Evolving Combinatorial Problem Instances That Are Difficult to Solve

Jano I. van Hemert http://www.vanhemert.co.uk/
National e-Science Centre, University of Edinburgh, United Kingdom

Abstract

This paper demonstrates how evolutionary computation can be used to acquire difficult to solve combinatorial problem instances. As a result of this technique, the corresponding algorithms used to solve these instances are stress-tested. The technique is applied in three important domains of combinatorial optimisation, binary constraint satisfaction, Boolean satisfiability, and the travelling salesman problem. The problem instances acquired through this technique are more difficult than the ones found in popular benchmarks. In this paper, these evolved instances are analysed with the aim to explain their difficulty in terms of structural properties, thereby exposing the weaknesses of corresponding algorithms.

Keywords

Binary constraint satisfaction, travelling salesman, Boolean satisfiability, 3-SAT, difficult combinatorial problems, problem hardness, evolving problems.

1 Introduction

With the current state of the complexity of algorithms that solve combinatorial problems, the task of analysing computational complexity (Papadimitriou, 1994) has become a task too difficult to perform by hand. To measure progress in the field of algorithm development, many studies now consist of performing empirical performance tests to either show the difference between the performance of several algorithms or to show improvements over previous versions. The existence of benchmarks readily available for download have contributed to the popularity of such studies. Unfortunately, in many of these studies, the performance is measured in a black box manner by running algorithms blindly on benchmarks. This reduces the contribution to some performance results, or perhaps a new found optimum for one or more benchmark problems. What is argued here is that a more interesting contribution can be made by showing performance results in relation to structural properties. From the no-free-lunch theorem (Wolpert and Macready, 1997) we know an overall best optimisation algorithm in general does not exist. Thus, it is important to provide information about where an algorithm excels and where it falls short. Here, we focus on the latter and introduce a methodology for creating difficult to solve problem instances for specific algorithms. After applying this technique we shall analyse these problem instances and then learn about the weaknesses of the those algorithms in terms of the common structure found in the problem instances. With respect to the algorithms, these results are of interest to their developers for further improvement as well as to their users for knowledge about these weaknesses.

To show the potential of using this technique we apply it to three important domains of combinatorial optimisation. The first is binary constraint satisfaction, which

has become important in the last decade as much work on phase transitions has focused on this domain with the aim to explain the large variance found in the level of difficulty of problem instances (Gent et al., 1996a; Kwan et al., 1998; Achlioptas et al., 2001). Second, Boolean satisfiability, or more specifically, 3-satisfiability (3-SAT), which is considered the archetypal problem when constraint satisfaction and the theory of non-deterministic problems is considered (Du et al., 1997). Its active community of developers has produced incredibly fast and successful SAT-solvers, which form important back-ends for other domains such as, automated proof systems (Zhang, 1997) and propositional planning (Kautz and Selman, 1992). Third, travelling salesman (Lawler et al., 1985), which has a long history of empirical studies, also forms an important component in vehicle routing studies (Lenstra and Rinnooy Kan, 1981), which is a field where much empirical comparison studies are performed.

1.1 Previous Work

Evolutionary algorithms are used as testing frameworks in a number studies. To our knowledge, the first occurrences appear in the form of generating tests for a sequential circuit (Saab et al., 1992; Srinivas and Patnaik, 1993; Rudnick et al., 1994), with the objective to locate any faults. Analogous to this is a more recently developed generator for fault testing integrated circuits (Corno et al., 2004). These studies differ from ours as they focus on testing one design at a time, whereas we focus on testing one algorithm at a time. Another difference lies in the goals of the studies. Previous studies have looked for faults in a design, which would make the design false. This study is concerned with the efficiency of combinatorial problem solvers. So far, the only work closely related to this involves estimating the complexity of sorting algorithms, where an evolutionary algorithm is used to provide the samples for the statistical analysis required to obtain the complexity (Cotta and Moscato, 2003).

1.2 Motivation

The central theme of this paper is the use of evolutionary computation for creating difficult to solve combinatorial problem instances. Where difficult to solve refers to computational complexity (Papadimitriou, 1994), which is loosely speaking, the amount of time required by an algorithm to perform its task. From the perspective of an algorithm that solves combinatorial problems, this process can also be described as a stress test, as the aim here is to break its performance in efficiency. In all three problem domains we show that the proposed technique is able to produce interesting problem instances, in the sense that these instances are more difficult to solve than existing benchmarks for a given algorithm and exhibit properties that provide insight into the weaknesses of that algorithm.

When we have repeatedly used the process of evolution, the result is a set of difficult to solve instances. The second stage of the study is to analyse these instances and explain their difficulty in terms of a commonly shared structure. The type of analysis performed depends on the problem domain. For binary constraint satisfaction we will concentrate on the difficulty of unsolvable problem instances. For Boolean satisfaction, we focus on solvable 3-SAT problem instances. Last, for the travelling salesman problem we look at the distribution of path lengths in the symmetric variant of the problem.

1.3 Structure of the Paper

The remainder of the paper is structured as follows. First, in Section 2 we outline the main methodology used to evolve difficult to solve problem instances. Then, we apply

this methodology in three problem domains, binary constraint satisfaction (Section 3), Boolean satisfiability (Section 4), and the travelling salesman problem (Section 5). These sections all make use of the same structure; first, any adaptations to the main methodology are described, then the algorithms applied for solving problems are described, and finally, experiments and analysis of the results are provided. Conclusions are provided in Section 6.

