscapes of 9 shuffled versions of problem `manol-pipe-c7_i` for NOFLEARNTSINC / NOFLEARNTSDIVISOR using the full range (8×8 grid, timeout 120s, **`Pc3`**)



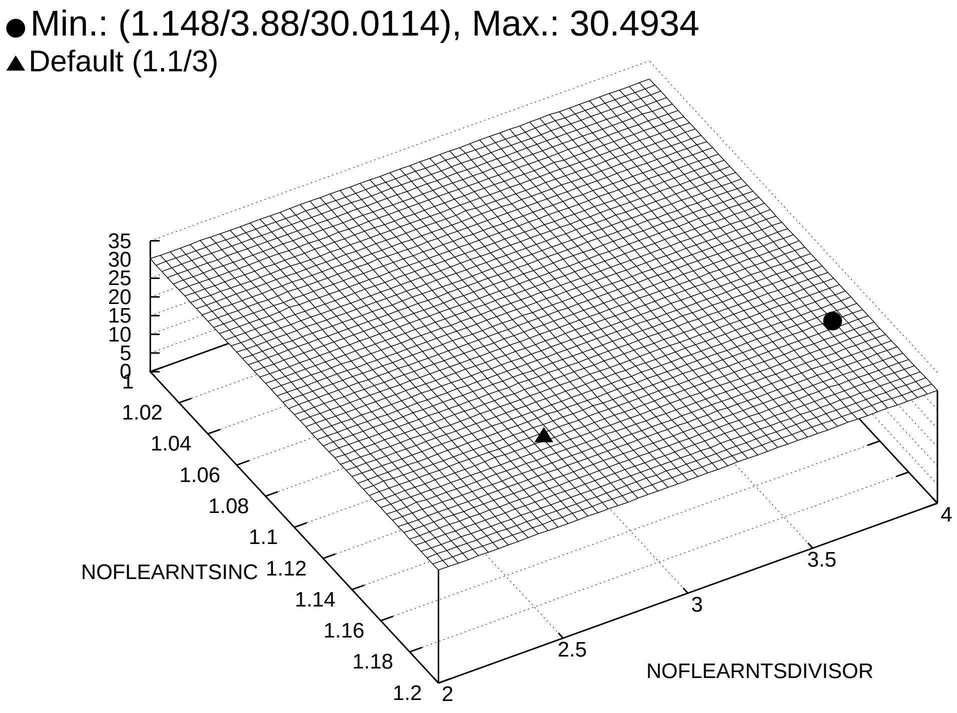


Figure 6.44: Fitness landscapes of problem `goldb-heqc-desmul` for NOFLEARNTSINC / NOFLEARNTSDIVISOR using the full range (top, 51×51 grid, timeout 300s, **Pc2**) and a close-up (bottom, 51×51 grid, timeout 300s, **Pc4**)

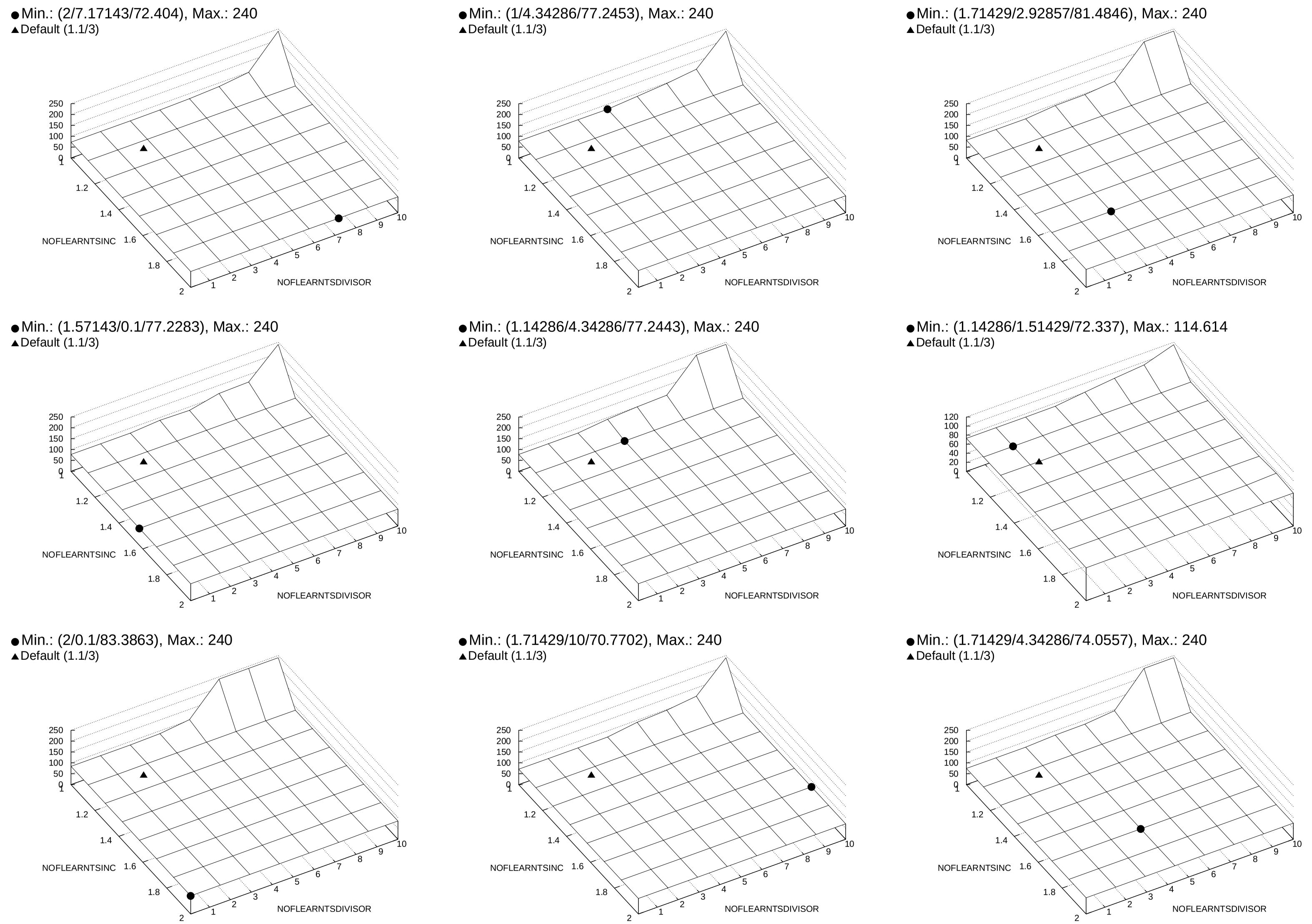
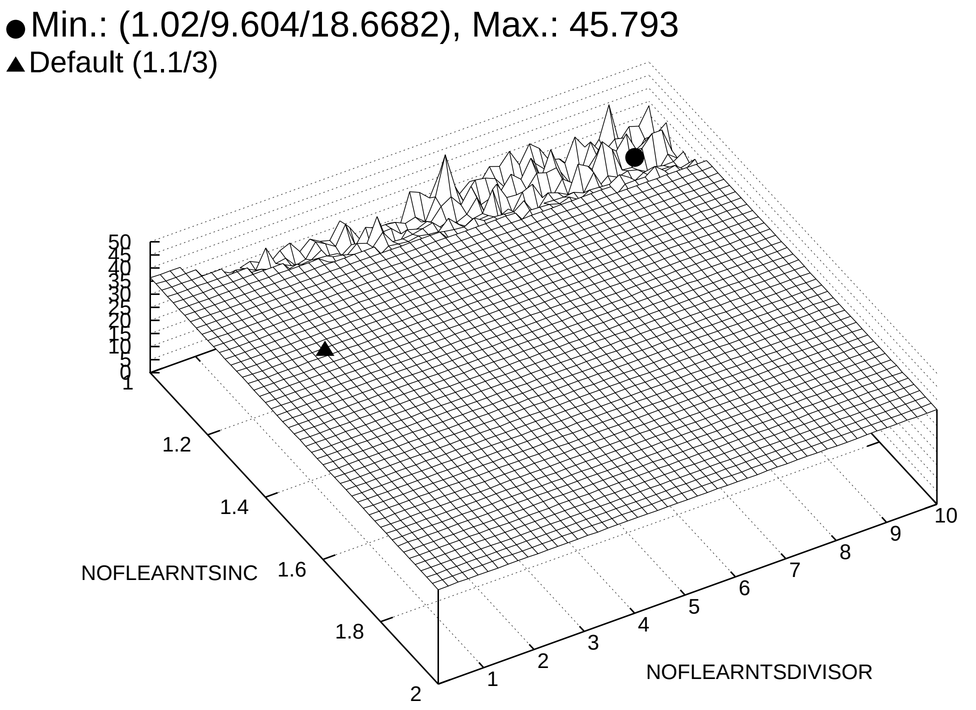


Figure 6.45: Fitness landscapes of 9 shuffled versions of problem `goldb-heqc-desmul` for NOFLEARNTSINC / NOFLEARNTSDIVISOR using the full range (8×8 grid, timeout 120s, **`Pc1`**)



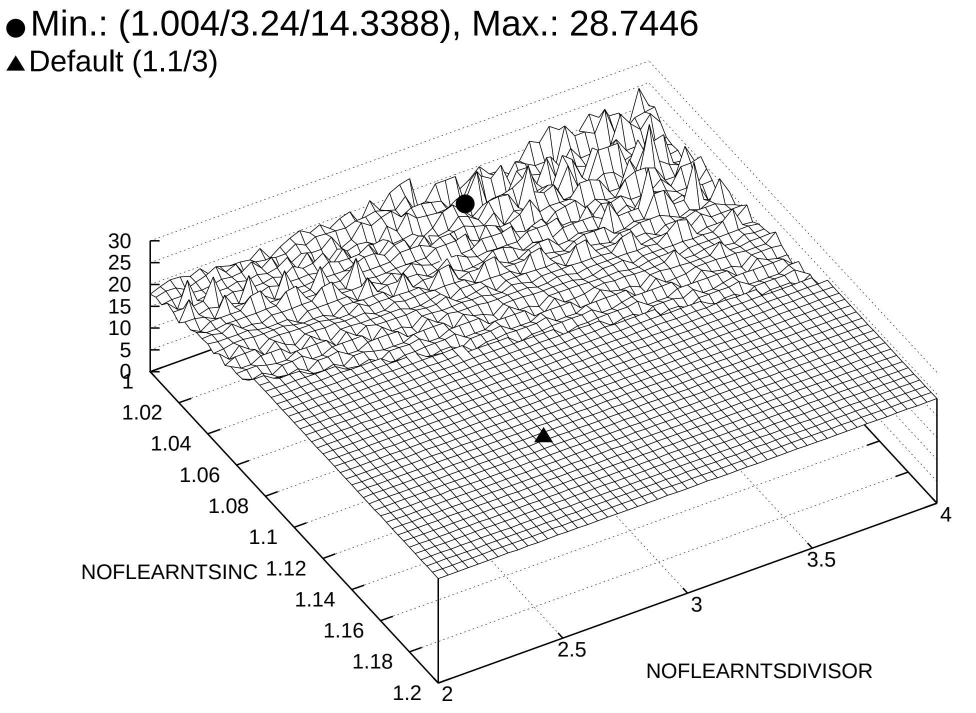


Figure 6.46: Fitness landscapes of problem `velev-eng-uns-1.0-04a` for NOFLEARNTSINC / NOFLEARNTSDIVISOR using the full range (top, 51×51 grid, timeout 300s, **`Pc2`**) and a close-up (bottom, 51×51 grid, timeout 300s, **`Pc4`**)

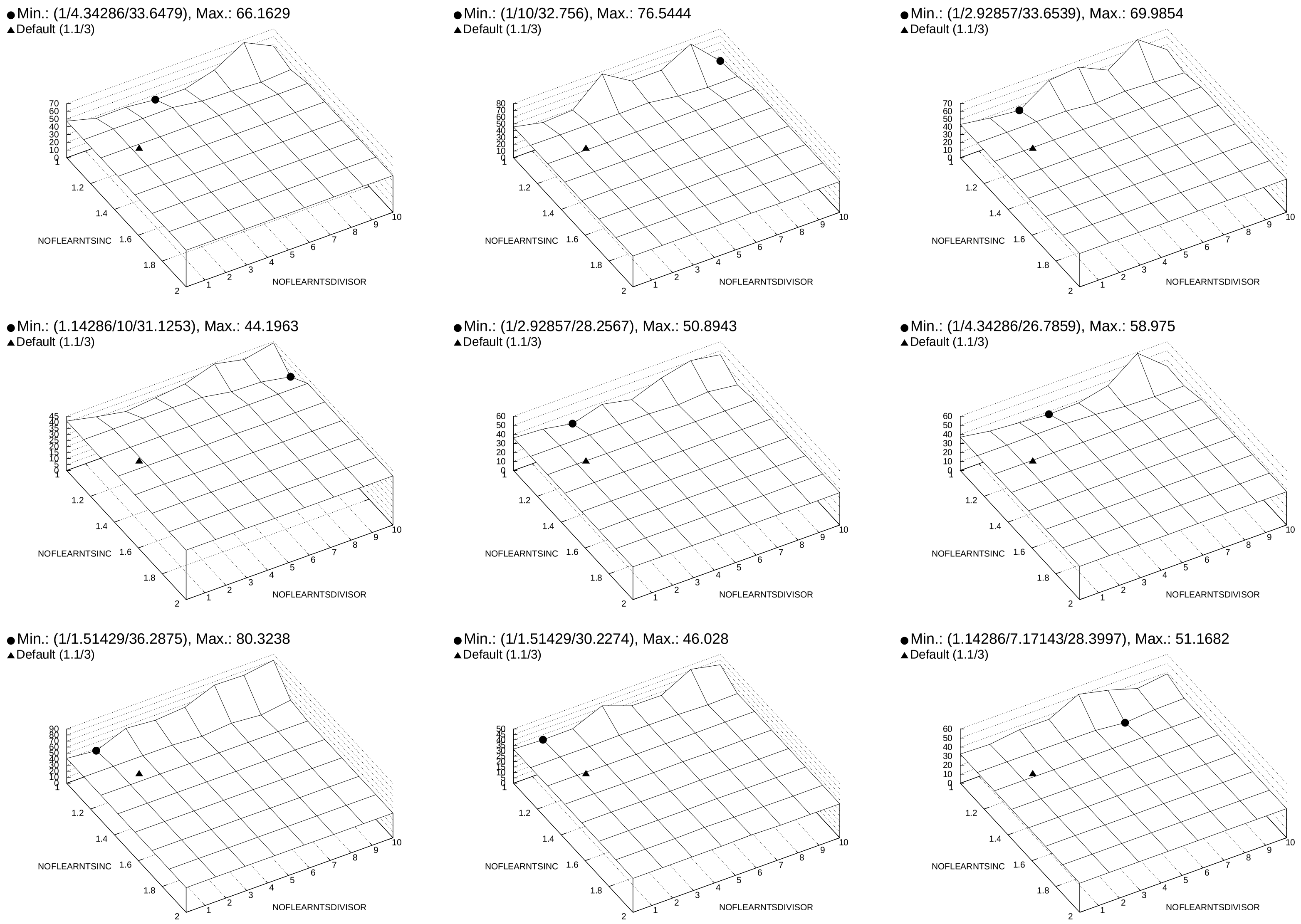
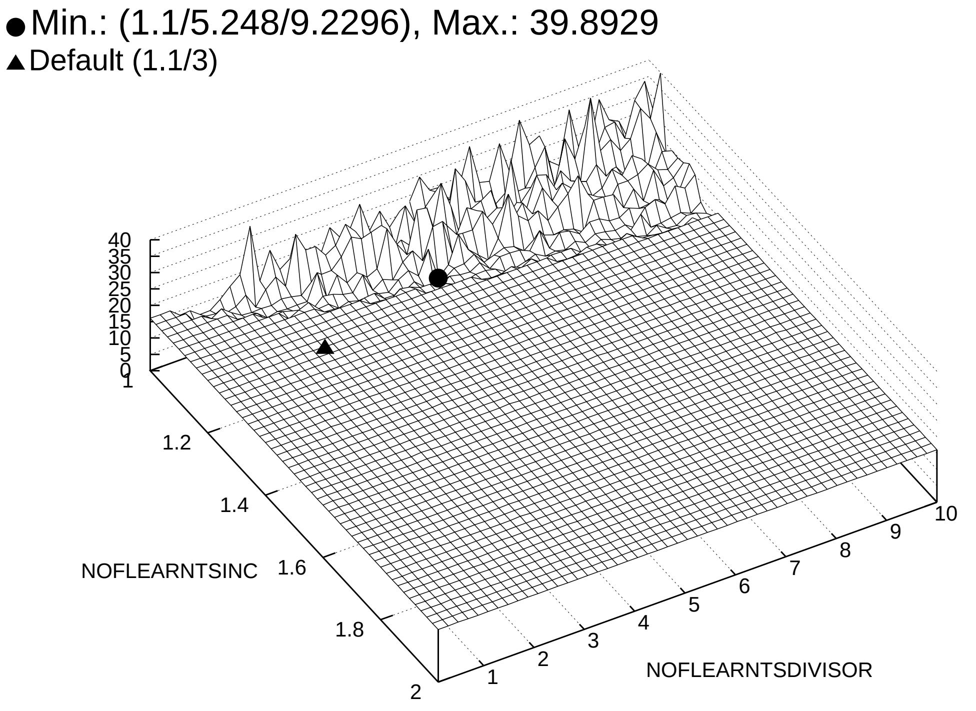


Figure 6.47: Fitness landscapes of 9 shuffled versions of problem `velev-eng-uns-1.0-04a` for NOFLEARNTSINC / NOFLEARNTSDIVISOR using the full range (8×8 grid, timeout 120s, **`Pc1`**)



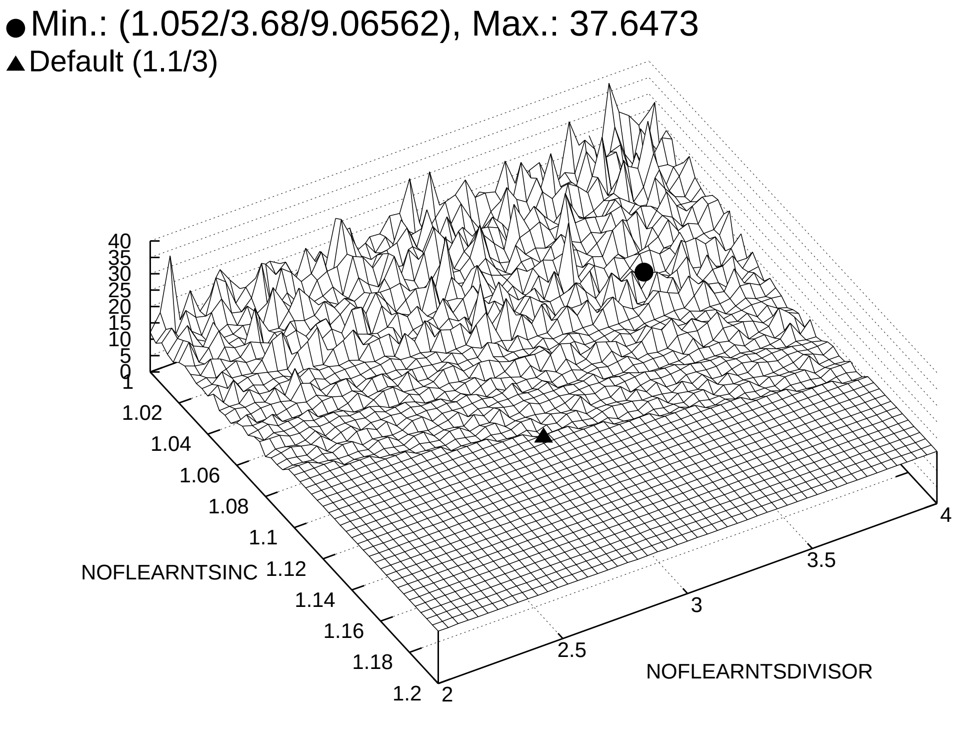


Figure 6.48: Fitness landscapes of problem `velev-npe-1.0-02` for NOFLEARNTSINC / NOFLEARNTSDIVISOR using the full range (top, 51×51 grid, timeout 120s, **`Pc2`**) and a close-up (bottom, 51×51 grid, timeout 120s, **`Pc2`**)