2 Methodology

The goal of the methodology is to create difficult to solve problem instances with respect to a combinatorial problem solver. To meet this goal, the chromosomes represent problem instances. The evolutionary algorithm maintains a population of these problem instances, where their structure is changed over time. The quality of the problem instances is measured by having them solved with an algorithm specifically designed for the job. One may have different definitions of quality; in this study we restrict ourselves to the difficulty of the problem instance by measuring the search effort required by the algorithm under testing. Thus, the goal of the evolutionary algorithm becomes to maximally aggravate the algorithm by trying to increase the search effort that it requires.

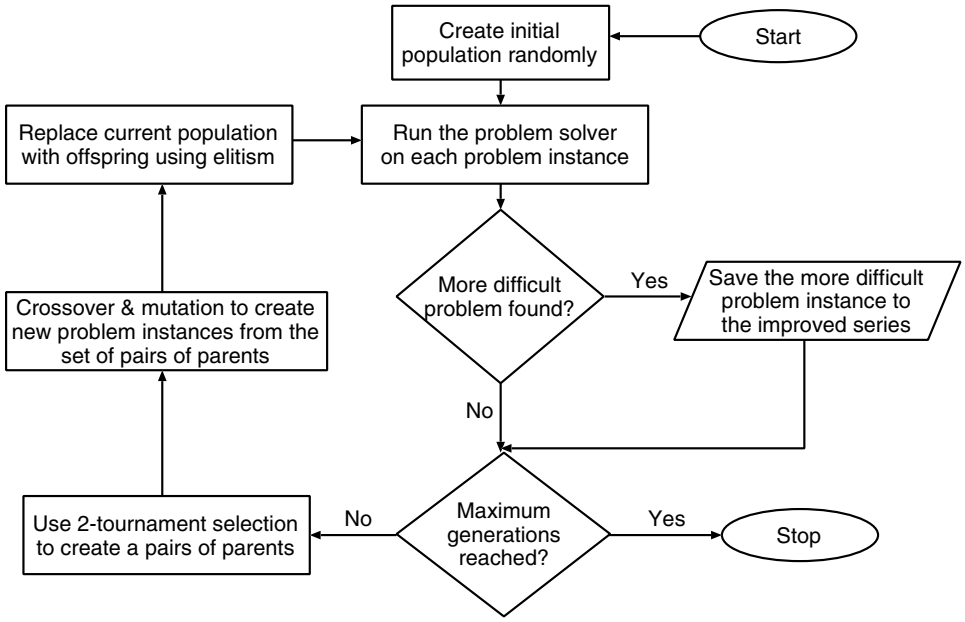


Figure 1: General outline of the evolutionary algorithm to produce difficult to solve problem instances.

The general outline of the algorithm presented in Figure 1 does not differ much from a typical evolutionary algorithm. The process stores an instance whenever it is better than anything found previously. The *series of improved problem instances* that is thus created helps us in the analyses presented later. The following components make up the evolutionary algorithm; they are summarised in Table 1.

2.1 Initialisation

The population size used throughout the experiments in the three problem domains is thirty. Although small, this seems sufficient for the problems in this study to successfully explore the space of problem instances. It also keeps the computational overhead low. The initial population is created uniform randomly with a method that depends on the domain and is always a commonly accepted way of sampling the space of problem instances.

Table 1: A summary of the components of the evolutionary algorithm for evolving difficult to solve combinatorial problem instances.

component	value
individual representation	problem dependent
initialisation	uniform randomly created
population size	30
crossover	uniform crossover
mutation	uniform probability with adapting mutation rate
parent selection	2-tournament
evolutionary model	generational with elitism of size one
termination condition	after fixed number of generations
fitness function	search effort of the combinatorial problem solver
goal	maximisation

2.2 Variation Operators and Selection

The evolutionary algorithm performs a fixed number of generations, set in each individual experiment, before it terminates. This number depends on two major factors. First, it is limited by the intensity of computation required by the problem solvers. Second, it needs to be sufficiently high to allow sufficient time to reach the extremely difficult problem cases. It uses a generational scheme, which employs elitism of size one to keep the best solution in the population (Bäck et al., 1997, C2.7.4). To create the new population, it uses binary tournament selection (Bäck et al., 1997, C2.3) to select two parents, after which crossover is performed to create one offspring. This offspring then undergoes mutation with a mutation rate of pm.

The uniform crossover operator takes two parent individuals as input and creates one offspring. In all problem domains a form of uniform crossover is used. However, the precise definition depends on the representation, and is given separately for each domain.

Mutation is generally defined as a small step made in the space of problem instances. The precise definition is given separately for each problem domain. A common scheme is used for changing the mutation rate during a run. The mutation rate pm is varied over the generations using,

$$pm = pm_{end} + (pm_{start} - pm_{end}) \cdot 2^{\frac{-generation}{bias}},$$

from (Kratica et al., 2003) where the parameters are set as $bias = 3$, $pm_{start} = 1/2$, $pm_{end} = 1/\text{chromsome-size}$, and $generation$ is the current generation. This scheme makes it possible to take reasonably large steps in the search space at the start, while

keeping changes small toward the end of the run. It is similar to the cooling-down process in simulated annealing (Metropolis et al., 1953).