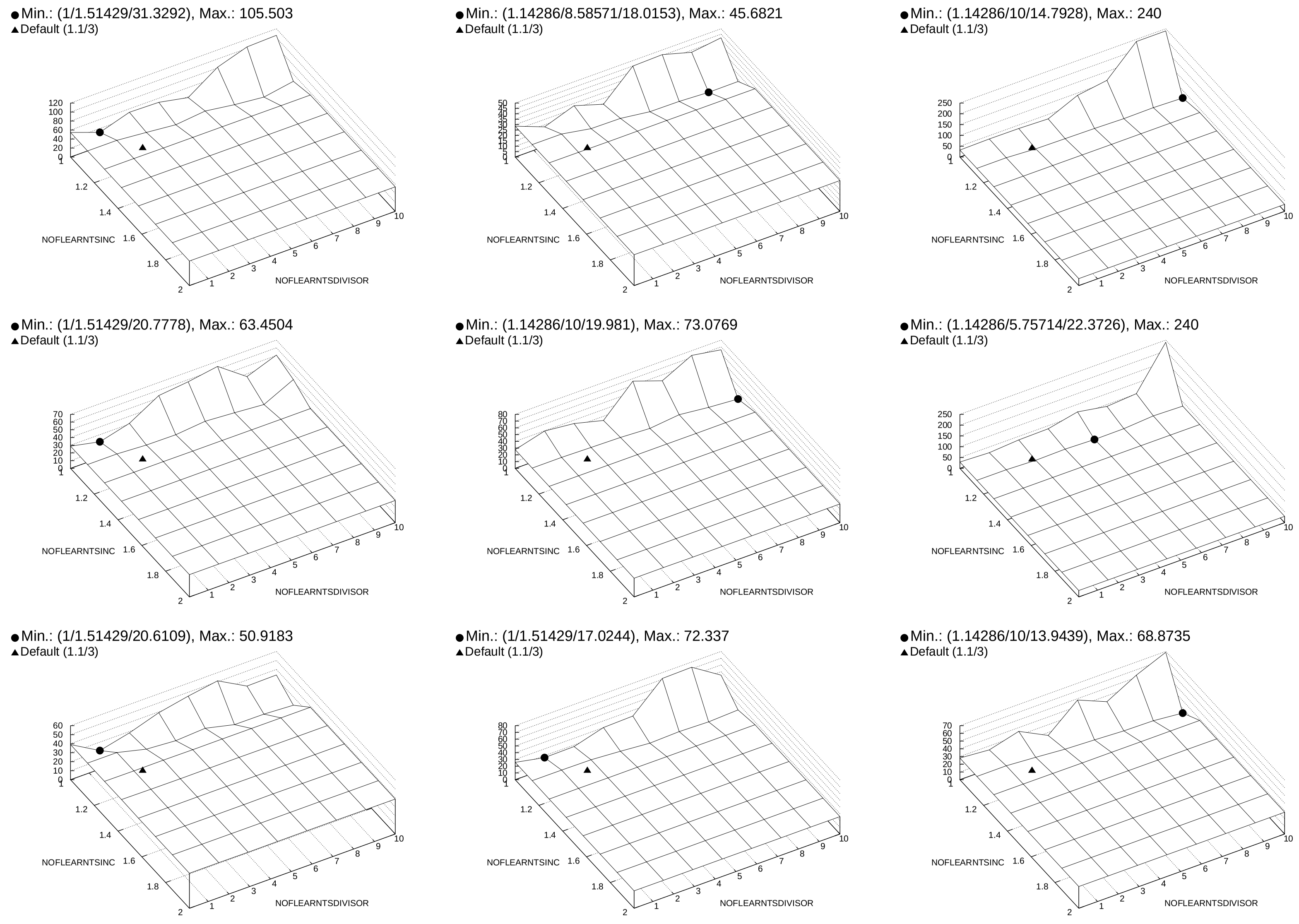


Figure 6.49: Fitness landscapes of 9 shuffled versions of problem `velev-npe-1.0-02` for NOFLEARNTSINC / NOFLEARNTSDIVISOR using the full range (8×8 grid, timeout 120s, **`Pc2`**)

6.4.6 RANDOMVARFREQ

The remaining parameter RANDOMVARFREQ determines the probability with which the decision heuristic chooses a random variable rather than the one with the currently highest activity. The (two-dimensional) fitness landscapes in Fig. 6.50 to Fig. 6.52 chart the solving time for the test problems for different values of RANDOMVARFREQ over the full range.

6.4.6.1 Observations

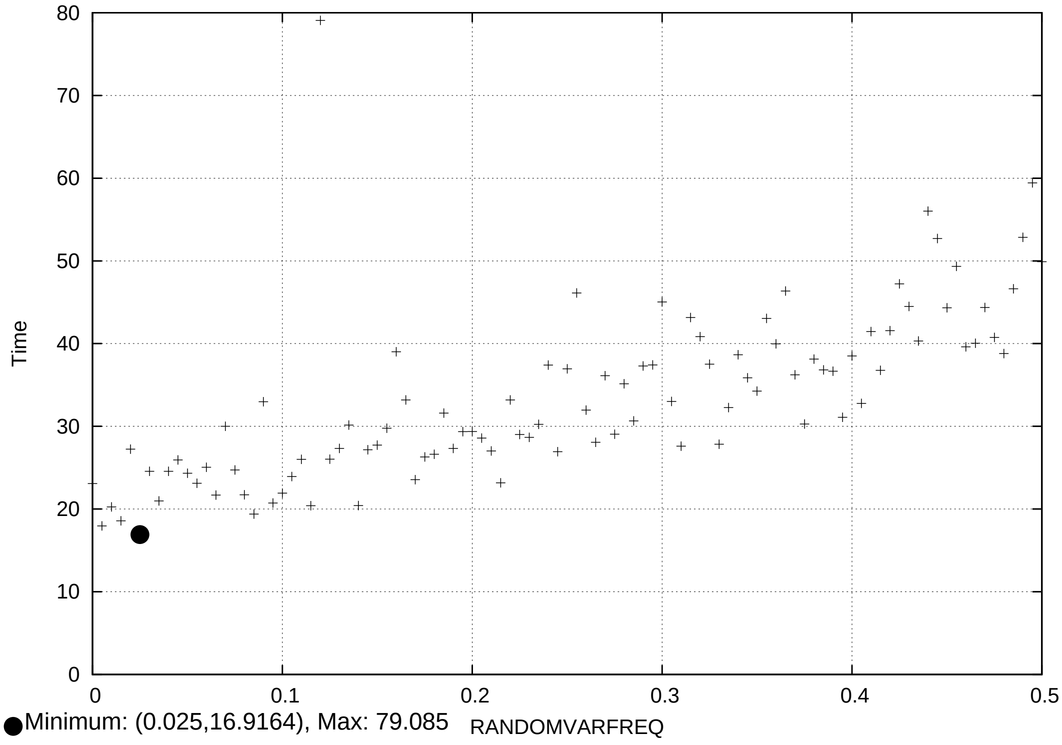
1. The data points form rough "tube" shapes that are slightly upwardly inclined. The widths of the tubes are different for each problem, ranging from about 3 seconds up to about 20 seconds.

2. The general trend is that solving times get worse the higher RANDOMVARFREQ is but due to the spread a solving time can also be lower for a point further to the right.

3. The best solving times are found near the lower end of the range; only in one case (for problem `goldb-heqc-desmul`) out of the five shown was the minimum at exactly 0 (meaning no random decisions). In four out of five cases the solving time was better with some amount of randomness (ranging from 2.5% to 12.5%).

6.4.6.2 Interpretation

The reasoning behind using rare random decisions is that it helps the solver solve some problems without causing too much overhead in other problems. The fitness landscapes seem to confirm this: in four out of five cases the best solving time was achieved with some added randomness, and in the one case it did not help the solving times with a small amount of randomness are not substantially higher.



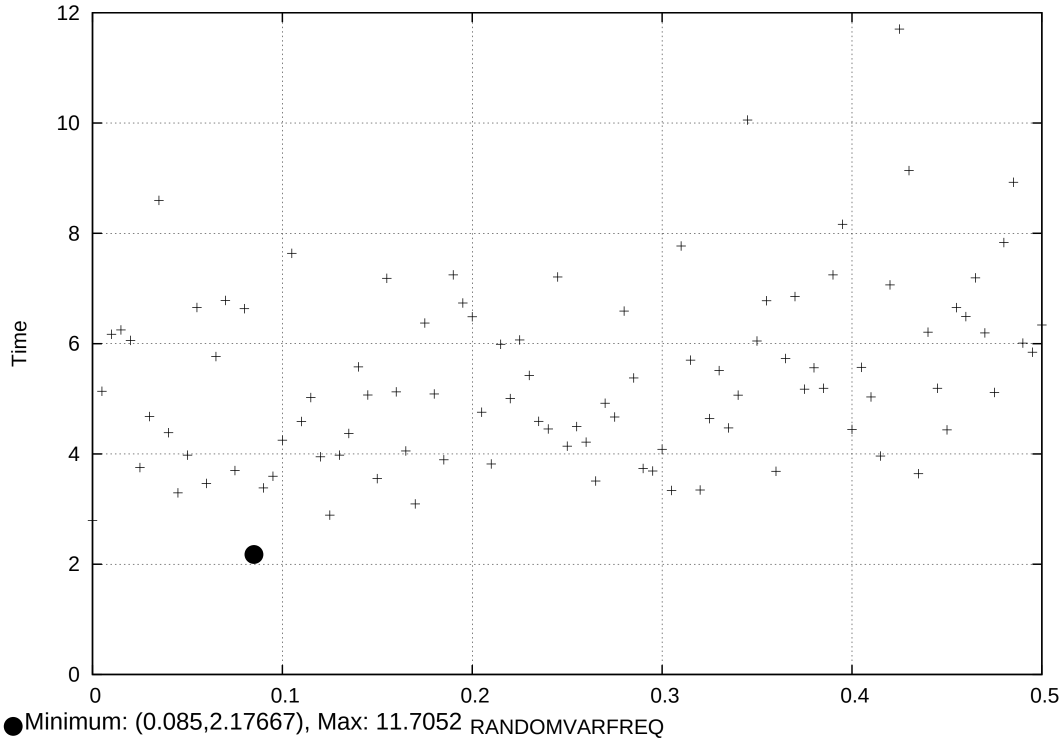
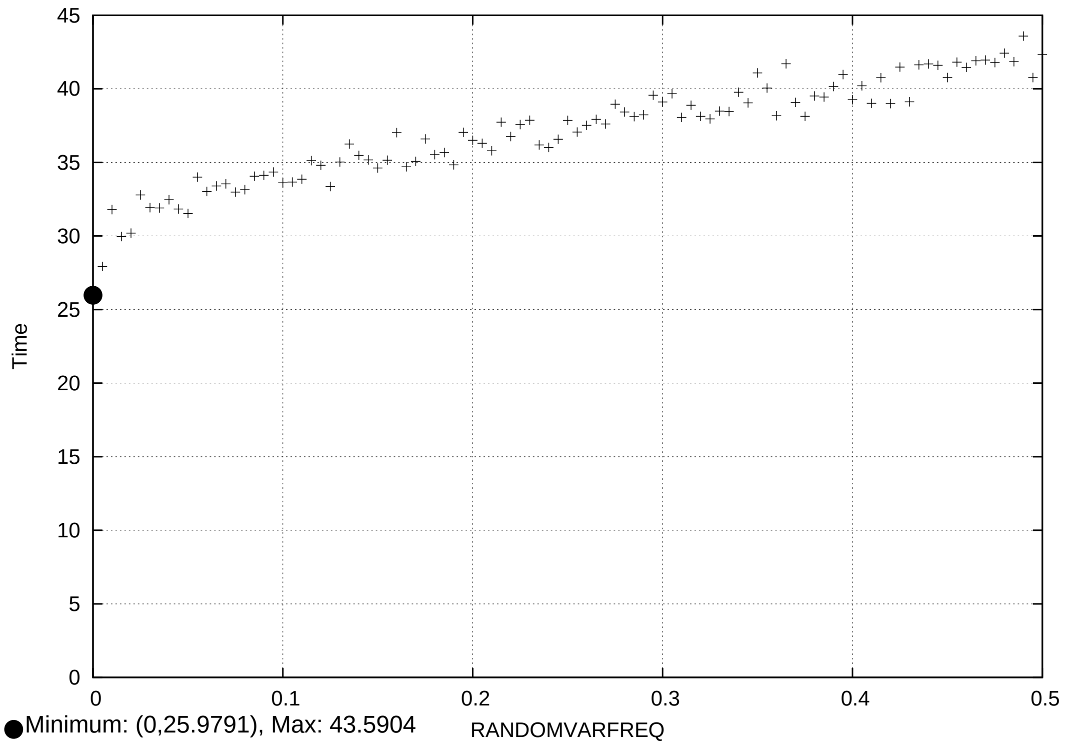


Figure 6.50: Fitness landscapes for RANDOMVARFREQ (default: 0.02) of problems `manol-pipe-c6n` (top) and `manol-pipe-c7_i` (bottom), both using 100 points, timeout 120s on **`Pc4`**



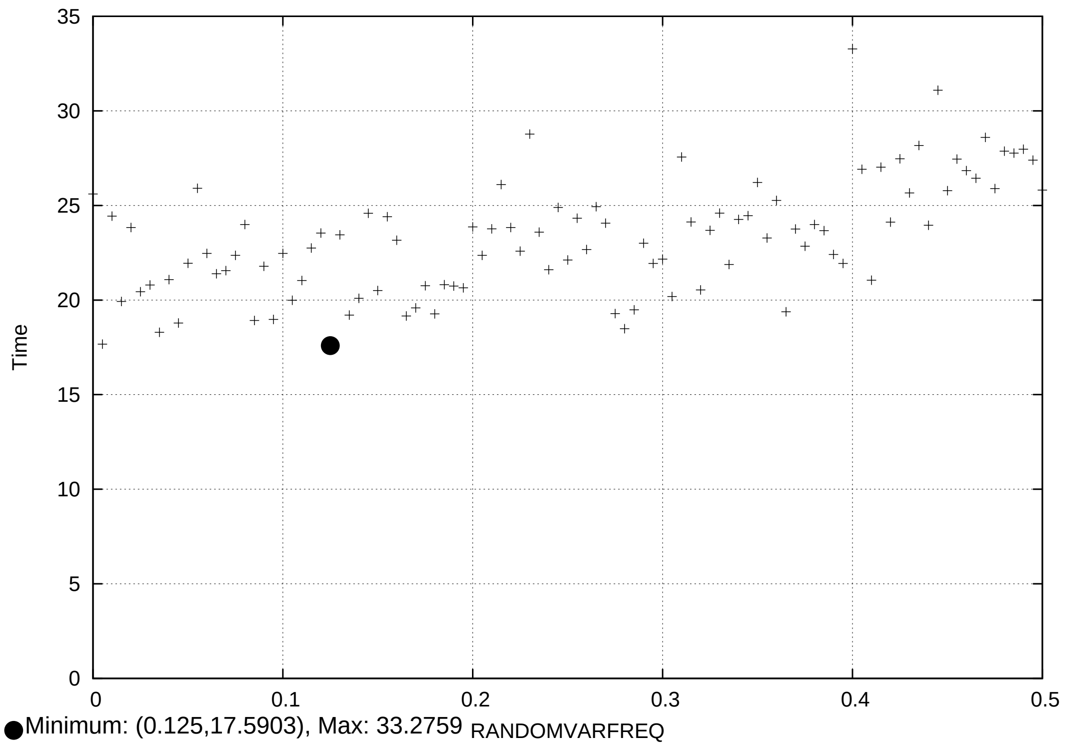


Figure 6.51: Fitness landscapes for RANDOMVARFREQ (default: 0.02) of problems `goldb-heqc-desmul` (top) and `velev-eng-uns-1.0-04a` (bottom), both using 100 points, timeout 120s on **`Pc4`**

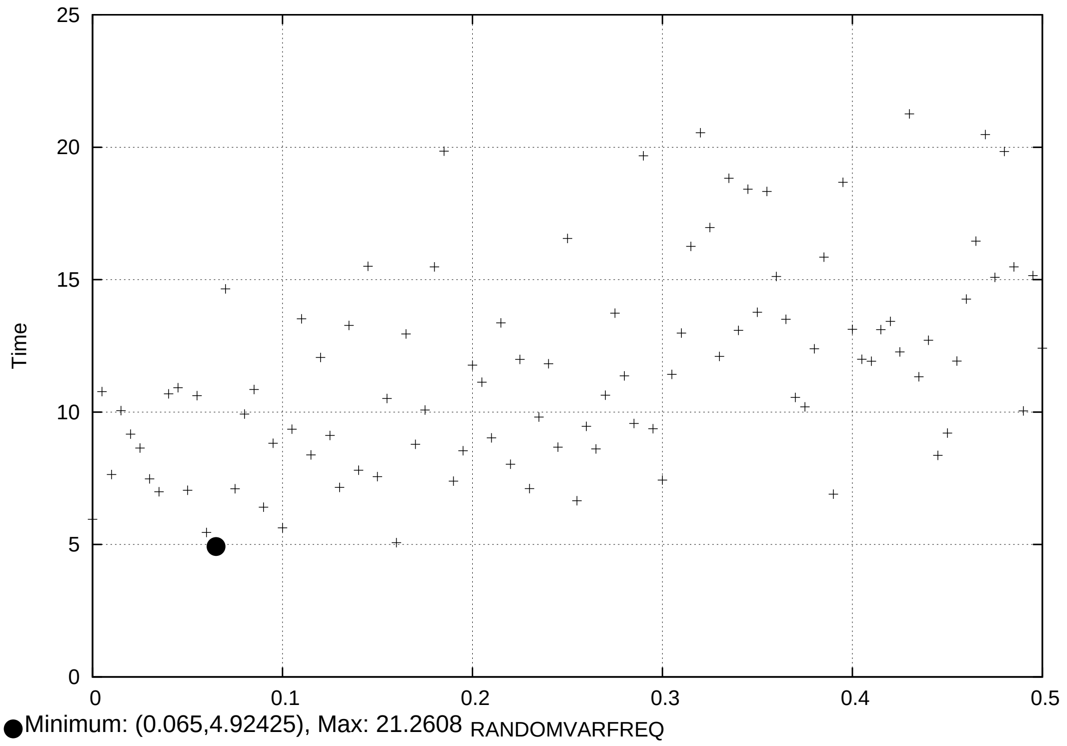


Figure 6.52: Fitness landscapes for RANDOMVARFREQ (default: 0.02) of problem `velev-npe-1.0-02` using 100 points, timeout 120s on **`Pc4`**

6.4.7 Conclusion

In this section, fitness landscapes were generated for the seven MINISAT parameters using five different SAT problems and their shuffled versions. In summary, following conclusions could be drawn:

- The landscapes of the parameters $\mathrm{VARDECAY}$ and $\mathrm{CLAUSEDECAY}$ are similar for all problems and their shuffled versions. Small variations of $\mathrm{VARDECAY}$ have a stronger impact on solving time than variations of $\mathrm{CLAUSEDECAY}$, which can be seen on ridges in the landscape extending in the direction of the $\mathrm{CLAUSEDECAY}$ axis. The default values $\mathrm{VARDECAY} = 0.95$ and $\mathrm{CLAUSEDECAY} = 0.999$ used in MINISAT are essentially optimal.