2.3 Fitness Evaluations

To calculate the fitness of newly created offspring we let a tailored algorithm solve it and take as the fitness for that offspring the search effort required by the algorithm. The search effort is measured by counting an atomic operation, which is known to increase exponentially when problems get more difficult (Papadimitriou, 1994). Most often this involves counting the number of constraint checks performed. If the algorithm is complete, we count constraint checks until the algorithm finds a solution or proofs no solution exists. If it is incomplete, we count the constraint checks until the algorithm finds a solution or reaches a predetermined maximum. The goal of the evolutionary algorithm is to maximise the search effort, thus to search for the most difficult to solve problem instances.

3 Binary Constraint Satisfaction

Constraint satisfaction problems (CSPs) (Tsang, 1993) form a class of models representing problems that have as common properties, a set of variables and a set of constraints. The variables should be instantiated from a discrete domain while making sure the constraints that restrict certain combinations of variable instantiations to exist, are satisfied. It is well known that the class of CSP is a member of the class of NP-complete problems (Garey and Johnson, 1979).

The famous CSP problems being considered here are, graph k-colouring, n-queens, and 3-SAT. Over a decade ago, interest has grown in studying binary constraint satisfaction problems, which restrict the general model by only allowing constraints over at most two variables. Rossi et al. proved that for every CSP there exists an equivalent binary CSP (Rossi et al., 1990). However, from empirical evidence we know that the method used to convert one CSP into another may have a large impact on the performance of the algorithms that try to solve it (Bacchus and van Beek, 1998; Walsh, 2000). Here we will be concerned only with binary CSPs created directly, i.e., no conversion has been applied.

A *constraint satisfaction problem* is defined as a tuple $\langle X, D, C\rangle$ where,

- $X = \{x_1, \ldots, x_n\}$ is a finite set of variables,
- $D = \{D_1, \ldots, D_n\}$ is a finite set of domains, one domain for each variable
- and C is a finite set of constraints, every constraint $C_i(x_{i_1}, \ldots, x_{i_j})$ is a subset of $D_{i_1} \times \cdots \times D_{i_j}$, which *restricts* combinations of assignments.

The objective is to assign each variable $x_i \in X$ one value $d \in D_i$ from its domain, denoted as $\langle x_i, d\rangle$, such that none of the constraints in C is violated. This set of assignments is called a solution to the CSP. It is possible to have so many constraints present in the CSP that it is impossible to find a valid value assignment for each variable. Such problem instances are called *unsolvable*. Algorithms that always return a solution if it exists are called complete. Thus, when such an algorithm terminates without returning a solution, we have proof that the problem instance is unsolvable.

A *binary constraint satisfaction problem* (BINCSP) is a CSP where every constraint $c \in C$ restricts at most two variables (Palmer, 1985). Often, network graphs are used to visualise CSP instances. In Figure 2, we provide an example of a the restricting hypergraph of a BINCSP. It consists of three variables $X = \{x_1, x_2, x_3\}$, all

of which have a domain of $D = \{a, b\}$. In a hypergraph every vertice corresponds with one value of the variable it represents in the BINCSP. Every edge shows the value pairs which are forbidden by the set of constraints C. In the example, we show all the edges that correspond to the following set of forbidden value pairs $C = \{ \{\langle x_1, a\rangle, \langle x_2, a\rangle\}, \{\langle x_1, a\rangle, \langle x_3, b\rangle\}, \{\langle x_1, b\rangle, \langle x_2, a\rangle\}, \{\langle x_1, b\rangle, \langle x_2, b\rangle\}, \{\langle x_1, b\rangle, \langle x_3, a\rangle\}, \{\langle x_1, b\rangle, \langle x_3, b\rangle\}, \{\langle x_2, a\rangle, \langle x_3, a\rangle\}, \{\langle x_2, a\rangle, \langle x_3, b\rangle\} \}$.

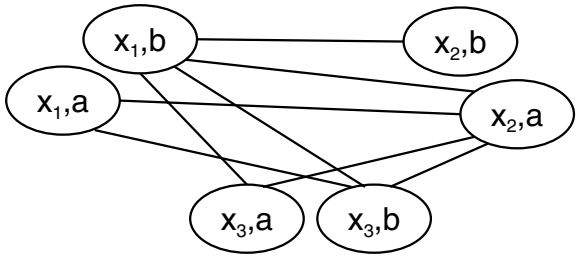


Figure 2: Example of a $|X|$-partite hypergraph of a BINCSP with one solution: $(\langle x_1, a\rangle, \langle x_2, b\rangle, \langle x_3, a\rangle)$.

Cheeseman et al. show for a number of NP-hard problems that these exhibit a transition from solvable to unsolvable problem instances when one looks at an *order parameter*. This parameter characterises a certain aspect of the problem (Cheeseman et al., 1991). For example, in graph colouring the edge connectivity is an order parameter. Such a transition in a complex system is referred to as a *phase transition*. Random BINCSP also exhibits such a transition (Prosser, 1994; Prosser, 1996) when one looks at the order parameters, constraint density and constraint tightness. Here, we shall work with BINCSP in the form of hypergraphs, and the order parameter will be the ratio of forbidden value pairs to the total number of value pairs. Moreover, we assume the domains are equal for each variable, which we denote by D'. Basically, this is the overall tightness of the problem instance. We define this *tightness* of a BINCSP with homogeneous domains $\langle X, D', C\rangle$ as,

$$\textit{tightness}(\langle X, D', C\rangle) = \frac{|C|}{\binom{|X|}{2}|D'|^2},$$

where C is the set of forbidden value pairs. The example in Figure 2 has homogeneous domains where $|D'| = 2$, and consequently with $|C| = 8$ and $\binom{|X|}{2} = 3$ it has a tightness of $2/3$.

An extensive study (Smith, 1994; Smith and Dyer, 1996; Prosser, 1996) on random binary CSPs provides evidence that the phase transition coincides with a peak in the search effort required to either find one solution for an instance or determine that it is unsolvable. The *search effort* is measured as the number of constraint checks needed by an algorithm to either find a solution or to determine that a problem instance is unsolvable. A *constraint check* occurs when an algorithm tests whether the value assignment of two variables is valid, i.e., it is not forbidden by any constraint from the set of constraints C. The class of randomly generated binary CSPs is of interest, because accurate order parameters help explain better why one problem instance is much harder to solve than another.

3.1 Adaptations to the Methodology

The evolutionary algorithm maintains a population of binary CSPs and it changes the structure of the population over time. Basically, its variation operators alter the individual conflict pairs between two values of two variables, i.e., the edges of its corresponding hypergraph. The set of variables and each variable's domain values are left untouched. Only the structure of the constraints can vary. The evolutionary algorithm is run for 500 generations in order to get significantly difficult problem instances.

The population consists of thirty binary CSPs. The initial population is created with Model E (Achlioptas et al., 2001) using the program RandomCSP (van Hemert, 2003b) with $|X| = 15$ variables, a homogeneous domain size $|D'| = 15$ and the p parameter of Model E is set to 0.02. This last parameter determines how many forbidden value pairs will exist on average in the randomly created problem instances. The value chosen here makes sure that the initial population will consist of problem instances with many solutions, which are easy to find. The precise definition of Model E is,

> The graph C^{Π} is a random $|X|$-partite graph with $|D'|$ nodes in each part that is constructed by uniformly, independently and with repetitions selecting $p\binom{|X|}{2}|D'|^2$ edges out of the $\binom{|X|}{2}|D'|^2$ possible ones.

The uniform crossover operator takes two parent individuals, i.e., two binary CSPs, as input and creates offspring in the form of a binary CSP with the same number of variables and domain sizes as the parents. It then iterates through all possible value pairs of the offspring, i.e., possible edges in the corresponding hypergraph, and chooses every time with equal probability either the first or the second parent. It sets the value pair to be a forbidden value pair if and only if the chosen parent's value pair is forbidden. Then the offspring undergoes mutation by performing an iteration of the value pairs where, with a chance of *pm*, the state of a value pair will be flipped, i.e., a conflict pair will be removed if it exists or created if it does not exist.

3.2 Problem Solvers

In the experiments presented next, two constraint solvers will be tested independently; *chronological backtracking* (Golomb and Baumert, 1965) and forward checking with conflict-directed backjumping, both implemented in (van Beek, 2005). The latter algorithm uses the *forward checking* (FC) of Haralick and Elliot (1980) for its constraint propagation, which means that the algorithm first assigns a value to a variable and then checks all unassigned variables to see if a valid solution is still possible. The goal of forward checking is to prune the search tree by detecting inconsistencies as soon as possible. To improve upon the speed, *conflict-directed backjumping* (CBJ) (Dechter, 1990) was added to it by Prosser (1993) to form FC-CBJ. The speed improvement comes from the ability to make larger jumps backward when the algorithm gets into a state where it needs to backtrack to a previous set of assignments by jumping back over variables of which the algorithm knows that these will provide no solutions. For both algorithms no variable ordering heuristic or value ordering heuristic were used.

3.3 Experiments

Preliminary results on evolving binary constraint satisfaction problems are given in (van Hemert, 2003a). However, the results in that study, only marginally improved upon the difficulty of previously generated problem instances. Moreover, in that study, no analysis of the problem instances is performed.

One experiment consists of 100 independent runs of the evolutionary algorithm with one problem solver, as presented in Section 3.1. During one run, we save the following statistics about every generated problem instance; its tightness, the search effort needed to find a solution or determine that it is unsolvable, and whether it is solvable or not. Also, each time the evolutionary algorithm finds a problem instance that is more difficult to solve then any found so far in the run, we save that problem instance. The last one of this series of improved problem instances is commonly referred to as the best individual of the run.

We perform two experiments, one with chronological backtracking (BT) and one with forward checking with conflict-directed backjumping (FC-CBJ). Over a whole experiment, at least 96% of all the problem instances created by the evolutionary algorithm are different.

3.3.1 Convergence of the Evolutionary Algorithm

Figure 3(a) shows how the mean fitness of the evolutionary algorithm is converging in both experiments, averaged over the 100 independent runs. 95% confidence intervals are computed but are not displayed because they are very small. In the first four generations, we observe a rapid increase in the fitness, i.e., the search effort required for the problem instances. Also, at the same time the tightness of the problem is quickly increasing to about 0.38, as shown in Figure 3(b). Problem instances that have such a large tightness have a high probability of not being solvable (Williams and Hogg, 1994; Gent et al., 1996b), which is confirmed by Figure 3(c).