- The parameters $\mathrm{NOFCONFLICTSINC}$ and $\mathrm{NOFCONFLICTSBASE}$ strongly affect the solving time, but their effects are so random and so sensitive even to tiny changes that no recommendation can be made for their choice apart from that extremely frequent restarts (around the point $\mathrm{NOFCONFLICTSINC} = 1$ and $\mathrm{NOFCONFLICTSBASE} = 10$) should be avoided. The default parameters seem as good a choice as any.

- The landscapes for $\mathrm{NOFLEARNTSINC}$ and $\mathrm{NOFLEARNTSDIVISOR}$ are divided into a "mountainous zone" where clause deletion is frequent and a "flat zone" where it is relatively rare. In the mountainous zone the solving times fluctuate strongly and may be substantially shorter than in the flat zone, but may also be longer; no rule is obvious how to set the parameters so that the solving time is shorter. In the flat zone the solving time is the same everywhere; the default parameters are located in the flat zone.

- The landscapes for shuffled problems are in general similar to those of the original problems apart from localized differences. This was not tested for $\mathrm{NOFCONFLICTSINC}$ and $\mathrm{NOFCONFLICTSBASE}$ since those landscapes were very random anyway.

6.5 SAT optimization with GA and ES

6.5.1 Objective of the experiment

In this experiment, the GA and ES will be used to optimize a varying number of MINISAT parameters (using the ranges given in Sec. 6.4.2.2) by running the solver on SAT problems and attempting to minize the run time. This is the core operation of the SAT solver optimization procedure (see Sec. 1.8.2.1): if this operation is too unreliable or even impossible, the entire procedure can not work. While the ES was shown to be faster and more precise than the GA when finding the optimum of noisy test functions (see Sec. 6.3), tests are necessary to determine if the situation is the same when the actual SAT solver is under optimization. The standard EA parameters described in Sec. 6.2.3 will be used.

6.5.2 Setup

Since a single run does not provide reliable evidence of an EA's efficiency, many runs are needed; 10 runs total were executed for each partial experiment. Due to the very large runtimes involved when optimizing the SAT solver the experiment had to be performed on several computers, one distinct run on each PC, to be able to finish an acceptable number of runs in the available time. The experiments were performed on several different computers; run times reported here will be given normalized to the reference machine (see Sec. 6.4.2.1). On that machine, the SAT Race problem `manol-pipe-c6n` takes a little less than 1 minute to solve and was chosen as the target problem. The GA and ES start with an initial population of random individuals and will attempt to find a configuration of the parameters that result in the SAT problem being solved as quickly as possible.

6.5.3 Optimizing one parameter

In the first attempt at optimization only the VARDECAY parameter was the target of optimization, while all others were held at their default values. The GA and ES were then run and the progress recorded. The fitness landscape experiment (Sec. 6.4) showed that VARDECAY has a strong influence on solving times, and expected optimal results were values near but not equal to 1.

Tab. 6.5 shows the best results found over all generations of the 10 GA and ES runs using `manol-pipe-c6n` as the training problem. The solving times given in the table are normalized to the reference machine and sorted from lowest to highest. Also shown are the minimum, maximum, median, average, standard deviation and standard deviation in percent of the average of each column. The table shows that both the GA and the ES found their best results in the range between ca. 0.9 and 0.97, with an average of 0.936 for the GA and 0.94 for the ES (the default value of

#	Result GA	Time GA	Result ES	Time ES
1	0.944	24.671	0.928	23.14
2	0.947	24.704	0.928	23.22
3	0.939	24.830	0.960	23.99
4	0.958	24.902	0.919	24.02
5	0.899	25.043	0.946	24.11
6	0.973	25.470	0.946	24.16
7	0.937	25.831	0.939	24.47
8	0.909	27.052	0.937	24.56
9	0.922	27.575	0.949	25.72
10	0.937	28.202	0.954	25.73
Min.	*0.899*	*24.671*	*0.919*	*23.143*
Max.	*0.973*	*28.202*	*0.960*	*25.732*
Median	*0.938*	*25.256*	*0.942*	*24.133*
Avg.	*0.936*	*25.828*	*0.940*	*24.312*
Std. dev.	*0.022*	*1.308*	*0.013*	*0.875*
Std. dev. %	*2.34%*	*5.06%*	*1.37%*	*3.60%*

Table 6.5: Best results of 10 GA and ES runs optimizing VARDECAY for training problem `manol-pipe-c6n`

VARDECAY in MINISAT is 0.95). This is as expected, considering the fitness landscape for the training problem `manol-pipe-c6n` constructed in the last section. The spread of the results is slightly lower for the ES than for the GA, though with such a small sample and with the amount of noise involved the significance of this is debatable and will not be considered valid evidence of the ES's superiority yet.

Fig. 6.53 charts the average over all runs of the best result in each generation over the course of 50 generations for the GA and the ES, given as a fraction of the timeout (if the solving time was 30 seconds and the timeout was 60 seconds, the charted value would be 0.5). Both the GA and the ES were capable of successfully navigating the search space of the problem and found values of VARDECAY that are in line with what was expected after computing the fitness landscape. The ES seemed to be slightly more efficient than the GA, similar to the situation in the experiments with noisy test functions, since the graph of the ES reaches lower (better) values quicker than the GA.

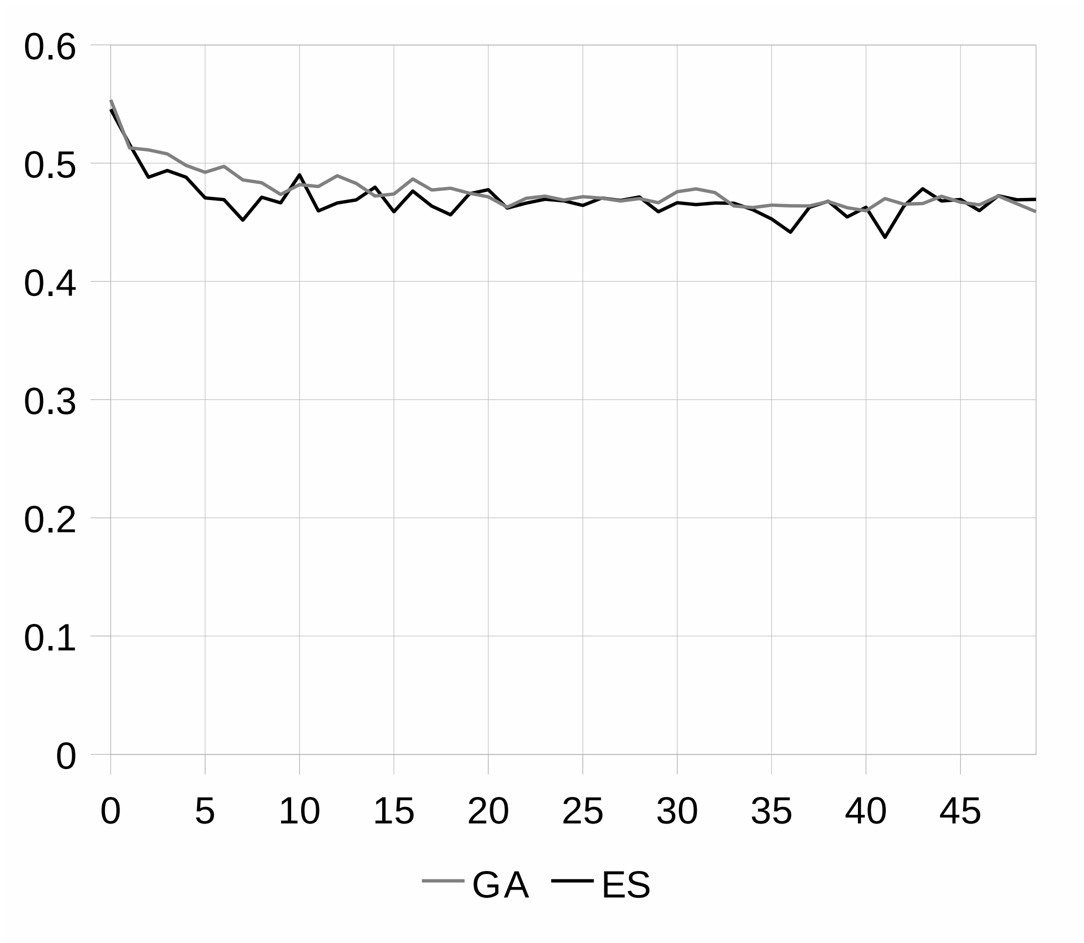


Figure 6.53: Progression of the lowest solving time (given as a fraction of the timeout) per generation of the GA and ES optimizing only VARDECAY for training problem `manol-pipe-c6n`

6.5.4 Optimizing the complete parameter set

Next, both GA and ES were used to optimize the entire parameter set, again on problem `manol-pipe-c6n`. Tab. 6.6 and Tab. 6.7 show the best results found in each run (run times are normalized to the reference machine). The entry "% of range" in the tables gives the ratio of the entire search range for the respective variable into which the results fall.

The best run times of all 10 runs have a somewhat higher spread for the GA than for the ES. It was noticeable in the experiments with test functions (see Sec. 6.2 and Sec. 6.3) that the GA tends to be less good at pinpointing optima than the ES, which seems to be the case here as well.

In both tables that the results for the parameters VARDECAY, CLAUSEDECAY and RANDOMVARFREQ are significantly more localized than those of the parameters NOFCONFLICTSINC, NOFCONFLICTSBASE, NOFLEARNTSINC and NOFLEARNTSDIVISOR. For the GA, the results for the first set of parameters cover only a range of 8-33% of the full allowed range, and the ES 3-16%. The results of the second set of parameters cover a range of 65-94% (GA) and 68-100% (ES) of the full range.

It was shown in Sec. 6.4 that the fitness landscape for VARDECAY / CLAUSEDECAY has a global minimum in the region near the point VARDECAY $= 1$ and CLAUSEDECAY $= 1$; this minimum was relatively universal for all the tested problems as well as their shuffled versions. Both the GA and the ES return results for these parameters that are in the general vicinity of this optimum, though the ES seems to pinpoint the results more exactly.

The parameters NOFCONFLICTSINC, NOFCONFLICTSBASE, NOFLEARNTSINC and NOFLEARNTSDIVISOR have in common that the fitness landscapes generated from them are highly multimodal, though for different reasons. The fitness landscapes for NOFCONFLICTSINC / NOFCONFLICTSBASE are very random and "sandpaper"-like, with a huge number of very narrow local minima. These landscapes also look the same anywhere in the allowed region apart from one small "raised corner", which makes it impossible to decide which of these might be better than any other. In contrast, the landscapes for NOFLEARNTSINC / NOFLEARNTSDIVISOR contain large areas which are completely flat (in addition to a "mountainous zone" where solving times are very random), so that there is a large set of possible and equivalent choices for the values. The EAs in this experiment had to navigate through the multidimensional search space that includes both of these parameter pairs, therefore it is not surprising that their values in the results are from a fairly large range (or even the entire range).

No fitness landscapes were generated that included the parameter RANDOMVARFREQ, but it is known that this value (which controls the frequency of random choices in the decision heuristic) must not be too high; the default value is only 0.02 (2%). The best results of the GA had values for

RANDOMVARFREQ ranging from less than 1% to about 6%, the ES from 0% to a little less than 2%. This indicates that random variable decisions must be very rare or they disrupt the search process too strongly.

Fig. 6.54 charts the average over all runs of the best result in each generation over the course of 50 generations for the GA and the ES, given as a fraction of the timeout. Although the initial populations were apparently worse for the ES, it reaches the best results quicker than the GA. After less than 5 generations the ES surpasses the GA's results and after about 15 generations it has made the majority of progress, whereas the GA needs about 35 generations until it reaches its best results.

It is interesting that both EAs are not "confused" by the parameters with very random effects on the run time while simultaneously optimizing seven different parameters. This is an encouraging result that suggests that EAs are useful for optimization even when faced with highly noisy (apart from measurement imprecisions) multidimensional problems.

6.5.5 Conclusion

The GA and ES were used to optimize the parameters of MINISAT on the SAT problem `manol-pipe-c6n`, first just one parameter, then all seven simultaneously. In both cases the EAs were successful in gradually improving the best solving time over the generations, showing that either of them can serve as the optimization algorithm for the SAT solver optimization procedure. It was found that the ES converges faster than the GA; this is in line with the findings on noisy test functions in Sec. 6.3. The best results found by both algorithms are of comparable quality, though the ES has superior precision; this was also as expected after the earlier findings. An encouraging result of the experiments was that the EAs are not confused by the highly random effects of four of the parameters and can optimize the remaining three parameters effectively.

For the parameters VARDECAY, CLAUSEDECAY and RANDOMVARFREQ the EAs find values that are very close to the defaults (0.95, 0.999 and 0.02) used in MINISAT, the latter two of which were in turn confirmed to be near optimal by the fitness landscape experiments in Sec. 6.4. The fitness landscapes also indicated that a small but not too high amount of randomness (meaning values around 0.1 and less) in the variable decision heuristic is beneficial for the solver; the best results of the EA optimization had small values for RANDOMVARFREQ.

It is noticeable for parameter NOFCONFLICTSINC that the median value of the results found by the ES is a very high 1.939, which is close to the maximum allowed value; the (usually less precise) GA found values from the entire range. For the parameters NOFCONFLICTSBASE, NOFLEARNTSINC and NOFLEARNTSDIVISOR neither of the EAs found clear results. The fitness landscapes of these parameters were found to be highly multimodal earlier, which explains this result: there are are large number of viable choices for these parameters

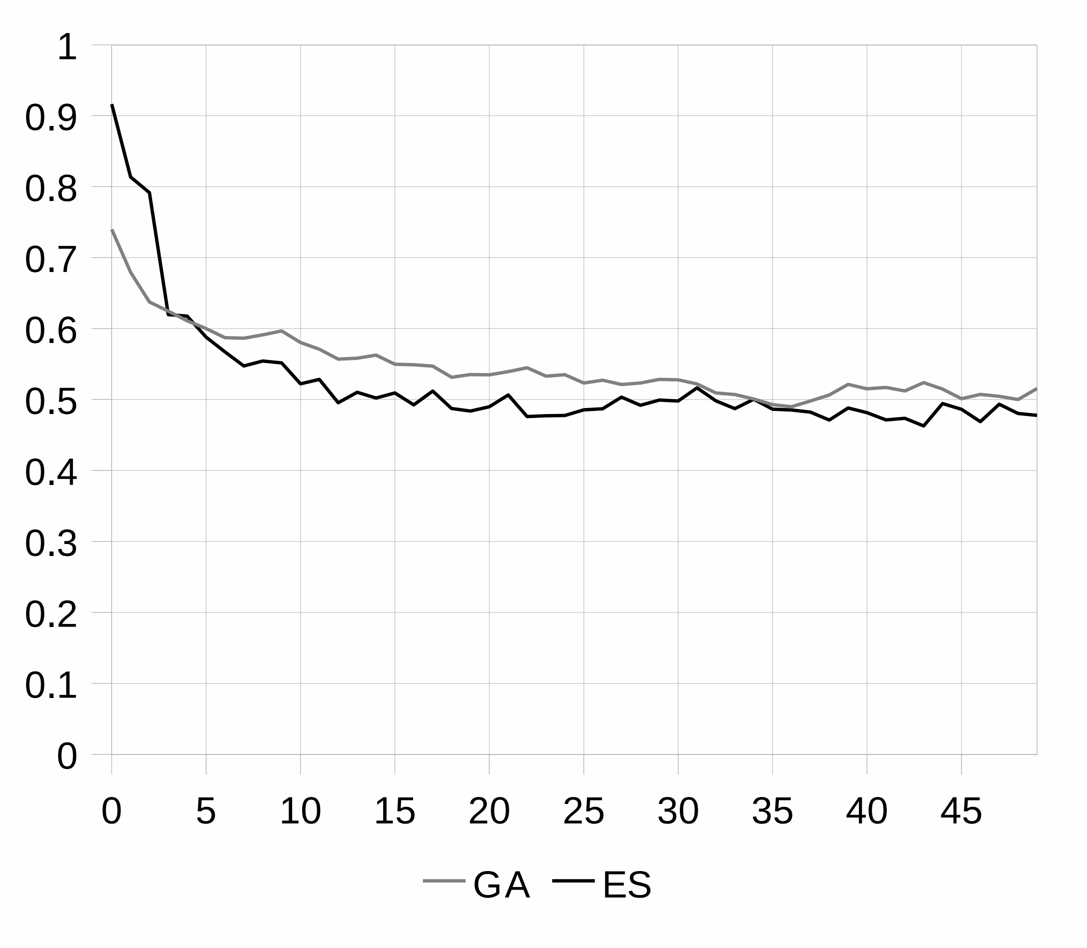


Figure 6.54: Progression of the lowest solving time (given as a fraction of the timeout) per generation of the GA and ES optimizing all seven solver parameters for training problem `manol-pipe-c6n`

which had no discernable order to them. It is understandable then that the EAs did not converge on any particular value, but rather resulted in any of the great number of possible choices.