In Figure 3(c) we observe that the ratio of solvable instances drops from one to almost zero within the first five generations. Then, this ratio increases to almost 0.18 at 28 generations, after which it stays low and shows an erratic pattern. This matches the tightness in Figure 3(b), which converges to 0.301, near the estimated location of the phase transition (Williams and Hogg, 1994; Gent et al., 1996b). At the same time, it is on the under-constrained side, where the probability for solvable problems is higher, but where it is finding unsolvable problems that are difficult to solve.

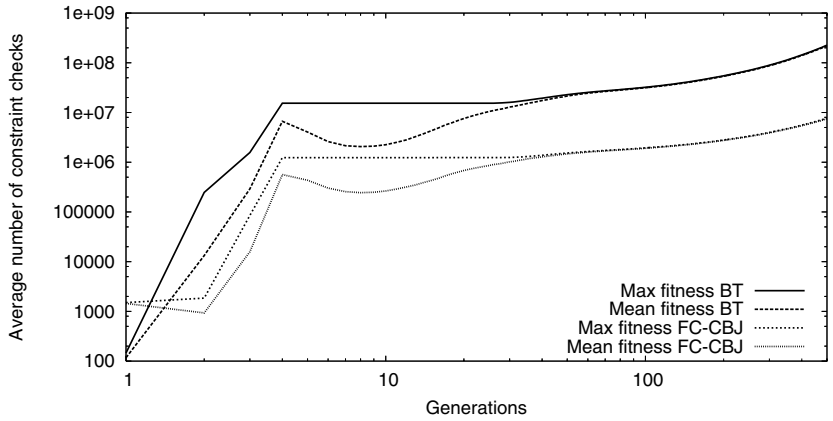
The characterisation of the search of the evolutionary algorithm is almost identical for both complete solvers. The exception is the difference in the search effort, which is significantly lower for the more sophisticated FC-CBJ.

3.3.2 Comparison with Model E

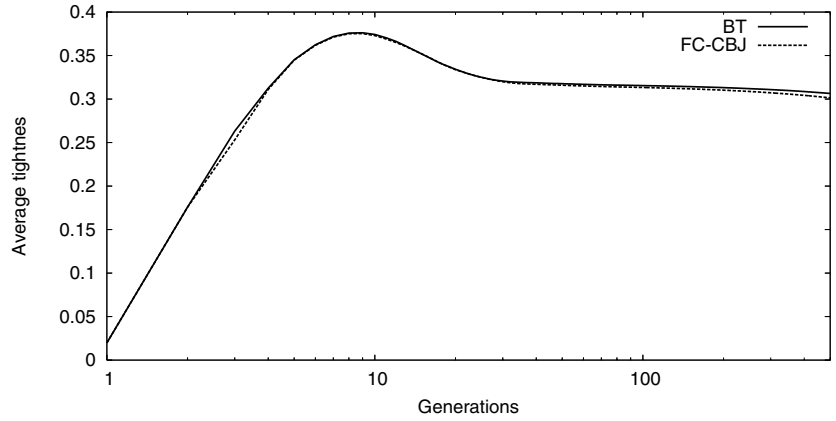
Using Model E, as defined in Section 3.1, we generate one million randomly created BINCSP instances with a tightness equally distributed over the range $(0.29, 0.38)$, which encompasses the phase transition. Of these problem instances we calculate the search effort in conflict checks using both constraint solvers. Figure 4(a) and Figure 4(b) show the minimum, mean and maximum search effort together with the search effort of the most difficult problem instances found by the evolutionary algorithm for the tightness values in that range.

Figure 4 shows, in conjunction with Figure 3(b), the evolutionary algorithm starts finding more difficult problem instances when it is creating problem instances with a tightness of 0.32. Typically, it finds these more difficult instances after only 100 generations of the 500 have passed (see Figure 3(a)). As apparent from Figure 3(b), the evolutionary algorithm progresses by finding problem instances with lower tightness values, thereby getting closer to where more solvable problem instances are located. Its convergence in the conflict checks is also observed in Figure 4 as the flattened peak between 0.295 and 0.320 for BT and 0.295 and 0.309 for FC-CBJ.

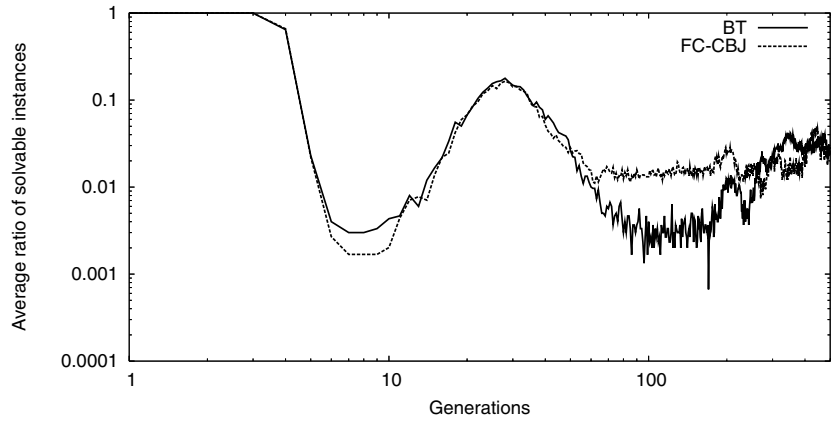
As a final note we point out the large variation of difficulty found in the phase



(a) Mean and maximum fitness, in number of constraint checks required for solving the problem, per generation instances

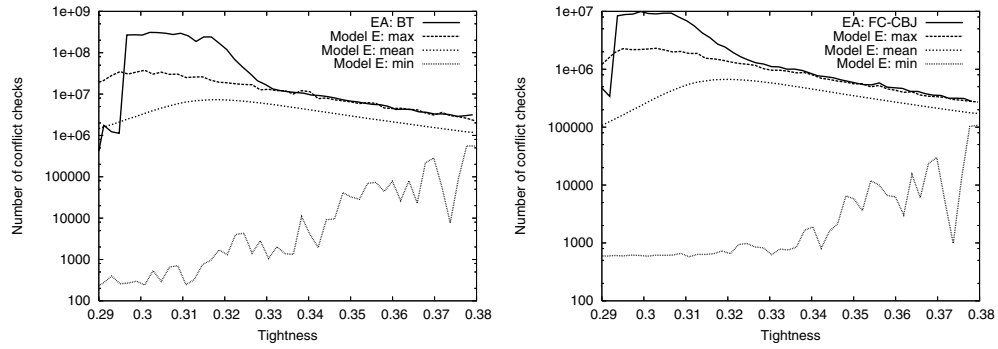


(b) Mean tightness of the problem instances per generation



(c) Mean ratio of solvable instances per generation

Figure 3: Convergence analysis averaged over 100 runs of the EA for BT and FC-CBJ.



(a) Chronological backtracking (BT)

(b) Forward checking with conflict-directed backjumping (FC-CBJ)

Figure 4: The number of conflict checks for problem instances with a tightness in the range $(0.29, 0.38)$. Presented are the most difficult problem instances created by the evolutionary algorithm averaged over 100 independent runs and the maximum, mean and minimum of constraint checks on one million problem instances randomly created with Model E.

transition, seen by the large difference between the minimum and maximum amount of conflict checks required to solve problems as created by Model E. The peak of the maximum difficulty is shifted to the left, i.e., lower tightness values, with respect to the peak of the average difficulty.

3.3.3 Minimal Unsolvable Sub-Problems

A *minimal unsolvable sub-problem* (MUS) is defined as the smallest sub-problem of an unsolvable problem that is itself unsolvable (Mammen and Hogg, 1997). Thus, any subset taken from a MUS is solvable. It was first studied as minimal satisfiable subsets in SAT (Gent and Walsh, 1996). This property is only of interest to unsolvable problem instances, which matches perfectly the problem instances found in the experiments by the evolutionary algorithm.

Intuitively, one would assume that problem instances with a large MUS are more difficult to prove unsolvable than those with a small MUS because a large MUS may contain solvable sub-problems. These sub-problems can prolong the search of a constraint solver for an unsolvable sub-problem. However, according to Culberson and Gent in (Culberson and Gent, 1999) and (Gent and Walsh, 1996), extremely difficult problem instances should have small minimal unsolvable sub-problems which are unique, but difficult problem instances have larger MUSes, which are not unique. Their results however, suggest that this may not hold for large problem instances. The uniqueness of the MUS should make it difficult for a solver to determine unsolvability.

We calculate the size of the minimal unsolvable sub-problem for every unsolvable problem instance in the series of improved instances for each run. Without exception, after the first twenty generations, the size of the MUS equals the number of variables of every problem instance. In other words, the problem instance itself is the MUS. There is only one MUS, which makes it unique. Furthermore, it is also the maximum size and therefore satisfies the conjectures about which problems are most difficult. However, as this size is the same throughout the search, it does not help explain why the evolutionary algorithm keeps on finding unsolvable problems that take more and more search

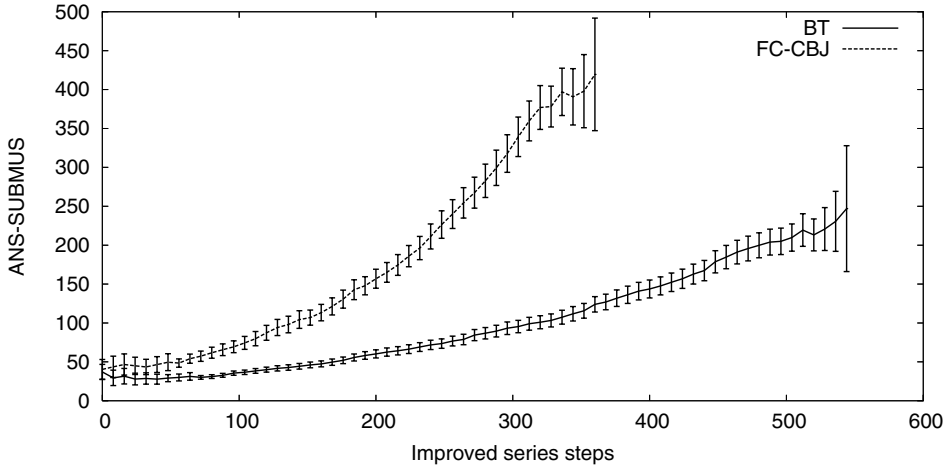


Figure 5: The average number of solutions for subsets of size $n-1$ (ANS-SUBMUS) throughout the improved series over 100 independent runs of the EA for BT and FC-CBJ. Included are 95% confidence intervals.

effort as shown in Figure 3(a).