#	VAR DECAY	CLAUSE DECAY	NOF CONFLICTS INC	NOF CONFLICTS BASE	NOF LEARNTS INC	NOF LEARNTS DIVISOR	RANDOM VARFREQ	Time
1	0.964	0.824	1.861	219	1.988	9.785	0.020	21.1
2	0.984	0.635	1.616	525	1.908	9.848	0.007	24.2
3	0.968	0.918	1.471	563	1.543	1.859	0.043	24.5
4	0.940	0.850	1.276	26	1.595	7.384	0.009	25.9
5	0.942	0.922	1.398	347	1.152	0.944	0.057	26.3
6	0.953	0.965	1.380	109	1.752	4.039	0.036	27.5
7	0.920	0.782	1.547	86	1.092	0.531	0.027	28.1
8	0.919	0.955	1.859	443	1.890	7.590	0.061	28.9
9	0.902	0.713	1.481	423	1.367	2.526	0.064	31.9
10	0.904	0.861	1.049	675	1.050	7.639	0.044	42.3
Default	*0.95*	*0.999*	*1.5*	*100*	*1.1*	*3*	*0.02*	
Range	*0.01-1*	*0.01-1*	*1-2*	*10-1000*	*1-2*	*0.1-10*	*0-0.5*	
Min.	**0.902**	**0.635**	**1.049**	**26**	**1.050**	**0.531**	**0.007**	**21.1**
Max.	**0.984**	**0.965**	**1.861**	**675**	**1.988**	**9.848**	**0.064**	**42.3**
Med.	**0.941**	**0.856**	**1.476**	**385**	**1.569**	**5.711**	**0.039**	**26.9**
Avg.	***0.940***	***0.843***	***1.494***	***342***	***1.534***	***5.215***	***0.037***	***28.1***
Std. dev.	***0.028***	***0.108***	***0.248***	***222.422***	***0.354***	***3.631***	***0.021***	***5.8***
Std. dev. %	**3.0%**	***12.8%***	***16.6%***	***65.1%***	***23.1%***	***69.6%***	***56.5%***	***20.7%***
% of range	***8.3%***	***33.4%***	***81.2%***	***65.5%***	***93.8%***	***94.1%***	***11.3%***	

Table 6.6: Best results of 10 GA runs optimizing all seven solver parameters for training problem `manol-pipe-c6n`

#	VAR DECAY	CLAUSE DECAY	NOF CONFLICTS INC	NOF CONFLICTS BASE	NOF LEARNTS INC	NOF LEARNTS DIVISOR	RANDOM VARFREQ	Time
1	0.937	1.000	1.828	289	1.878	0.100	0.001	22.6
2	0.974	0.842	2.000	625	2.000	10.000	0.003	22.8
3	0.968	1.000	1.862	355	1.274	7.239	0.011	22.9
4	0.915	1.000	2.000	497	2.000	9.086	0.000	23.3
5	0.930	0.939	1.868	931	2.000	0.100	0.000	23.5
6	0.968	0.924	1.888	505	1.707	10.000	0.000	24.2
7	0.914	1.000	2.000	10	2.000	1.815	0.000	24.7
8	0.862	1.000	1.317	707	1.176	0.100	0.000	25.0
9	0.918	1.000	1.481	994	1.139	7.426	0.000	25.7
10	0.934	1.000	1.499	878	2.000	10.000	0.019	26.8
Default	*0.95*	*0.999*	*1.5*	*100*	*1.1*	*3*	*0.02*	
Range	*0.01-1*	*0.01-1*	*1-2*	*10-1000*	*1-2*	*0.1-10*	*0-0.5*	
Min.	**0.862**	**0.842**	**1.317**	**10**	**1.139**	**0.100**	**0.000**	**22.6**
Max.	**0.974**	**1.000**	**2.000**	**994**	**2.000**	**10.000**	**0.019**	**26.8**
Med.	**0.932**	**1.000**	**1.865**	**565**	**1.939**	**7.332**	**0.000**	**23.9**
Avg.	***0.932***	***0.971***	***1.774***	***579***	***1.717***	***5.587***	***0.003***	***24.2***
Std. dev.	***0.033***	***0.053***	***0.249***	***311.574***	***0.373***	***4.489***	***0.006***	***1.4***
Std. dev. %	***3.6%***	***5.5%***	***14.0%***	***53.8%***	***21.7%***	***80.3%***	***192.4%***	***5.8%***
% of range	***11.2%***	***16.0%***	***68.3%***	***99.4%***	***86.1%***	***100.0%***	***3.7%***	

Table 6.7: Best results of 10 ES runs optimizing all seven solver parameters for training problem `manol-pipe-c6n`

6.6 SAT solver optimization experiments

6.6.1 Objective of the experiment

In the experiment described in this section MINISAT was optimized using a larger set of SAT problems and their shuffled versions. This experiment implements the "optimization on training sets" step of the SAT solver optimization procedure (see Sec. 1.8.2.1).

One of the results of the previous Sec. 6.5 was that the ES is generally faster and more precise when optimizing the SAT solver. The ES seems to reach a plateau for the average best objective value found after about 15 generations, while the GA takes longer. Since a SAT solver optimization run takes very long due to the evaluation times, it is even more important that the optimization algorithm converges quickly so that the maximum number of generations can be kept small. Therefore the ES is the more promising algorithm in a "production" setting where a solver is optimized for real-world applications, and will be exclusively used in this experiment.

6.6.2 Setup

For the ES all the standard settings were used as described in Sec. 6.2.3, except that the maximum number of generations was limited to 15 instead of 50; this means that in total $15 \times 50 = 750$ individuals were evaluated. The SAT problems to solve were drawn from the first qualification round of the 2006 SAT Race [Sinz06]; 27 different problems were used. The problems were chosen arbitrarily from the easier instances in the set; the relatively wide selection was intentional so that results could be analyzed for different types of problems. It was expected that the wide selection would also lead to a relatively "general purpose" configuration in the result candidate selection step of the optimization procedure (see Sec. 1.8.2.1), which would be compared to the default parameters which are also intended to be applicable in a wide range of problems.

In each optimization run the ES attempted to minimize the sum of solving times for one of the problems together with nine shuffled versions of that problem; this means that for every individual's evaluation 10 SAT problems were solved (so the SAT solver was started $750 \times 10 = 7500$ times). If any of the solving attempts resulted in a timeout, twice the timeout in seconds was added to the objective value. The timeouts were adjusted individually to each machine so that it was about twice the solving time for the original SAT problem using default parameters. In an industrial setting it would not be possible to predetermine timeouts in this manner of course, but it was done here to save time.

6.6.3 Results

Tab. 6.8 lists the best solutions found in each ES run for the respective problem, along with the default values, ES ranges, minimum, maximum, median, average, standard deviation (as value and in %) and the ratio of the full range covered by the solutions in each column. The problems are sorted in ascending order by the objective value of the best solution, normalized to the reference machine **`Pc1`**, so the table gives a rough estimate of how hard the problems are in relation to each other.

In the "result candidate selection" step of the SAT solver optimization procedure (see Sec.1.8.2.1) some algorithm has to look at the data gathered in the trial database and choose some configurations suitable as the final result. It may look at all trials for that purpose, not just the top results. As a simple example of such a wider choice for the candidate selection, the three best parameter configurations were extracted from each ES run. Tab. 6.8 does not list the second- and third-best results, but shows the statistics for this case.

To examine the results of the ES runs more closely, the three best solutions for each of the 27 problems were charted as points in XY-diagrams (81 points total) for the three parameter pairs and histograms were made to show in which ranges the points accumulate. The histograms partition the allowed range[2] of the parameter into 50 bins and sort all recorded solutions into these bins, so that the number of solutions that fall into a partial range can be counted; the sum over all bin counts is 81, the total number of solutions. Fig. 6.55 to Fig. 6.61 show the XY-diagrams and histograms, and give the statistical results of the relevant parameters taken from Tab. 6.8. Following observations can be made in the results:

1. $\mathrm{VARDECAY}$: Both the XY-diagram and the histogram show that optimal values for this parameter are almost always near or very near 1, as was already known from previous EA experiments as well as fitness landscapes. The histogram makes clear that $\mathrm{VARDECAY}$ is best chosen at values from 0.9 to 1 (but *not* exactly 1) regardless of the type of problem and if the problem is shuffled or not. Nevertheless, there are a handful of outliers outside of this range, including the lowest allowed value 0.01.

2. $\mathrm{CLAUSEDECAY}$: For this parameter the diagrams show that the optimal value is exactly 1 in the vast majority of cases (the median is 1, meaning that for more than half of the solutions $\mathrm{CLAUSEDECAY} = 1$). This result coincides with those from the previous experiments. There are outliers further away from 1 including the lowest possible value 0.01.

3. $\mathrm{NOFCONFLICTSINC}$: The median result for this parameter is 1, which is very different from the results of the previous experiments where no order could be detected for what value is optimal. The second noticeable peak in

[2]Where the lower limit was very near to 0, exactly 0 was used as the lowest bin

Problem	VARDE CAY	CLAUSE DECAY	NOFCO NFLICTS INC	NOFCON FLICTSB ASE	NOFLEA RNTSINC	NOFLEA RNTSDIVI SOR	RANDOM VARFRE Q
manol-pipe-f6b	0.917	1.000	1.000	91	1.729	7.258	0.000
manol-pipe-f6n	0.958	1.000	1.353	769	1.898	10.000	0.000
manol-pipe-g7n	0.881	1.000	1.000	113	1.998	0.100	0.004
manol-pipe-g6bid	0.962	1.000	1.034	368	1.738	0.100	0.000
vange-color-inc-54	0.910	1.000	2.000	691	1.704	2.728	0.004
velev-eng-uns-1.0-04	0.906	1.000	1.758	699	1.000	0.208	0.000
manol-pipe-c7_i	0.938	1.000	1.000	696	1.046	8.724	0.079
manol-pipe-c8_i	0.945	1.000	1.000	407	1.050	3.415	0.339
grieu-vmpc-s05-24s	0.999	0.010	1.174	565	1.000	8.898	0.009
velev-live-sat-1.0-01	0.852	0.010	1.000	485	1.000	0.100	0.000
velev-npe-1.0-02	0.946	1.000	1.015	55	1.000	0.899	0.000
manol-pipe-c10ni_s	0.971	0.058	1.000	450	1.000	3.756	0.000
manol-pipe-c6nid_s	1.000	1.000	1.000	357	1.844	0.100	0.000
velev-eng-uns-1.0-04a	0.954	1.000	2.000	892	1.309	10.000	0.000
velev-npe-1.0-03	0.795	1.000	1.000	471	2.000	4.357	0.288
goldb-heqc-rotmul	0.910	1.000	1.980	211	1.352	10.000	0.000
manol-pipe-c6n	0.934	1.000	1.350	133	1.091	10.000	0.000
schup-l2s-s04-abp4	0.488	1.000	1.000	764	1.010	10.000	0.000
velev-pipe-1.1-05	0.900	1.000	1.000	451	1.483	6.432	0.000
goldb-heqc-desmul	0.978	1.000	1.167	217	1.000	0.152	0.000
hoons-vbmc-s04-07	0.991	0.213	1.331	440	1.617	0.100	0.000
velev-fvp-sat-3.0-12	0.989	1.000	1.194	328	1.000	2.072	0.000
manol-pipe-c8b_i	0.943	1.000	1.000	657	1.952	7.671	0.000
manol-pipe-c6id	0.947	1.000	1.000	848	2.000	9.595	0.000
manol-pipe-c8n	0.964	1.000	1.000	716	2.000	6.270	0.000
manol-pipe-c7idw	0.916	1.000	1.001	874	2.000	4.478	0.000
manol-pipe-g10idw	0.970	0.428	1.000	491	1.466	0.529	0.000
Default	*0.95*	*0.999*	*1.5*	*100*	*1.1*	*3*	*0.02*
Range	*0.01-1*	*0.01-1*	*1-2*	*10-1000*	*1-2*	*0.1-10*	*0-0.5*
Min.	0.488	0.010	1.000	55	1.000	0.100	0.000
Max.	1.000	1.000	2.000	892	2.000	10.000	0.339
Med.	**0.945**	**1.000**	**1.000**	**471**	**1.466**	**4.357**	**0.000**
Avg.	**0.921**	**0.841**	**1.198**	**490**	**1.455**	**4.739**	**0.027**
Std. dev.	*0.098*	*0.346*	*0.335*	*251*	*0.414*	*3.976*	*0.084*
Std. dev. %	*10.6%*	*41.1%*	*28.0%*	*51.3%*	*28.5%*	*83.9%*	*314.8%*
% of range	*51.7%*	*100.0%*	*100.0%*	*84.6%*	*100.0%*	*100.0%*	*67.8%*

Results using the 3 best solutions of each run:

Min.	0.010	0.010	1.000	48	1.000	0.100	0.000
Max.	1.000	1.000	2.000	994	2.000	10.000	0.347
Med.	**0.939**	**1.000**	**1.000**	**451**	**1.483**	**4.433**	**0.000**
Avg.	***0.909***	***0.891***	***1.230***	***460***	***1.476***	***4.755***	***0.020***
Std. dev.	*0.134*	*0.247*	*0.365*	*246*	*0.392*	*3.922*	*0.065*
Std. dev. %	*14.8%*	*27.7%*	*29.7%*	*53.4%*	*26.6%*	*82.5%*	*325.2%*
% of range	*100.0%*	*100.0%*	*100.0%*	*95.6%*	*100.0%*	*100.0%*	*69.5%*

Table 6.8: Best solutions found when optimizing 7 heuristic parameters in MINISAT with an ES using 27 problems from the 2006 SAT Race (below are statistics for 3 best solutions of each run)

the histogram is at 2, the other extreme value, but this value is optimal for a far smaller number of cases. Many values inbetween the two extremes also were optimal sometimes, but only for 1 or 2 cases each.

4. NOFCONFLICTSBASE: Optimal solutions for this variable are widely distributed over the whole range. There are 3 big peaks centered around 120, 420 and 720, but it is not clear if this is simply an artifact of randomness. No value seems particularly better than another, since even the peaks are only 5 occurrences each.

5. NOFLEARNTSINC: The points in the XY-diagram for this parameter are widely distributed, but it can be seen in the histogram that approximately a quarter of the values congregate at the extreme ends of the range (respectively 19 and 13 of 81 values), with the rest evenly distributed inbetween. It is unknown if the massing of points near the extremes is due to a real advantage of those values or if this is simply an artifact of the ES getting stuck near the limits in the absence of clear selective pressure.

6. NOFLEARNTSDIVISOR: The histogram for this parameter looks similar to the one for the previous parameter. The largest peaks are found at the extreme ends of the range, which contain respectively 13 and 15 of the 81 points.

7. RANDOMVARFREQ: This parameter is optimal at exactly 0 for the vast majority of cases, which was also indicated by the previous experiments.

To summarize, optimal values for the parameters according to the ES optimization experiments were most often:

VARDECAY	≈ 1
CLAUSEDECAY	$= 1$
NOFCONFLICTSINC	$= 1$
NOFCONFLICTSBASE	no recognizable pattern
NOFLEARNTSINC	no recognizable pattern
NOFLEARNTSDIVISOR	no recognizable pattern
RANDOMVARFREQ	$= 0$

An investigation of the correlation between parameters that were not paired and their results depicted in the XY-diagrams was not performed in this work. This is an area that requires further study.

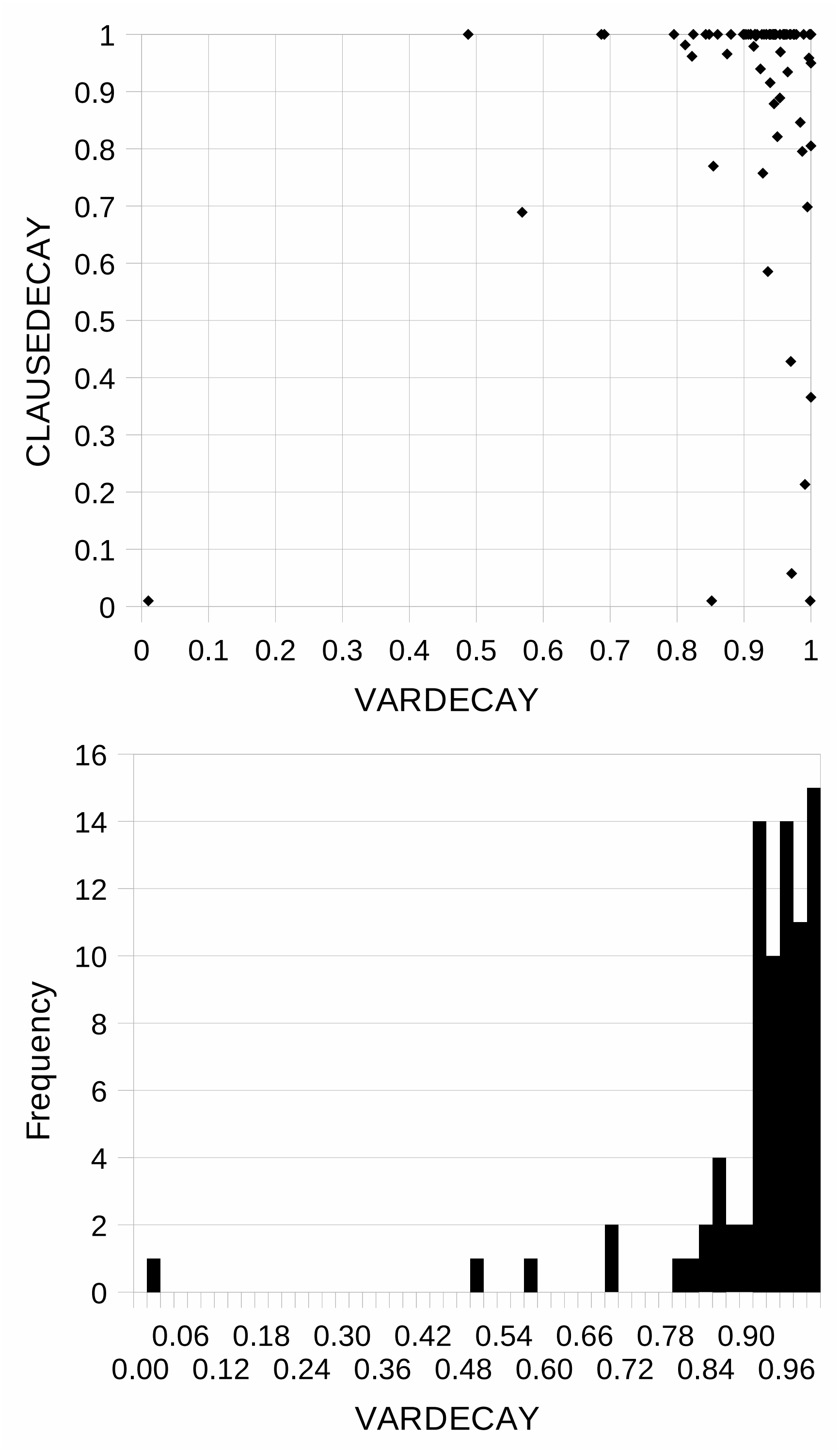
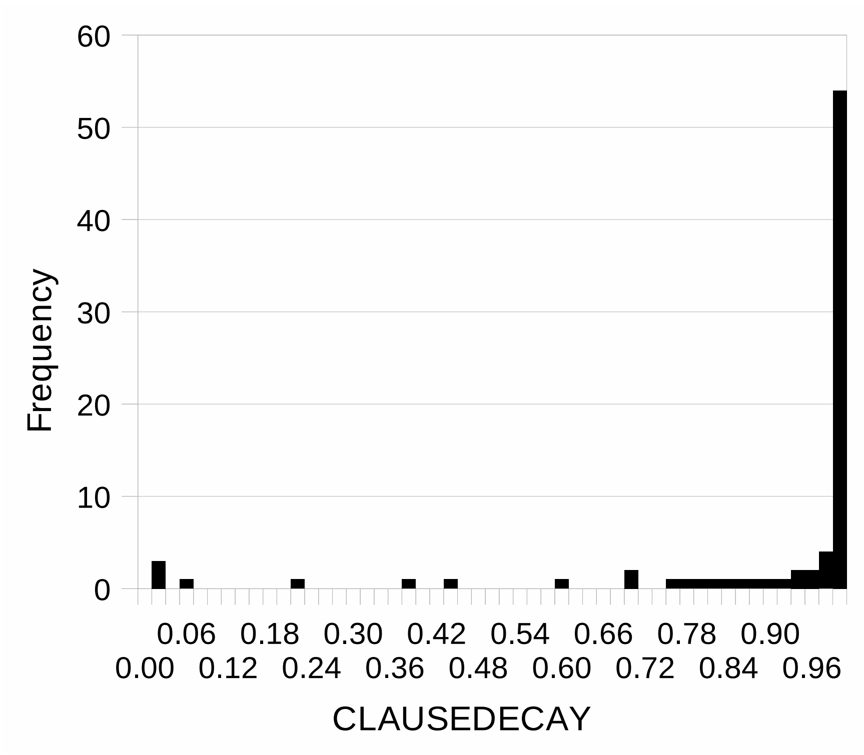


Figure 6.55: All best 3 points in ES optimization results: XY-diagram for VARDECAY / CLAUSEDECAY and histogram for VARDECAY



	VAR DECAY	CLAUSE DECAY
Default	*0.95*	*0.999*
Range	*0.01-1*	*0.01-1*
Min.	0.010	0.010
Max.	1.000	1.000
Med.	0.939	1.000
Avg.	*0.909*	*0.891*
Std. dev.	*0.134*	*0.247*
Std. dev. %	*14.8%*	*27.7%*
% of range	*100.0%*	*100.0%*

Figure 6.56: All best 3 points in ES optimization results: histogram for CLAUSEDECAY and statistics for VARDECAY / CLAUSEDECAY

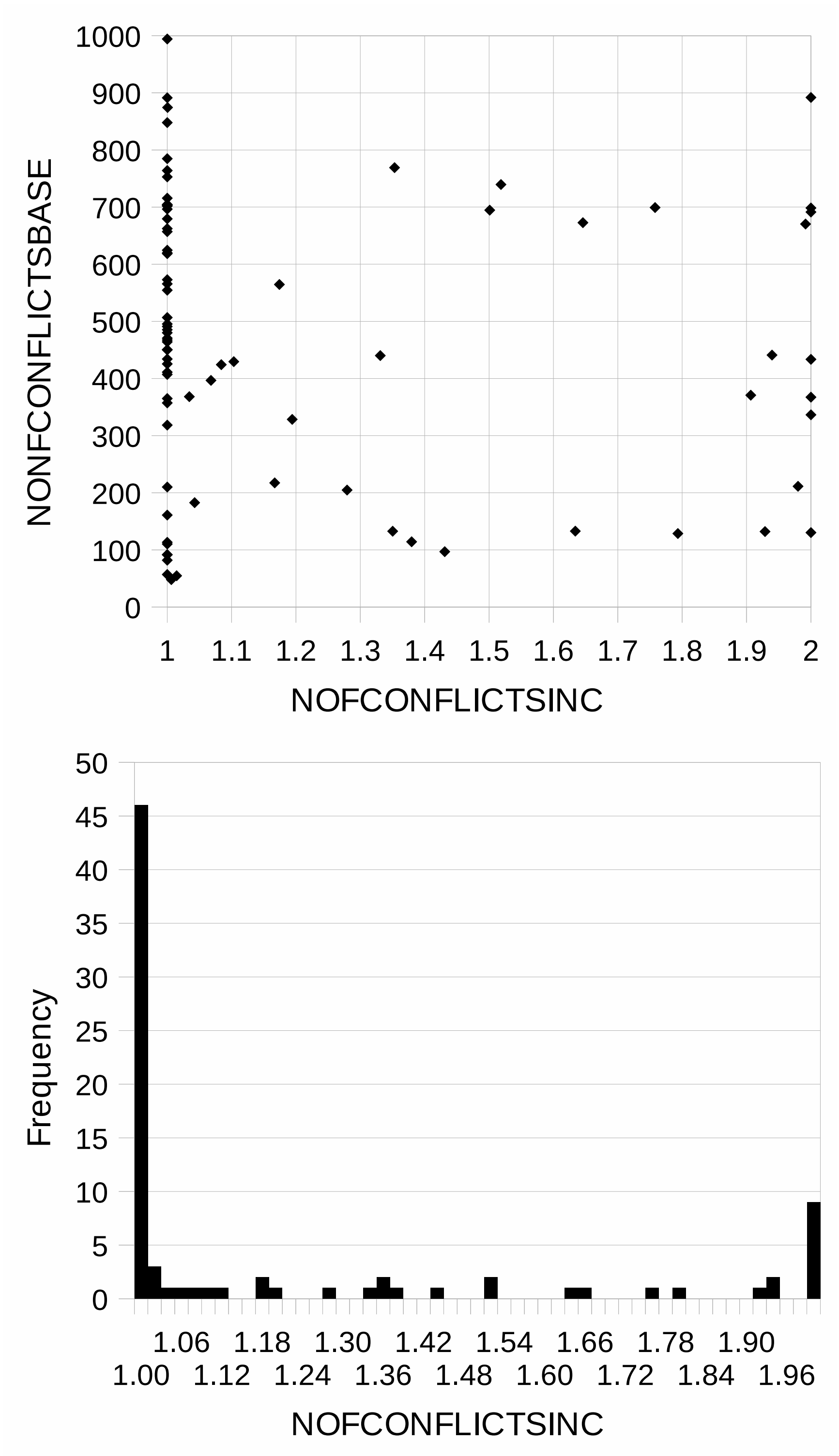
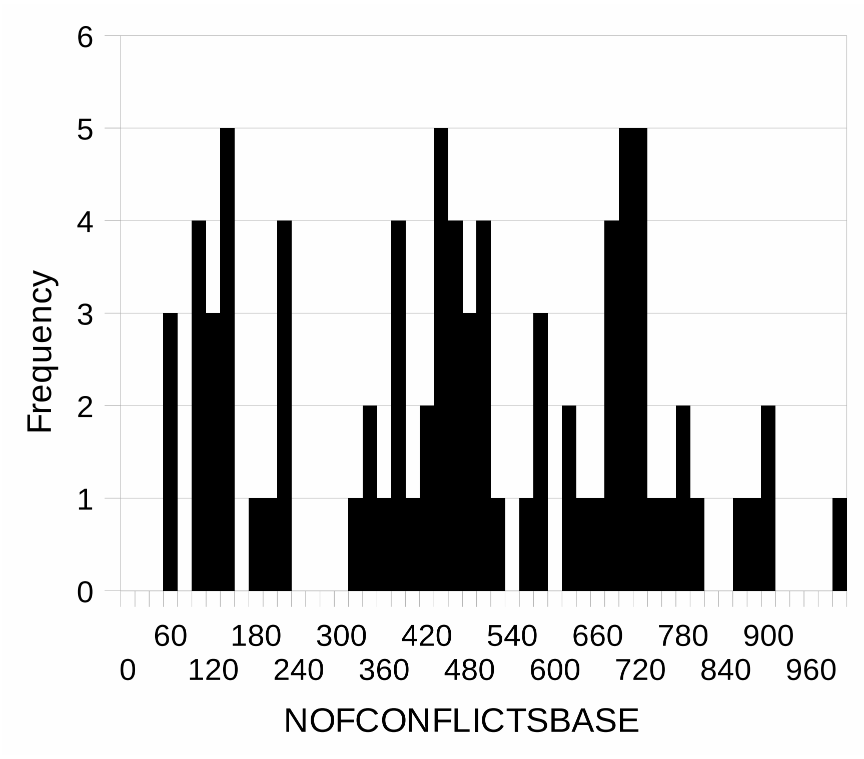
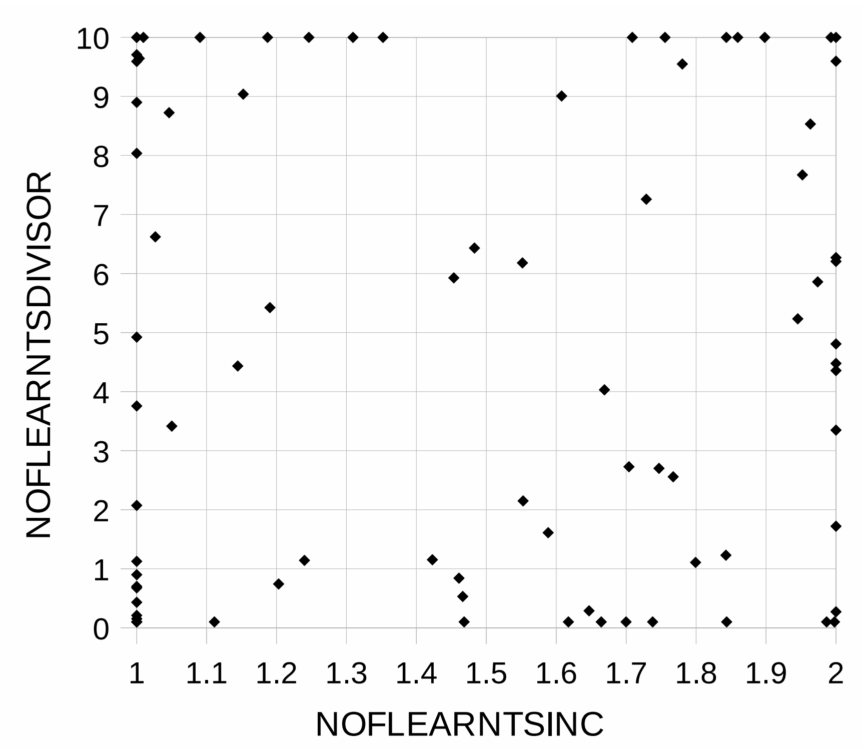


Figure 6.57: All best 3 points in ES optimization results: XY-diagram for NOFCONFLICTSINC / NOFCONFLICTSBASE and histogram for NOFCONFLICTSINC



	NOF CONFLICTS INC	NOF CONFLICTS BASE
Default	*1.5*	*100*
Range	*1-2*	*10-1000*
Min.	1.000	48
Max.	2.000	994
Med.	1.000	451
Avg.	*1.230*	*460*
Std. dev.	*0.365*	*246*
Std. dev. %	*29.7%*	*53.4%*
% of range	*100.0%*	*95.6%*

Figure 6.58: All best 3 points in ES optimization results: histogram for NOFCONFLICTSBASE and statistics for NOFCONFLICTSINC / NOFCONFLICTSBASE



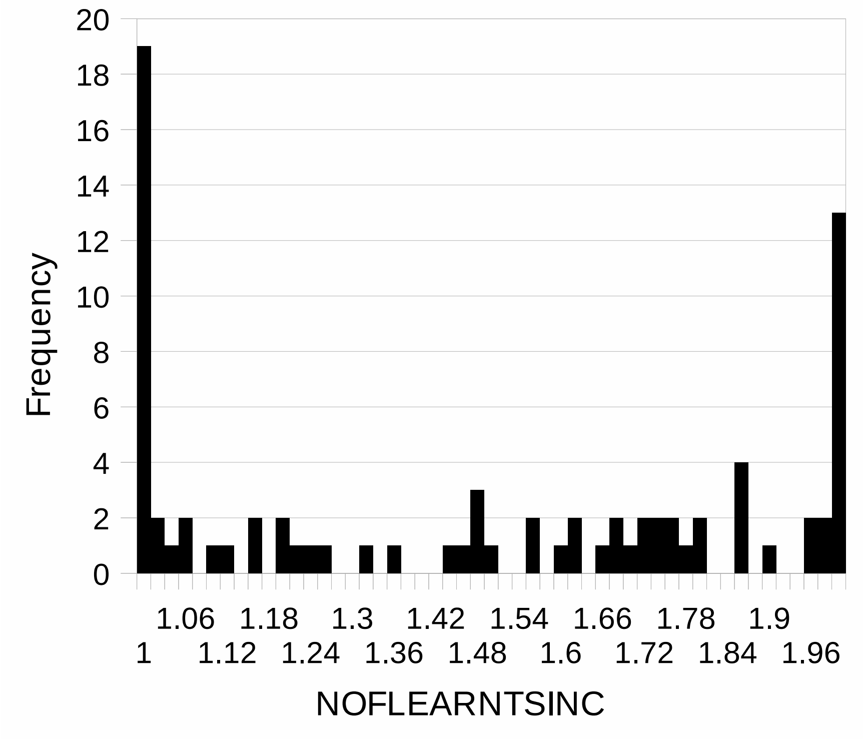
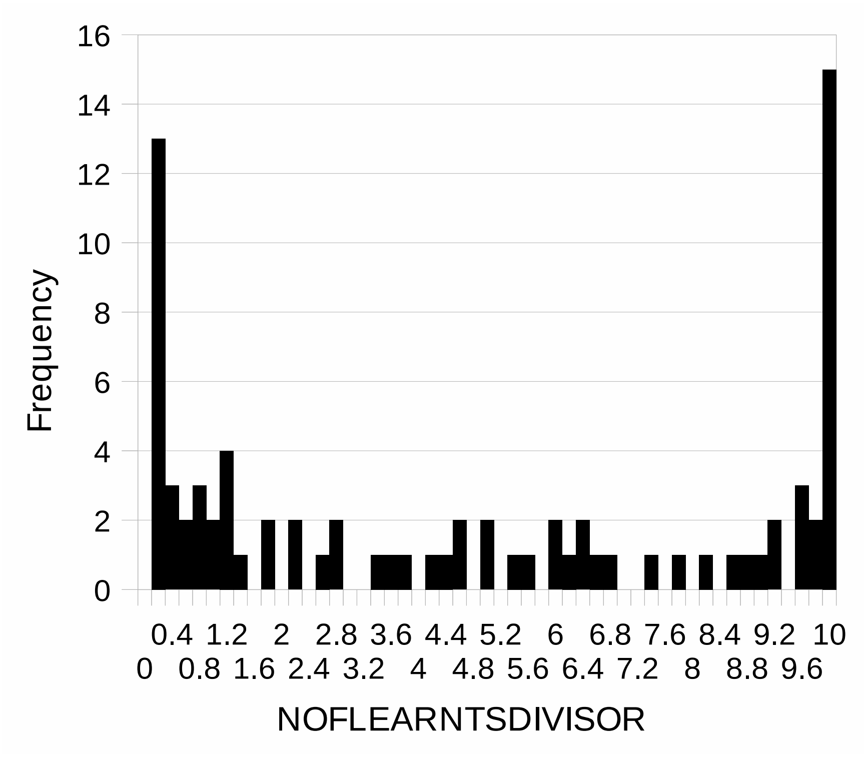
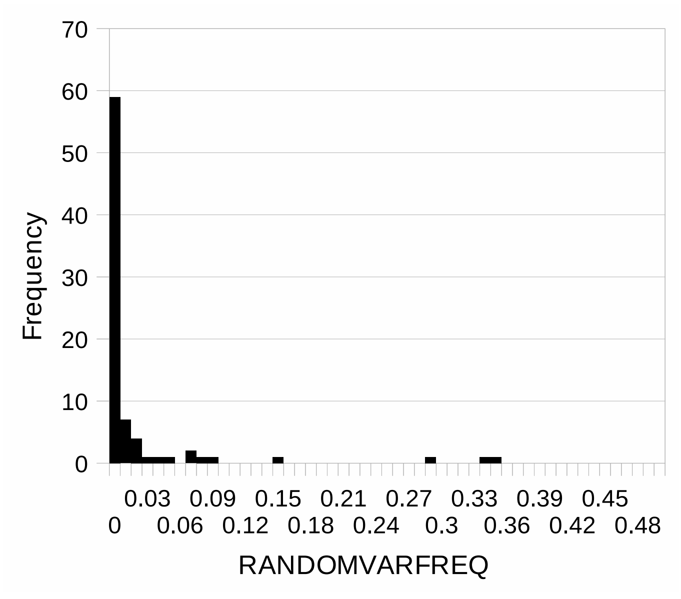


Figure 6.59: All best 3 points in ES optimization results: XY-diagram for NOFLEARNTSINC / NOFLEARNTSDIVISOR and histogram for NOFLEARNTSINC



	NOF LEARNTS INC	NOF LEARNTS DIVISOR
Default	*1.1*	*3*
Range	*1-2*	*0.1-10*
Min.	1.000	0.100
Max.	2.000	10.000
Med.	1.483	4.433
Avg.	*1.476*	*4.755*
Std. dev.	*0.392*	*3.922*
Std. dev. %	*26.6%*	*82.5%*
% of range	*100.0%*	*100.0%*

Figure 6.60: All best 3 points in ES optimization results: histogram for NOFLEARNTSDIVISOR and statistics for NOFLEARNTSINC / NOFLEARNTSDIVISOR



	RANDOM VARFREQ
Default	*0.02*
Range	*0-0.5*
Min.	**0.000**
Max.	**0.347**
Med.	**0.000**
Avg.	***0.020***
Std. dev.	***0.065***
Std. dev. %	***325.2%***
% of range	***69.5%***

Figure 6.61: All best 3 points in ES optimization results: histogram and statistics for RANDOMVARFREQ

6.6.4 Visualization of the search

To investigate the behavior of the ES in the search space of the SAT solver optimization problem some diagrams were generated that display all the tested points. Problem `manol-pipe-c7idw`, together with nine shuffled versions of itself, was the training set, and the run was executed on the machine **`Pc3`**. The ES was applied on this set for 15 generations, which yields a total of 750 tested points. Fig. 6.62 and Fig. 6.63 show the resulting diagrams. The usual pairs of parameters were the X- and Y-axis of three-dimensional diagrams, and the objective values are the Z-axis (RANDOMVARFREQ is in an additional two-dimensional diagram). Each cross marks a point tested by the ES, and its height (and shade of gray) marks its objective value. These diagrams are essentially "incomplete fitness landscapes" since they only show the portion of the landscape actually explored by the ES.

It can be observed that in the VARDECAY / CLAUSEDECAY diagram the points mass in the vicinity of the known global optimum. The diagram for RANDOMVARFREQ shows that the best results all share a very low value for this parameter.