We introduce a measurement to further explain the increase in difficulty. By definition the MUS is the smallest sub-problem that is unsolvable, hence each sub-problem of a MUS is solvable. We take every possible sub-problem that forms a strict subset of a MUS and that is as large as possible. Together, these subsets form SUBMUS, defined as,

$$\text{SUBMUS} = \{P | \forall P, Q \subset \text{MUS} : |Q| \leq |P|\}\,.$$

The problem instances in the experiments all have $n = 15$ variables, and because the size of the MUS is always n, we get $|\text{SUBMUS}| = n = 15$. The subsets are all of size $n - 1 = 14$.

We can calculate the number of solutions to a sub-problem belonging of a SUBMUS using a complete problem solver. Let *solutions*(s) provide the number of solutions to a sub-problem s, then

$$\text{ANS-SUBMUS} = \frac{1}{|\text{SUBMUS}|} \sum_{s \in \text{SUBMUS}} \textit{solutions}(s)$$

defines the *average number of solutions* (ANS) for the sub-problems of the minimal unsolvable sub-problem (SUBMUS).

We take the series of improved instances for all 100 runs and plot the ANS-SUBMUS throughout it. The result is shown in Figure 5 for BT and FC-CBJ. It shows a clear trend where throughout the improved series, i.e., with increasingly more difficult problem instances, we find larger ANS-SUBMUSes. The 95% confidence intervals are sufficiently small to claim significant results.

3.4 Conclusions on Binary Constraint Satisfaction

Although on average, problem instances become more difficult to solve near the phase transition, the variation in the search effort required to solve problem instances in a

small range of the tightness may vary immensely around the phase transition. By using an evolutionary algorithm to evolve BINCSPs we are able to find difficult to solve problem instances without needing to know the exact structural properties that make these problem instances so difficult to solve.

Exceptionally difficult problem instances, as they are called in (Smith and Grant, 1997), form a very small subset of the set of problem instances with the same order parameters. In (Smith and Grant, 1997), $50\,000$ problem instances are created for each setting of the tightness in order to locate them. Our evolutionary algorithm detects problem instances with similar difficulty by generating only $3\,000$ of them. It then continues to find more and more difficult problem instances. Potentially, this makes it a valuable tool to locate interesting problem instances for the performance analysis of constraint solvers.

The series of improved problem instances in runs of the evolutionary algorithm consist of only unsolvable problems. The size of the minimal unsolvable sub-problem for these instances provides a first reason for why these problems are difficult to solve. However, we find every one of them has the same size, which prohibits it from explaining the significant increase in difficulty in every series. We propose to look at the number of solutions to the subsets of size $n-1$ of these unsolvable problems.

The difficult problem instances found during the runs of the evolutionary algorithm all have a minimal unsolvable sub-problem of size n. When observing the n solvable sub-problems of size $n-1$ of each problem instance, we notice a trend where the number of solutions for sub-problems of size $n-1$ increases when the difficulty increases. This suggests that the difficulty of such an unsolvable problem is caused by the intricate way in which the combination of n solvable problems of size $n-1$ create exactly one unsolvable problem. A larger number of solutions possible for each of the sub-problems make it more difficult to detect that the whole problem is unsolvable.

4 Boolean Satisfiability

A Boolean satisfiability problem $\langle U, W \rangle$ in conjunctive normal form consists of a finite set of boolean variables U and a finite set of clauses W. Each clause $w \in W$ is in the form $\langle l_1 \vee l_2 \vee \ldots \vee l_k \rangle$, where l_i, $1 \leq i \leq k$ are called literals and a literal l_i is either u or $\neg u$ with $u \in U$. The objective is to assign each variable $u \in U$ either false or true such that all the clauses in W are true.

Boolean satisfiability was the first problem to be proved non-deterministic polynomial (Cook, 1971), and two decades later, it was one of the first problems where the phenomenon of a phase transition was found (Cheeseman et al., 1991). The phase transition shows the transition from solvable problems to unsolvable problems under an order parameter. For randomly generated satisfiability, the most commonly used order parameter is the number of clauses divided by the number variables. Empirical evidence shows that the location where we have a 50% chance that an instance is either solvable or unsolvable, coincides with a peak in the difficulty of solving these. This location lies at $m/n \approx 4.25$ (Mitchell et al., 1992; Selman et al., 1996; Hayes, 1997). However, the many studies on this location show large variations and exceptions to this (Cook and Mitchell, 1997).

4.1 Adaptations to the Methodology

We represent a 3-SAT problem by a list of 420 natural numbers. A number in the list, i.e., a gene, corresponds to a unique clause with three different literals. The number of possible unique clauses depends on the number of variables and the size of the clause.

Here, the number of variables is set to 100 and the size of the clause is three, hence it is equal to $1\,313\,400$. Although this representation seems more cumbersome to a straightforward representation that just uses one gene for every literal, and then takes groups of three genes to form clauses, there are strong advantages. Most importantly, it prevents duplicate variables in clauses, which reduces the state space and could otherwise introduce trivial clauses, e.g., $(x \vee \neg x \vee y)$, or 2-SAT clauses, e.g., $(x \vee x \vee y)$. Also, the variation operators now simply become mutation and uniform crossover for lists of natural numbers over a fixed domain.