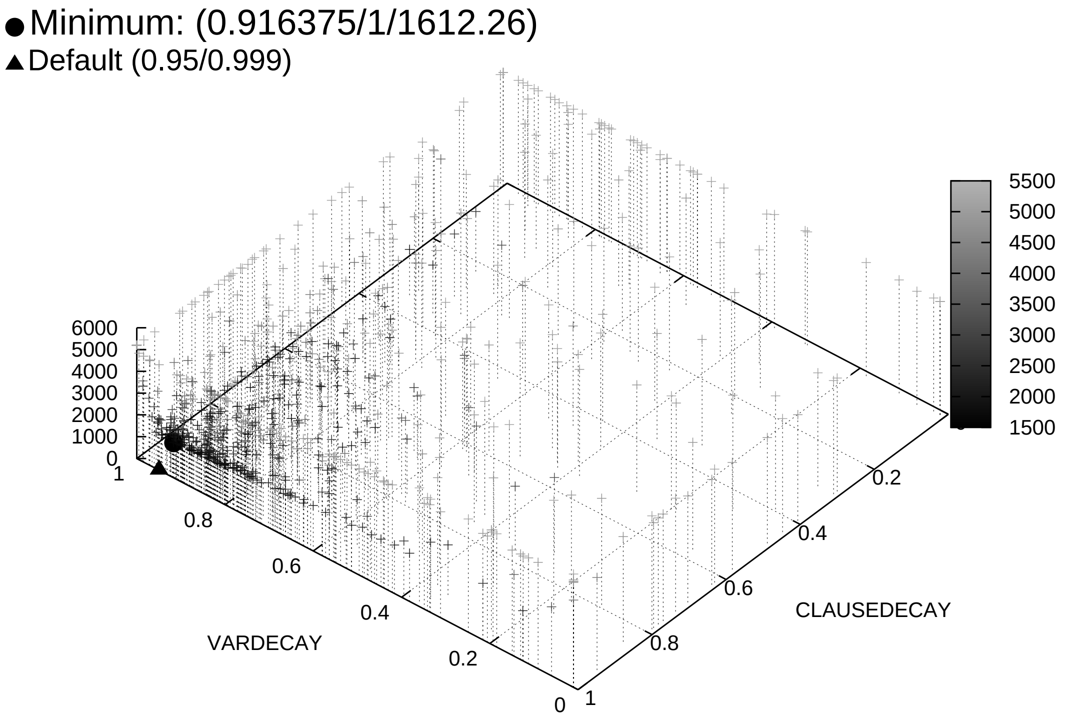
The diagram for NOFCONFLICTSINC / NOFCONFLICTSBASE shows two long masses of points ("stripes") that extend along the entire length of the NOFCONFLICTSINC-axis. The masses contain many optimal points, but also many timed-out points. It was found after some investigation that the particular values indicated on the NOFCONFLICTSBASE axis where the masses are located are not necessarily optimal, but rather that this phenomenon stems from a quirk of the ES algorithm used in this work. The ranges of the two parameters are very different, with NOFCONFLICTSINC having a range of only 1 and NOFCONFLICTSBASE a range of nearly 1000. The initial value of the standard deviation σ for the ES mutation operator is set to only 3 (see Sec. 6.2.3) for the parameter NOFCONFLICTSBASE, which is much less than the total range, meaning that every generational step only moves the children a small distance from their parents compared to the full range. On the other hand, the range of the parameter NOFCONFLICTSINC which is much smaller is fully explored. The σ values for NOFCONFLICTSBASE do not seem to grow much over the 15 generations, possibly because the fitness landscape of these parameters is so random. Indeed, using the ES on a completely random test function with variables with the same ranges revealed the same phenomenon of points congregating in "stripes".

The diagram for parameters NOFLEARNTSINC and NOFLEARNTSDIVISOR shows that the entire range of both parameters is explored by the ES, and that there are many semi-optimal points all over the search space with no visible order. This is in line with the findings of the fitness landscape experiments.

6.6.5 Conclusions

Optimization experiments were performed on MiniSAT using the ES as the optimization algorithm, with a shortened number of 15 generations maximum (instead of 50) which appeared to be adequate judging from the results of the previous section. The training sets contained one of 27 different SAT problems and additionally 9 shuffled version of that problem. Results showed that in the best configurations 4 of the 7 parameters converged to always the same values, while the remaining three were apparently random. Of the converging parameters, 3 were known to have relatively narrow optimal ranges from the fitness landscape experiments, while the parameter NOFCONFLICTSINC which seemed to have a completely random influence in the fitness landscapes now nearly always converged to the value 1. It is unknown why this happens, though it may have do with using many shuffled problems in the training set; this line of investigaton was not further followed.

A visualization of the search space explored by the ES was created for one of the experiments. The diagrams showed that VARDECAY, CLAUSEDECAY and RANDOMVARFREQ have easy to find global optima. NOFCONFLICTSINC is explored in the entire range, but NOFCONFLICTSBASE is apparently only explored in relatively narrow "stripes" due to the σ values of the ES being and remaining much smaller than the full range of the parameter. Parameters NOFLEARNTSINC and NOFLEARNTSDIVISOR are fully explored and have many optimal points across their entire ranges.



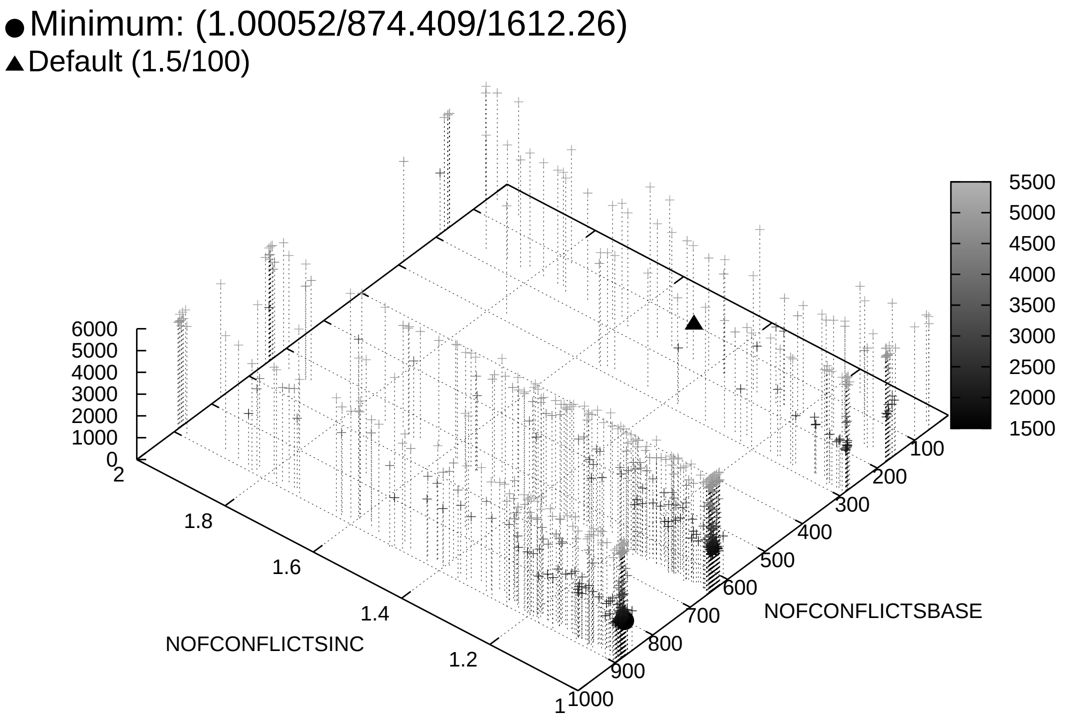


Figure 6.62: Visualization of the search space (first 4 parameters) after applying an ES on problem `manol-pipe-c7idw` together with nine shuffled versions

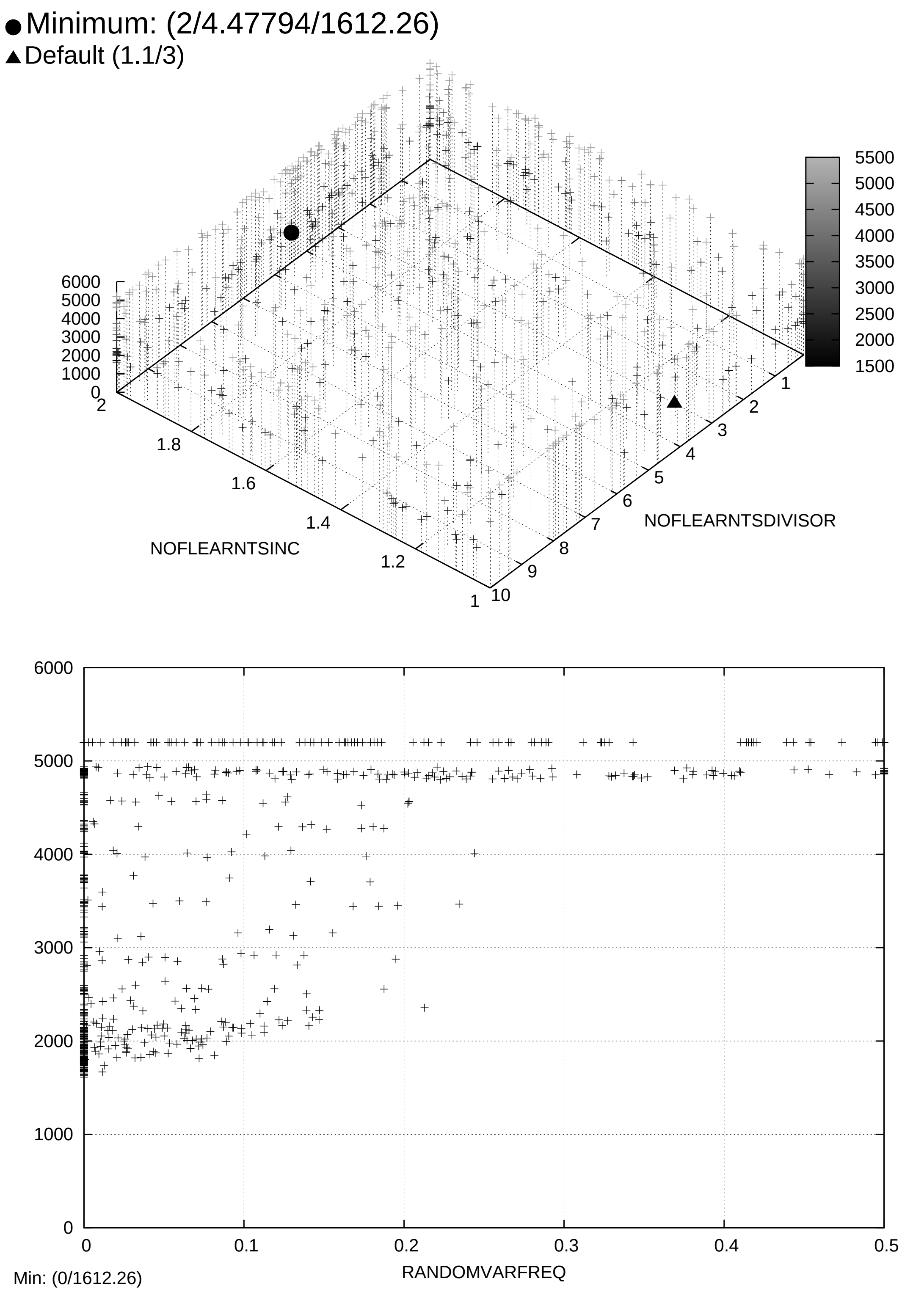


Figure 6.63: Visualization of the search space (last 3 parameters) after applying an ES on problem `manol-pipe-c7idw` together with nine shuffled versions

	VAR DECAY	CLAUSE DECAY	NOF CONFLICTS INC	NOF CONFLICTS BASE	NOF LEARNTS INC	NOF LEARNTS DIVISOR	RANDOM VARFREQ
Default	*0.95*	*0.999*	*1.5*	*100*	*1.1*	*3*	*0.02*
Results using the best solution of each run:							
(1med) Med.	**0.945**	**1.000**	**1.000**	**471**	**1.466**	**4.357**	**0.000**
(1avg) Avg.	**0.921**	**0.841**	**1.198**	**490**	**1.455**	**4.739**	**0.027**
Results using the 3 best solutions of each run:							
(3med) Med.	**0.939**	**1.000**	**1.000**	**451**	**1.483**	**4.433**	**0.000**
(3avg) Avg.	**0.909**	**0.891**	**1.230**	**460**	**1.476**	**4.755**	**0.020**

Table 6.9: Average and median of all parameters from the results of the ES optimization experiment

6.7 Result candidate selection and testing

In the final step of the SAT solver optimization procedure (Sec. 1.8.2.1) candidates for a “final” solver parameter configuration have to be chosen from the data gathered in the optimization step. These should be sets of parameter values that solve as many different problems as possible as fast as possible. Furthermore, this step has to be performed by an algorithm, not a human programmer who can analyze the large amount of data in detail. While sophisticated data mining algorithms may be the most promising algorithms for this task, this would go beyond the scope of this work. Instead, only some very basic algorithms will be tested.

6.7.1 Candidate selection

It was found from the results of the ES experiments presented in Sec. 6.6.3 that for the majority of problems following four parameter values were ideal:

$$\begin{aligned} \mathrm{VARDECAY} &\approx 1 \\ \mathrm{CLAUSEDECAY} &= 1 \\ \mathrm{NOFCONFLICTSINC} &= 1 \\ \mathrm{RANDOMVARFREQ} &= 0 \end{aligned}$$

Of these parameters, $\mathrm{CLAUSEDECAY}$, $\mathrm{NOFCONFLICTSINC}$ and $\mathrm{RANDOMVARFREQ}$ even had a median value of 1 or 0, but there were always some cases where a slightly different value was optimal. For the remaining three parameters $\mathrm{NOFCONFLICTSBASE}$, $\mathrm{NOFLEARNTSINC}$ and $\mathrm{NOFLEARNTSDIVISOR}$ no obvious choices were found; the results had values for these parameters from all over the allowed range.

A simple algorithm to choose a set of parameter values that conceivably might compromise between the many different solutions to create one that works acceptably well on many or even all of them is to choose the average or the median of the parameter results. Tab. 6.9 shows the averages and medians for all parameter values when using either only the best results for all problems or the three best results; the

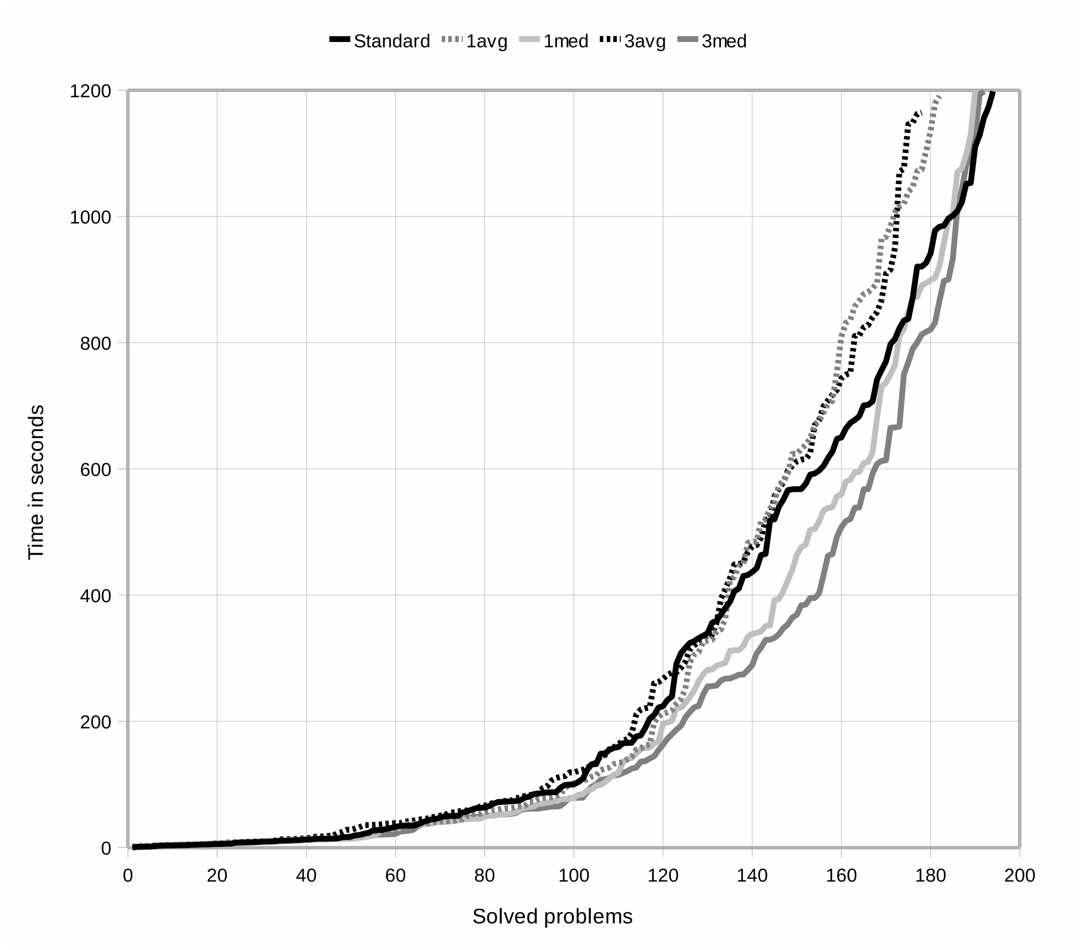


Figure 6.64: Snake plot for the candidate testing results on the test pool of 300 problems

four configurations have been designated ***1med***, ***1avg***, ***3med*** and ***3avg***; for comparison the table also shows the standard MINISAT parameters.

6.7.2 Candidate testing

While the differences between these configurations may be small, it was decided to determine how much of a difference such variations have when dealing with a large benchmark set. The *test pool* $\mathcal{P}_{\text{test}}$ (see Sec. 1.8.2.1) consisted of the full set of benchmarks of the final round of the 2006 SAT Race (which contains a total of 100 SAT problems) including two shuffled versions of each problem, bringing the total to 300 problems. All four of the computed configurations, along with the standard MiniSAT parameter set, were then used to solve the test pool problems with a timeout of 20 minutes. All times were taken on the machine **`Pc1`**.

The solving times of all configurations for the original, non-shuffled problems are listed in Tab. 6.10 and Tab. 6.11. Additionally, Fig. 6.64 shows a so-called *snake plot* that charts how many of the 300 problems each configurations solves (charted on the X-axis) if the timeout had been some amount of time (charted on the Y-axis). For example it can be seen that given a timeout of 600 seconds (10 minutes) the configurations ***1avg*** and ***3avg*** would have solved about 150 problems, but configuration ***3med*** would have solved nearly 170 problems. In the snake plot the fastest configurations have the lowermost line, while the configurations that ultimately solve the most problems end farthest to the right. For example it can be seen that the standard parameter configuration solves the most problems (194 total) after the maximum timeout of 1200 seconds (20 minutes) has elapsed, followed by configuration ***3med*** (192 total).

Tab. 6.10 and Tab. 6.11 show that both the average-derived configurations ***1avg*** and ***3avg*** solved the least amount of problems (67 and 66), but the median-derived result ***1med*** solved 71 problems and ***3med*** was equal to the standard parameter configuration with 72 solved problems each. Interestingly, ***1med*** and ***3med*** do not solve some problems that the standard parameters could, but on the other hand both did very well on the `velev` class of benchmarks: both successfully solved 17 of 20 problems and the default configuration only managed 9. On the other hand, both the median-derived configurations did noticeably badly on the `grieu` and `mizh` classes; it is unknown what causes this. The average-derived configurations seem to have no particular strengths.

The snake plot in Fig. 6.64 shows that both the average-derived configurations ***1avg*** and ***3avg*** are the slowest of the five tested configurations, since their lines are generally the highest and end farthest to the left. Interestingly, the median-derived configurations ***1med*** and ***3med*** both run below the line of the standard parameter configuration up to a time of about 1000 seconds (meaning they solve more problems in the same amount of time than the standard parameters), but then are overtaken by the standard parameters. Configuration ***3med*** is also noticeably better than ***1med***.

The simplest scoring scheme (used in the last step of the optimization procedure, see Sec. 1.8.2.1) that determines which configuration is the final result is counting the number of solved instances. More complicated scoring schemes might also take into account how quickly the problems were solved and other details; this was not further investigated in this work. Using an automatic procedure to choose the final result that only takes the total number of solved problems into account would result in ***3med*** being chosen as the winner, considering either only the original problems or original and shuffled problems together as the test pool. Fig. 6.65 shows a comparison of ***3med*** with the standard parameter configuration. Taken together, both configurations solve 217 of the 300 test pool problems. In 25.3% of these cases ***3med*** is either the only configuration that solves the problem or it is at least 10% of the timeout (2 minutes) faster than the standard configuration. In 55.8% of the 217 problems both ***3med*** and the standard configuration solve the problem with a solving time that differs by at most 10% of the timeout. For the remaining 18.9% the standard

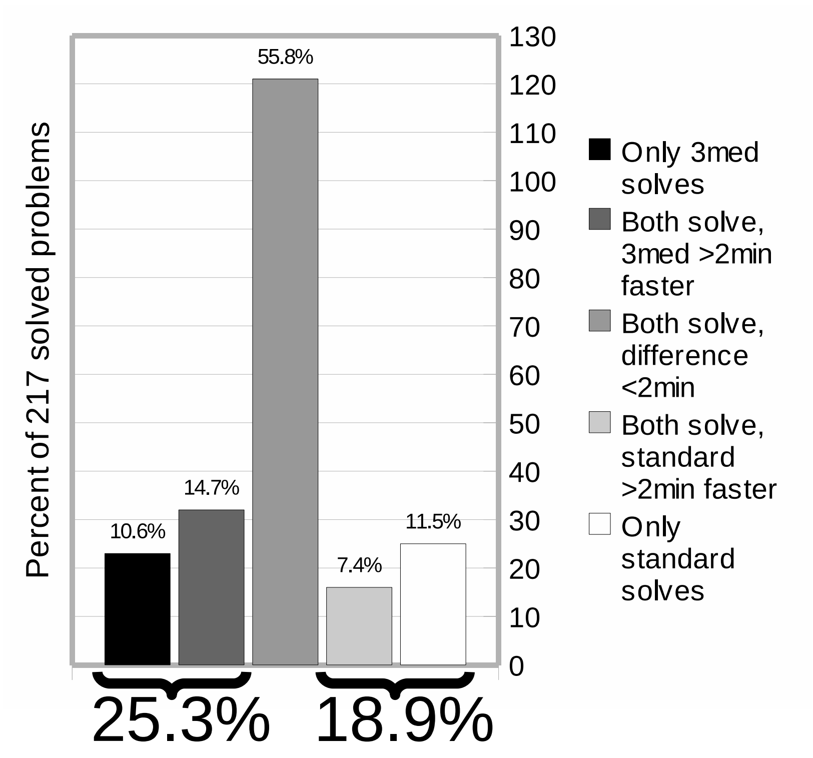


Figure 6.65: Problems solved using either the standard parameters or configuration ***3med*** out of the 300 test pool problems

parameter configuration solves the problems either more than 10% of the timeout faster than ***3med*** or is the only coniguration that solves them. In summary, for the chosen test pool configuration ***3med*** has a slight edge in solving speed compared to the standard parameter configuration, but ultimately solves slightly less problems in total.

6.7.3 Conclusions

Using the data gathered by the optimization on training sets, four different candidate configurations were generated: two configurations are made of the averages of all parameters of the best configurations or respectively the three best configurations of all runs, and the other two using the same source configurations but by computing the medians. Solving a test pool of 300 SAT problems (100 original plus two shuffled versions of each problem) showed that the median-derived configurations were roughly equal in quality to the default parameters, though they did better at some benchmark classes and worse at others compared to the default parameters. The average-derived configurations were noticeably worse.

The result generated from computing the median of the best three results of each optimization run was found to be the fastest overall. It had a slight edge in solving speed compared to the standard MINISAT parameters, but ultimately solved a few problems less after the maximum timeout period than the standard parameters.

#	Problem	Sat.	default	*1avg*	*1med*	*3avg*	*3med*
1	aloul-chnl11-13	UNSAT	*1156.9*	---	---	---	---
2	een-pico-prop01-75	UNSAT	*4.0*	4.3	4.7	4.9	4.8
3	een-pico-prop05-50	UNSAT	*56.7*	36.7	45.7	38.7	41.9
4	een-tip-sat-nusmv-t5.B	SAT	*8.0*	7.6	7.86	8.3	7.4
5	een-tip-sat-nusmv-tt5.B	SAT	*8.5*	8.3	7.75	8.2	7.7
6	een-tip-uns-nusmv-t5.B	UNSAT	*2.1*	1.7	2.26	2.2	1.6
7	goldb-heqc-alu4mul	UNSAT	*430.1*	708.3	750.03	568.6	607.9
8	goldb-heqc-dalumul	UNSAT	---	---	---	---	---
9	goldb-heqc-desmul	UNSAT	*50.2*	53.2	42.91	50.4	42.9
10	goldb-heqc-frg2mul	UNSAT	*233.8*	327.7	238.28	222.0	329.3
11	goldb-heqc-i10mul	UNSAT	---	---	---	---	---
12	goldb-heqc-i8mul	UNSAT	*649.8*	547.5	595.3	717.9	612.7
13	goldb-heqc-term1mul	UNSAT	---	---	---	---	---
14	grieu-vmpc-s05-25	SAT	*359.3*	---	---	159.1	---
15	grieu-vmpc-s05-27	SAT	*683.2*	---	---	---	---
16	grieu-vmpc-s05-28	SAT	---	---	---	---	---
17	grieu-vmpc-s05-34	SAT	---	---	---	---	---
18	hoons-vbmc-lucky7	UNSAT	*756.5*	13.0	53	42.8	20.3
19	ibm-2002-05r-k90	SAT	*34.5*	79.5	36.14	60.6	51.0
20	ibm-2002-07r-k100	UNSAT	*5.0*	5.7	11.89	3.4	4.7
21	ibm-2002-11r1-k45	SAT	*239.3*	228.2	221.72	337.3	154.5
22	ibm-2002-19r-k100	SAT	---	---	---	---	---
23	ibm-2002-21r-k95	SAT	*926.3*	---	1070.37	---	---
24	ibm-2002-26r-k45	UNSAT	*3.1*	0.9	0.56	4.0	1.5
25	ibm-2002-27r-k95	SAT	*49.3*	41.9	40	38.6	36.7
26	ibm-2004-03-k70	SAT	*39.9*	38.2	27.27	44.2	43.1
27	ibm-2004-04-k100	SAT	*835.1*	880.0	342.52	950.1	368.7
28	ibm-2004-06-k90	SAT	*132.1*	62.7	78.59	169.2	136.1
29	ibm-2004-1_11-k25	UNSAT	*5.4*	6.5	7.02	4.7	4.8
30	ibm-2004-1_31_2-k25	UNSAT	*99.2*	134.9	164.87	83.5	110.3
31	ibm-2004-19-k90	SAT	*977.7*	1074.5	1199.37	578.8	1193.3
32	ibm-2004-2_02_1-k100	UNSAT	*10.8*	10.6	8.98	13.2	9.3
33	ibm-2004-2_14-k45	UNSAT	*23.4*	23.8	13.1	18.5	21.5
34	ibm-2004-26-k25	UNSAT	*1.1*	53.3	0.18	4.9	0.2
35	ibm-2004-3_02_1-k95	UNSAT	*1.2*	1.8	2.98	1.1	0.4
36	ibm-2004-3_02_3-k95	SAT	*0.2*	15.5	3.72	5.8	0.8
37	ibm-2004-3_11-k60	UNSAT	---	---	---	---	---
38	ibm-2004-6_02_3-k100	UNSAT	*4.3*	3.8	3.89	3.3	5.6
39	manol-pipe-c10id_s	UNSAT	*8.2*	31.5	7.19	35.5	4.5
40	manol-pipe-c10nidw_s	UNSAT	*701.4*	293.3	157.84	---	136.9
41	manol-pipe-c6nidw_i	UNSAT	*590.9*	623.5	351.02	556.7	395.3
42	manol-pipe-c7b	UNSAT	*45.2*	57.0	40.42	44.9	47.8
43	manol-pipe-c7b_i	UNSAT	*63.0*	44.5	43.92	45.7	41.2
44	manol-pipe-c7bidw_i	UNSAT	---	---	---	---	---
45	manol-pipe-c7nidw	UNSAT	---	---	---	---	---
46	manol-pipe-c9	UNSAT	*13.7*	13.1	11.33	9.7	5.7
47	manol-pipe-c9nidw_s	UNSAT	*339.4*	484.7	62.93	468.1	141.1
48	manol-pipe-f10ni	UNSAT	---	---	---	---	---
49	manol-pipe-f6bi	UNSAT	*9.7*	8.6	6.03	5.3	4.2
50	manol-pipe-f7idw	UNSAT	*770.1*	638.0	890.5	1164.6	567.7

Table 6.10: Comparison of solving times for the SAT Race 2006 benchmarks using different parameter configurations (pt. 1)

#	Problem	Sat.	default	*1avg*	*1med*	*3avg*	*3med*
51	manol-pipe-f9b	UNSAT	---	---	---	---	---
52	manol-pipe-f9n	UNSAT	---	---	---	---	---
53	manol-pipe-g10b	UNSAT	*86.9*	125.2	84.15	146.7	62.8
54	manol-pipe-g10bidw	UNSAT	---	---	334.08	---	402.6
55	manol-pipe-g10id	UNSAT	*94.7*	62.4	76.56	79.3	63.7
56	manol-pipe-g10nid	UNSAT	*1129.0*	---	281.57	---	273.9
57	manol-pipe-g6bi	UNSAT	*1.3*	2.2	1.36	1.6	1.5
58	manol-pipe-g7nidw	UNSAT	*34.6*	28.0	13.74	28.3	14.4
59	maris-s03-gripper11	SAT	---	---	---	---	---
60	mizh-md5-47-3	SAT	*356.1*	1039.2	---	96.8	---
61	mizh-md5-47-4	SAT	*518.7*	648.2	503.25	604.7	---
62	mizh-md5-47-5	SAT	*592.7*	1179.2	---	355.5	383.8
63	mizh-md5-48-2	SAT	*176.1*	---	---	---	---
64	mizh-md5-48-5	SAT	*1051.7*	1006.0	---	---	---
65	mizh-sha0-35-2	SAT	*324.1*	28.6	103.68	624.5	114.6
66	mizh-sha0-35-3	SAT	*1008.5*	38.3	---	218.5	288.8
67	mizh-sha0-35-4	SAT	*31.0*	831.5	582.4	327.9	206.0
68	mizh-sha0-35-5	SAT	*307.9*	---	---	410.0	172.5
69	mizh-sha0-36-2	SAT	---	---	---	1148.6	---
70	narain-vpn-clauses-6	SAT	*805.8*	1019.8	870.87	1162.3	1043.2
71	schup-l2s-guid-1-k56	UNSAT	*983.7*	676.1	534.37	611.8	520.4
72	schup-l2s-motst-2-k315	SAT	*920.4*	866.3	902.66	867.5	1080.4
73	simon-s02b-dp11u10	UNSAT	*87.6*	198.3	220.47	145.0	255.7
74	simon-s02b-k2f-gr-rcs-w8	UNSAT	---	---	---	---	---
75	simon-s02b-r4b1k1.1	SAT	*566.3*	448.9	730.63	---	866.9
76	simon-s02-w08-18	SAT	*327.0*	155.3	288.35	313.5	264.7
77	simon-s03-fifo8-300	UNSAT	*177.5*	214.1	115.25	294.1	115.0
78	simon-s03-fifo8-400	UNSAT	*335.7*	579.9	338.52	449.0	332.2
79	vange-col-abb313GPIA-9-c	SAT	---	---	---	---	---
80	vange-col-inithx.i.1-cn-54	SAT	*16.3*	36.1	---	24.6	---
81	velev-engi-uns-1.0-4nd	UNSAT	*42.2*	41.7	45.69	32.5	49.4
82	velev-engi-uns-1.0-5c1	UNSAT	*3.8*	4.0	4.4	4.6	4.0
83	velev-fvp-sat-3.0-b18	SAT	---	---	---	---	---
84	velev-live-uns-2.0-ebuf	UNSAT	*32.7*	30.2	29.85	35.6	18.9
85	velev-npe-1.0-9dlx-b71	SAT	---	---	13.22	---	44.9
86	velev-pipe-o-uns-1.0-7	UNSAT	---	---	---	---	---
87	velev-pipe-o-uns-1.1-6	UNSAT	---	---	75.27	---	255.2
88	velev-pipe-sat-1.0-b10	SAT	---	5.3	3.27	---	11.3
89	velev-pipe-sat-1.0-b7	SAT	---	10.1	196.72	394.4	13.1
90	velev-pipe-sat-1.0-b9	SAT	*3.5*	9.6	8.45	3.7	4.1
91	velev-pipe-sat-1.1-b7	SAT	*14.1*	28.7	5.96	337.0	6.0
92	velev-pipe-uns-1.0-8	UNSAT	---	---	439.89	---	267.9
93	velev-pipe-uns-1.0-9	UNSAT	---	---	91.96	---	88.3
94	velev-pipe-uns-1.1-7	UNSAT	*9.5*	---	312.46	---	126.7
95	velev-vliw-sat-2.0-b6	SAT	*123.3*	12.5	16.21	9.3	7.8
96	velev-vliw-sat-4.0-b1	SAT	*9.2*	14.6	29.1	71.7	20.6
97	velev-vliw-sat-4.0-b3	SAT	*12.4*	8.7	14.45	18.7	14.9
98	velev-vliw-sat-4.0-b4	SAT	---	14.7	12.04	17.2	15.5
99	velev-vliw-uns-2.0-iq4	UNSAT	---	---	---	---	---
100	velev-vliw-uns-4.0-9C1	UNSAT	**---**	---	319.06	---	338.4
		Solved:	***72***	**67**	**71**	**66**	**72**

Table 6.11: Comparison of solving times for the SAT Race 2006 benchmarks using different parameter configurations (pt. 2)

Chapter 7

Conclusion and Future Work

Many problems of scientific and industrial interest can be translated into instances of the Boolean satisfiability problem (SAT) and then solved with SAT solvers. The most important SAT solver algorithm is the DPLL algorithm, which in its newest incarnations is controlled by a multitude of heuristics which determine the speed of the solver to a great degree. The heuristic parameters are for example numbers for which "good" values are needed: finding these is an optimization problem. Researchers and industry have published a large library of SAT problems for this reason; the problems serve as benchmarks for optimizing new solvers and algorithms. Manually tuning the (possibly large amount of) heuristic parameters of a DPLL SAT solver is time-consuming; as an alternative, optimization algorithms could automate the cycle of modifying parameter values and testing.

This work presented and tested a fully automatic heuristic parameter optimization procedure that is based on using local search algorithms which attempt to find optimal parameters for training sets of SAT problems. A result configuration is then synthesized drawing from the data that was gathered while optimizing on the training sets. Although the presented procedure makes no assumptions about the details of the local search algorithms (other than that they are capable of dealing with the types of heuristic parameters being optimized), the focus in this work was on optimization with Evolutionary Algorithms (EAs), robust and powerful stochastic local search algorithms modeled after natural evolution. EAs are a sub-field of Soft Computing, a collective term for "imprecise" problem solving techniques that are often inspired by natural processes. They have been applied successfully in a wide variety of fields and are intriguing due to their simplicity and surprising power. Subtypes of EAs differ in the genotype that encodes the solutions; in this work, Genetic Algorithms (GAs), which encode solutions as binary strings, and Evolution Strategies (ESs), which encode solutions as vectors of floating-point numbers, were used.

There exists some work that explored the use of automated reasoning algorithms to perform the optimization, but to the author's knowledge so far none that employ the classic GA and ES for optimizing a DPLL SAT solver's heuristic parameters. This work closes this gap by using implementations of a standard GA and ES to optimize seven parameters of the well-known open-source SAT solver MINISAT on a training set of SAT problems chosen from the portfolio of the SAT Race, a yearly competition for solvers usually held concurrently with the annual SAT conference.

As a countercheck for the results of the EAs and for general investigative purposes a number of fitness landscapes (visualizations of an optimization problem's search space) for the solver parameters on various training problems was generated. This is extremely time-consuming, and the author believes this to be a first in the context of SAT solver optimization.

7.1 Results

Some preliminary experiments were performed for investigating the amount of imprecision that should be expected from run time measurements under Linux, since these measurements would be the objective value for the optimization experiments. It was found that the higher the load of concurrent tasks on a machine is, the more imprecise the measurements become. All following experiments involving time measurements were performed under minimal load, ideally after booting into single-user mode where still a relative error of about 1-2% persists.

Tests of the EA implementations showed that they worked correctly on test functions and could find the optima even when substantial amounts of noise were added. The ES was found to be more precise and faster than the GA, which confirms what is stated in the literature.

The fitness landscapes that were generated for different problems revealed some patterns for three of the parameters, but also showed that the other four parameters would be nearly impossible to optimize since their effects are very random.

The SAT solver was then optimized using the GA and the ES, to find out which of these would serve better as the basis of the optimization procedure. Both algorithms could find the optima of the two parameters that had relatively clearly defined locations for the global optimum, and as expected returned strongly varying results for the other four parameters. It was also found that the ES was more precise and converged quicker, as in the experiments with noisy test functions.

Since the ES was found to be the superior optimization algorithm, it was then used to optimize the SAT solver using training sets made up of one of 27 different SAT problem together with nine shuffled versions of the same problem. The best three of the resulting configurations of each run were charted as points in XY-diagrams and the ranges into which the parameters fell were counted and set into histograms. It was found that optimal values for four of the seven parameters predominantly fell into the same narrow range; for one of the parameters this was unexpected, since it had shown no particular order in the fitness landscape experiments. The remaining three parameters showed no discernible order. For one of the runs all explored configurations were charted as points in three-dimensional diagrams to investigate the behavior of the ES. It was discovered that the ES does not vary one of the parameters to a large degree because its range is relatively large and the variation parameter of the ES does not seem to increase.

After the optimization runs had completed, two simple algorithms for the generation of the final result candidates were applied on respectively the best or the best three configurations collected in each run. One method calculates the candidate as a the average of all parameter values of one type, and the other as the median of those values. The test pool for comparing the candidates consisted of the 100 benchmarks of the final round of the 2006 SAT Race, plus two shuffled versions of each problem. The result generated from computing the median of the best three results of each optimization run was found to be the best overall; it had a slight edge in solving speed compared to the standard MINISAT parameters, but ultimately solved a few problems less after the maximum timeout period than the standard parameters.

7.2 Future work

Although the procedure framework is designed to be fully automatic, the experiments in this work were handled manually and were computed only on single, non-networked computers. All of the necessary steps can be relatively easily automated; for convenience a database system would be useful to store the generated configurations. If given net access, users elsewhere in the world could all feed their optimization results into the same central database, which would hold a huge store of configuration data to mine for patterns and hints on hidden structures in classes of SAT problems.

The SAT solver evaluations are by far the most computation-intensive part of the procedure. Distributed computing would help immensely to keep optimization times down. The SAT solver optimization step is easily paralellizable and would enable the use of much larger training sets, which presumably might yield results of higher quality. In this work the procedure was only tested on MINISAT, other solvers should be tested as optimization targets.

In this work only the simplest possible implementations of the various steps of the optimization procedure were tested. The choice of the problem pools was limited here to relatively small numbers of problems due to the lack of distributed computation. The effects of larger training and test sets should be studied if more computing power is available. The makeup of the training sets was chosen rather arbitrarily here; the effect of using a problem and its shuffled variants may not be the most ideal choice for getting usable optimization results.

In the optimization step, only a relatively simple form of the ES was used here; more powerful versions exist, and may help against the problem of not adequately explored parameters. The ES is also only intended to be used for real-valued variables, but SAT solvers can contain different types of parameters as well, for example Boolean variables.

In the candidate selection phase, only a very simple average and median of parameters was utilized, and then only took into account a small part of the generated configurations. There is much room here for trying sophisticated data mining and statistical methods. The candidate testing should be performed on very large sets of benchmarks to counter the randomness of SAT. Finally, the scoring scheme to determine the final result was very simple in this work and should be improved.

Bibliography

[Ackley87] David H. Ackley. *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers, Norwell, MA, USA, 1987. ISBN 0-89838-236-X.

[Audemard08] Gilles Audemard and Laurent Simon. Experimenting with small changes in conflict-driven clause learning algorithms. In *CP '08: Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*, pages 630–634. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-85957-4. doi: http://dx.doi.org/10.1007/978-3-540-85958-1_55.

[Bäck96] Thomas Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, Oxford, UK, 1996. ISBN 0-19-509971-0.

[Bäck97] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., Bristol, UK, UK, 1997. ISBN 0750303921.

[Baker87] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 14–21. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1987. ISBN 0-8058-0158-8.

[Bayardo97] Roberto J. Bayardo, Jr. and Robert C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence*, AAAI'97/IAAI'97, pages 203–208. AAAI Press, 1997. ISBN 0-262-51095-2. URL http://portal.acm.org/citation.cfm?id=1867406.1867438.

[Beame04] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Int. Res.*, 22:319–351, December 2004. ISSN 1076-9757. URL http://portal.acm.org/citation.cfm?id=1622487.1622497.

[Bergeron00] Janick Bergeron. *Writing testbenches: functional verification of HDL models*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-7766-4.

[Beyer95] Hans-Georg Beyer. Toward a theory of evolution strategies: Self-adaptation. *Evol. Comput.*, 3(3):311–347, 1995. ISSN 1063-6560. doi:http://dx.doi.org/10.1162/evco.1995.3.3.311.

[Beyer98] Hans-Georg Beyer. Evolutionary algorithms in noisy environments: Theoretical issues and guidelines for practice. In *Computer Methods in Applied Mechanics and Engineering*, pages 239–267. 1998.

[Beyer01] Hans-Georg Beyer. *The theory of evolution strategies*. Springer-Verlag New York, Inc., New York, NY, USA, 2001. ISBN 3-540-67297-4.

[Beyer02] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies –a comprehensive introduction. *Natural Computing: an international journal*, 1(1):3–52, 2002. ISSN 1567-7818. doi:http://dx.doi.org/10.1023/A:1015059928466.

[Biere99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207. Springer-Verlag, London, UK, 1999. ISBN 3-540-65703-7.

[Bonissone97] Piero P. Bonissone. Soft computing: the convergence of emerging reasoning technologies. *Soft Comput.*, 1(1):6–18, 1997.

[Boole54] George Boole. *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. 1854. URL http://www.gutenberg.org/ebooks/15114.

[Bryant01] Randal E. Bryant, Steven German, and Miroslav N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Comput. Logic*, 2(1):93–134, 2001. ISSN 1529-3785. doi:http://doi.acm.org/10.1145/371282.371364.

[Burch92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992. ISSN 0890-5401. doi:http://dx.doi.org/10.1016/0890-5401(92)90017-A.

[Burch94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 68–80. Springer-Verlag, London, UK, 1994. ISBN 3-540-58179-0.

[Buro93] M. Buro and H. K. Buening. Report on a sat competition, 1993.

[Coley98] David A. Coley. *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1998. ISBN 9810236026.

[Cook71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, New York, NY, USA, 1971. doi:http://doi.acm.org/10.1145/800157.805047.

[Crawford96] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3-sat. *Artif. Intell.*, 81:31–57, March 1996. ISSN 0004-3702. doi:10.1016/0004-3702(95)00046-1. URL http://portal.acm.org/citation.cfm?id=233339.233347.

[Davis60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. ISSN 0004-5411. doi:http://doi.acm.org/10.1145/321033.321034.

[Davis62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/368273.368557.

[Davis87] Lawrence Davis. *Genetic Algorithms and Simulated Annealing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987. ISBN 0934613443.

[De Jong75] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* Ph.D. thesis, Ann Arbor, MI, USA, 1975.

[De Jong97] Fogel D.B. De Jong, Kenneth A. and Schwefel H.-P. A history of evolutionary computation. IOP Publishing Ltd., Bristol, UK, UK, 1997. ISBN 0750303921.

[Deb99] K. Deb and H.-G. Beyer. Self-Adaptation in Real-Parameter Genetic Algorithms with Simulated Binary Crossover. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 172–179. Morgan Kaufmann, San Francisco, CA, 1999.

[Dershowitz05] N. Dershowitz, Z. Hanna, and J. Katz. Bounded model checking with qbf. In *in Intl Conf. on Theory and Applications of Satisfiability Testing*, pages 408–414. Springer, 2005.

[Dim] Dimacs (center for discrete mathematics and theoretical computer science). URL http://dimacs.rutgers.edu/.

[Dixon78] L.C.W. Dixon and G.P. Szegö. The global optimization problem: An introduction. In *Towards Global Optimization 2*, pages 1–15. North-Holland Pub. Co., 1978.

[Dixon04] Heidi E. Dixon, Matthew L. Ginsberg, and Andrew J. Parkes. Generalizing boolean satisfiability i: Background and survey of existing work. *J. Artif. Intell. Res. (JAIR)*, 21:193–243, 2004.

[Eén04] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer Berlin / Heidelberg, 2004. URL http://dx.doi.org/10.1007/978-3-540-24605-3_37.

[Eén05a] N. Eén and N. Sörensson. MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization, System description for the SAT competition, 2005. URL http://minisat.se/Papers.html.

[Eén05b] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Theory and Applications of Satisfiability Testing*, pages 61–75. 2005. doi:10.1007/11499107_5. URL http://dx.doi.org/10.1007/11499107_5.

[Eiben08] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Natural Computing. Springer, October 2008. ISBN 3540401849.

[Freeman95] Jon William Freeman. *Improvements to propositional satisfiability search algorithms (PhD thesis)*. Ph.D. thesis, Philadelphia, PA, USA, 1995.

[Fukunaga04] Alex S. Fukunaga. Evolving local search heuristics for sat using genetic programming. In *Genetic and Evolutionary Computation — GECCO 2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 483–494. Springer Berlin / Heidelberg, 2004. URL http://dx.doi.org/10.1007/978-3-540-24855-2_59.

[Garey90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0716710455.

[Gill81] Philip E. Gill, Walter. Murray, and Margaret H. Wright. *Practical optimization / Philip E. Gill, Walter Murray, Margaret H. Wright*. Academic Press, London ; New York :, 1981. ISBN 0122839528 0122839501.

[Giunchiglia02] Enrico Giunchiglia, Marco Maratea, and Armando Tacchella. Dependent and independent variables in propositional satisfiability. In Sergio Flesca, Sergio Greco, Giovambattista Ianni, and Nicola Leone, editors, *Logics in Artificial Intelligence*, volume 2424 of *Lecture Notes in Computer Science*, pages 296–307. Springer Berlin / Heidelberg, 2002.

[Goldberg89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. ISBN 0201157675.

[Gomes98] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 431–437. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1998. ISBN 0-262-51098-7. URL http://portal.acm.org/citation.cfm?id=295240.295710.

[Grefenstette] John J. Grefenstette. A user's guide to genesis: Genetic search implementation system 5.0.

[Gu96] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (sat) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1996.

[Hachtel96] Gary D. Hachtel and Fabio Somenzi. Logic synthesis and verification algorithms, 1996.

[Haken84] Armin Haken. *The intractability of resolution (complexity)*. Ph.D. thesis, Champaign, IL, USA, 1984.

[Hansen96] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. *In Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, pages 312–317, 1996.

[Herbstritt04] Marc Herbstritt and Bernd Becker. Conflict-based selection of branching rules. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 290–291. Springer Berlin / Heidelberg, 2004.

[Holland92] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0262082136.

[Hooker95] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[Hoos04] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004. ISBN 1-55860-872-9.

[Huang07] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th international joint conference on Artifical intelligence*, pages 2318–2323. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. URL http://portal.acm.org/citation.cfm?id=1625275.1625649.

[Hutter07] Frank Hutter, Domagoj Babic, Holger H. Hoos, and Alan J. Hu. Boosting verification by automatic tuning of decision procedures. *Formal Methods in Computer Aided Design*, 0:27–34, 2007. doi: http://doi.ieeecomputersociety.org/10.1109/FAMCAD.2007.9.

[Jain07] H. Jain and E. Clarke. Sat solver descriptions: Cmusat-base and cmusat. *System description for the SAT competition 2007*, 2007.

[Jeroslow90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990. ISSN 1012-2443. URL http://dx.doi.org/10.1007/BF01531077. 10.1007/BF01531077.

[Karatsuba62] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. In *Proceedings of the USSR Academy of Sciences 145*, pages 293–294. 1962.

[Karp72] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kibria06] Raihan H. Kibria and You Li. Optimizing the initialization of dynamic decision heuristics in dpll sat solvers using genetic programming. In *EuroGP*, pages 331–340. 2006.

[Kibria07] Raihan H. Kibria. Evolving a neural net-based decision and search heuristic for dpll sat solvers. In *IJCNN*, pages 765–770. 2007.

[Kilby76] J.S. Kilby. Invention of the integrated circuit. *Electron Devices, IEEE Transactions on*, 23(7):648 – 654, jul. 1976. ISSN 0018-9383.

[Koza92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, December 1992. ISBN 0262111705.

[Li96] Chu Min Li. Exploiting yet more the power of unit clause propagation to solve 3-sat problems. *ECAI'96 Workshop on Advances in Propositional Deduction*, pages 11–16, 1996.

[Li97] Chu Min Li and Anbulagan Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th international joint conference on Artifical intelligence - Volume 1*, pages 366–371. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-555860-480-4. URL http://portal.acm.org/citation.cfm?id=1624162.1624216.

[Li00] Chu Min Li. Integrating equivalency reasoning into davis-putnam procedure. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 291–296. AAAI Press, 2000. ISBN 0-262-51112-6. URL http://portal.acm.org/citation.cfm?id=647288.760210.

[Liberatore00] Paolo Liberatore. On the complexity of choosing the branching literal in dpll. *Artif. Intell.*, 116(1-2):315–326, 2000. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/S0004-3702(99)00097-1.

[Luby93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47:173–180, September 1993. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/0020-0190(93)90029-9. URL http://dx.doi.org/10.1016/0020-0190(93)90029-9.

[Lynce01] I. Lynce and J. P. Marques-Silva. The puzzling role of simplification in propositional satisfiability. *EPIA'01 Workshop on Constraint Satisfaction and Operational Research Techniques for Problem Solving (EPIA-CSOR'01)*, December 2001.

[Lynce04] Inês Lynce and João P. Marques Silva. On computing minimum unsatisfiable cores. In *SAT*. 2004.

[Marić10] Filip Marić and Predrag Janičić. Formal correctness proof for dpll procedure. *Informatica*, 21:57–78, January 2010. ISSN 0868-4952. URL http://portal.acm.org/citation.cfm?id=1804226.1804231.

[Marques-Silva99] Joao P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/12.769433.

[Marques-Silva08] J. Marques-Silva. Practical applications of boolean satisfiability. *Workshop on Discrete Event Systems (WODES'08)*, 2008.

[Michalewicz94] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (2nd, extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1994. ISBN 3-540-58090-5.

[Moore65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965. doi:10.1109/JPROC.1998.658762. URL http://dx.doi.org/10.1109/JPROC.1998.658762.

[Moskewicz01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. pages 530–535. 2001.

[Murty87] Katta Murty and Santosh Kabadi. Some np-complete problems in quadratic and nonlinear programming. *Mathematical Programming*, 39:117–129, 1987. ISSN 0025-5610. URL http://dx.doi.org/10.1007/BF02592948. 10.1007/BF02592948.

[Nadel10] Alexander Nadel and Vadim Ryvchin. Assignment stack shrinking. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing SAT 2010*, volume 6175 of *Lecture Notes in Computer Science*, pages 375–381. Springer Berlin / Heidelberg, 2010. URL http://dx.doi.org/10.1007/978-3-642-14186-7_35.

[Nam99] Gi-Joon Nam, Karem A. Sakallah, and Rob A. Rutenbar. Satisfiability-based layout revisited: detailed routing of complex fpgas via search-based boolean sat. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, FPGA '99, pages 167–175. ACM, New York, NY, USA, 1999. ISBN 1-58113-088-0. doi:http://doi.acm.org/10.1145/296399.296450. URL http://doi.acm.org/10.1145/296399.296450.

[Nieuwenhuis05] Robert Nieuwenhuis and Albert Oliveras. Decision procedures for sat, sat modulo theories and beyond. the barcelogictools. In *LPAR*, pages 23–46. 2005.

[Nieuwenhuis06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, 2006. ISSN 0004-5411. doi:http://doi.acm.org/10.1145/1217856.1217859.

[Nissen98] Volker Nissen and Jörn Propach. Optimization with noisy function evaluations. In *PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, pages 159–168. Springer-Verlag, London, UK, 1998. ISBN 3-540-65078-4.

[O. Dubois93] Y. Boufkhad J. Carlier O. Dubois, P. Andre. Sat versus unsat. In *D.S. Johnson, M.A. Trick (Eds.), Second DIMACS Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS*, pages 415–436. Providence, RI, 1993.

[Pearl84] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984. ISBN 0-201-05594-5.

[Pena-Reyes00] Carlos Andrés Pena-Reyes and Moshe Sipper. Evolutionary computation in medicine: An overview. *ARTIFICIAL INTELLIGENCE IN MEDICINE*, 19(1):1–23, 2000.

[Pipatsrisawat07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of the 10th international conference on Theory and applications of satisfiability testing*, SAT'07, pages 294–299. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 978-3-540-72787-3. URL http://portal.acm.org/citation.cfm?id=1768142.1768170.

[Pretolani93] D. Pretolani. Efficiency and stability of hypergraph sat algorithms. In *Proceedings of the DIMACS Challenge II Workshop, 1993*. 1993.

[Prosser93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[Rechenberg65] I. Rechenberg. Cybernetic solution path of an experimental problem. In *Royal Aircraft Establishment Translation No. 1122, B. F. Toms, Trans.* Ministry of Aviation, Royal Aircraft Establishment, Farnborough Hants, August 1965.

[Ryvchin08] Vadim Ryvchin and Ofer Strichman. Local restarts. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, pages 271–276. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 3-540-79718-1, 978-3-540-79718-0. URL http://portal.acm.org/citation.cfm?id=1789854.1789879.

[Schwefel65] Hans-Paul Schwefel. *Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik*. Diplomarbeit, Technische Universität Berlin, Hermann Föttinger–Institut für Strömungstechnik, März 1965.

[Schwefel74] Hans-Paul Schwefel. Adaptive Mechanismen in der biologischen Evolution und ihr Einfluss auf die Evolutionsgeschwindigkeit. Technical Report of the Working Group of Bionics and Evolution Techniques at the Institute for Measurement and Control Technology Re 215/3, Technical University of Berlin, July 1974.

[Schwefel81] Hans-Paul Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, Inc., New York, NY, USA, 1981. ISBN 0471099880.

[Schwefel95] Hans-Paul Schwefel. *Evolution and Optimum Seeking*. Sixth-Generation Computer Technology. Wiley Interscience, New York, 1995. ISBN 0-471-57148-2.

[Silva96] Joao P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227. IEEE Computer Society, Washington, DC, USA, 1996. ISBN 0-8186-7597-7.

[Silva99] João P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*, EPIA '99, pages 62–74. Springer-Verlag, London, UK, 1999. ISBN 3-540-66548-X. URL http://portal.acm.org/citation.cfm?id=645377.651196.

[Sinz06] Carsten Sinz. Sat-race 2006. 2006. URL http://fmv.jku.at/sat-race-2006/index.html.

[Sörensson09] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Proceedings of the 12th International Conference on Theory* *and Applications of Satisfiability Testing*, SAT '09, pages 237–243. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-02776-5. doi:http://dx.doi.org/10.1007/978-3-642-02777-2_23. URL http://dx.doi.org/10.1007/978-3-642-02777-2_23.

[T. Schubert07] N. Kalinnik T. Schubert, M. Lewis and B. Becker. Miraxt - a multi-threaded sat solver. *System description for the SAT competition 2007*, 2007.

[Todd94] Stephen Todd and William Latham. *Evolutionary Art and Computers*. Academic Press, Inc., Orlando, FL, USA, 1994. ISBN 012437185X.

[Torn89] Aimo Torn and Antanas Zilinskas. *Global optimization*. Springer-Verlag New York, Inc., New York, NY, USA, 1989. ISBN 0-387-50871-6.

[Tseitin68] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[vanHarmelen07] Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors. *Handbook of Knowledge Representation (Foundations of Artificial Intelligence)*. Elsevier Science, 2007. ISBN 0444522115.

[Vassilev00] Vesselin K. Vassilev, Dominic Job, and Julian F. Miller. Towards the automatic design of more efficient digital circuits. In *EH '00: Proceedings of the 2nd NASA/DoD workshop on Evolvable Hardware*, page 151. IEEE Computer Society, Washington, DC, USA, 2000. ISBN 0-7695-0762-X.

[Wright32] Sewall Wright. The roles of mutation, inbreeding, crossbreeding, and selection in evolution. *Proceedings of the Sixth International Congress on Genetics*, 1932.

[Zabih88] Ramin Zabih and David Mcallester. A rearrangement search strategy for determining propositional satisfiability. In *in Proceedings of the National Conference on Artificial Intelligence*, pages 155–160. 1988.

[Zadeh94] Lotfi A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Commun. ACM*, 37(3):77–84, 1994. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/175247.175255.

[Zhang96] H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*. Fort Lauderdale (Florida USA), 1996.

[Zhang01] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, pages 279–285. IEEE Press, Piscataway, NJ, USA, 2001. ISBN 0-7803-7249-2. URL http://portal.acm.org/citation.cfm?id=603095.603153.

[Zhang02] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 17–36. Springer-Verlag, London, UK, 2002. ISBN 3-540-43997-8.

[Zhang03] Lintao Zhang. *Searching for truth: techniques for satisfiability of boolean formulas*. Ph.D. thesis, Princeton, NJ, USA, 2003. AAI3102236.

Lebenslauf

Name, Vorname	Kibria, Raihan Hassnain
Geburtsdatum/ort	20.07.1976 in Dacca / Bangladesh
Staatsangehörigkeit	Deutsch
1982–1986	Grundschule Goetheschule Mainz
1986–1995	Frauenlob-Gymnasium Mainz, Abschluss: Abitur (Note: 2,3)
01.08.1995–31.08.1996	Zivildienst an den Unikliniken Mainz
September 1996	Grundpraktikum bei Adam Opel AG, Rüsselsheim am Main
1996–2003	Studium Elektrotechnik und Informationstechnik, Studienrichtung Datentechnik an der Technischen Universität Darmstadt, Abschluss: Diplom-Ingenieur (Note: Gut)
17.07.2000–13.10.2000	Fachpraktikum am Institut für Sichere Telekooperation (SIT) der GMD – Forschungszentrum Informationstechnik GmbH Darmstadt
15.05.2003–15.05.2008	Wissenschaftlicher Mitarbeiter am Fachgebiet Rechnersysteme (TU Darmstadt) bei Prof. Dr.-Ing. H. Eveking
01.08.2008–31.01.2009	Softwareentwickler bei kimeta GmbH, Darmstadt
01.02.2009–12.04.2011	Promotionsstudent

Darmstadt, den 11. April 2011