The major difference with the other two problem domains lies in the fitness function. We have chosen here to try to evolve satisfiable problem instances, i.e., problem instances with a solution. However, during the search, the evolutionary algorithm consistently explores the space of unsatisfiable problem instances. Therefore, we have altered the fitness function such that given two SAT problems A and B, we say that A is better than B iff A and B are either both solvable or unsolvable, and A takes more time to solve than B *or* A is solvable and B is not solvable. This simple approach meets our needs, as the evolutionary algorithm finds difficult satisfiable problems. It may hamper the efficiency of our search as we do not know its effect on the search landscape.

The number of generations until the evolutionary algorithm terminates is set to 200 generations.

4.2 Problem Solvers

Both problem solvers used for solving 3-SAT are based on the Davis-Putnum Procedure (Davis and Putnam, 1960) but are highly optimised. This process of optimising SAT solvers is fuelled partially by an annual competition (Le Berre and Simon, 2005). The two solvers used here, zChaff and Relsat, compete in this competition and score well.

zChaff (Fu, 2004) is based on Chaff (Moskewicz et al., 2001), a SAT solver that employs a particularly efficient implementation of Boolean constraint propagation and a novel low overhead decision strategy. Relsat (Bayardo, 2005) is explained in (Bayardo Jr and Schrag, 1997a; Bayardo Jr and Pehoushek, 2000). It employs constraint satisfaction lookback techniques such as conflict-directed backjumping (Dechter, 1990). The authors put emphasis on using relevance-bounded learning to improve the effectiveness, which keeps new learnt clauses during solving for as long as they are relevant. A clause is relevant if it contains sufficient number of variables currently assigned in the search.

In both solvers we count the number of states of instantiations. This is the amount of nodes in the Davis-Putnum search tree, which both algorithms use. Their difference in operation, and thus in performance, lies in how they build up the tree and how they traverse it.

4.3 Experiments and Analysis

The difficulty of the resulting instances are compared with other benchmark suits. Then we show that structures typically used to explain difficulty, do not explain the increase in difficulty of evolved instances. Last, we provide a property of the instances that does correlate with the difficulty of the problem instances, which is based on the way variables are distributed over clauses.

4.3.1 Comparison to other Benchmark Suits

We compare the difficulty of 600 problem instances acquired using the evolutionary algorithm with other public benchmarks with the same number of variables. A num-

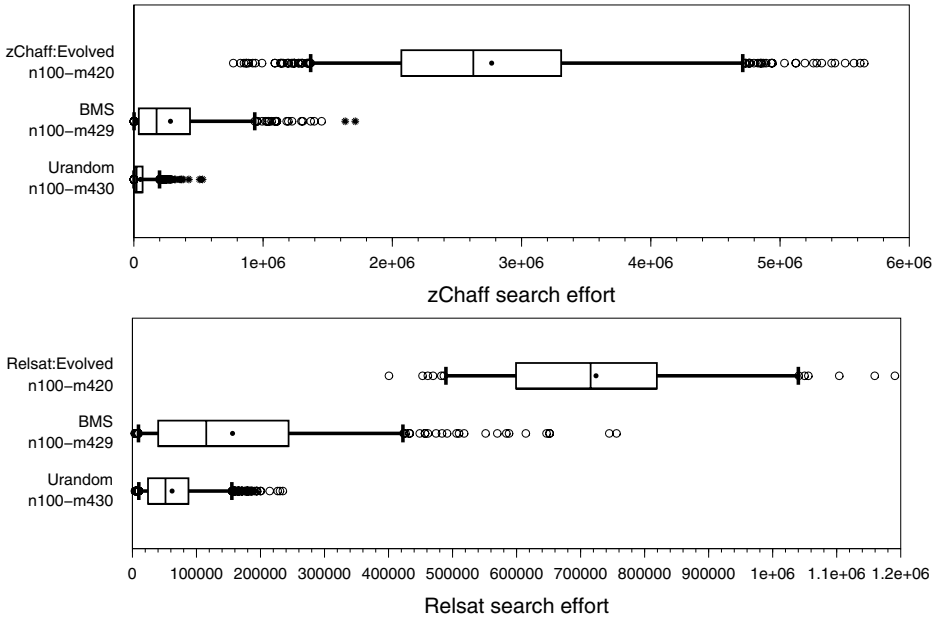


Figure 6: Box-and-whisker plots of the search effort in terms of the number of states of instantiations required by zChaff and Relsat on the zChaff:Evolved set and Relsat:Evolved set respectively, compared to the Backbone Minimal Set (BMS) and the uniform randomly generated set (Urandom).

ber of benchmark suits exists, often subdivided into sets with similar characteristics. We choose the set that first of all, has the same number of variables, then consists of the most difficult instances on average compared to the other sets, and has a similar number of clauses.

Figure 6 shows a comparison of the evolved sets to two other benchmarks for both zChaff and Relsat respectively. These benchmarks are the Backbone-Minimal Subinstance (BMS) (Singer et al., 2000) and the uniform random 3-SAT (Urandom) (Cheeseman et al., 1991). The terminology around backbones is explained in the next section. The first is obtained by taking randomly generated 3-SAT instances with a backbone of size 90 and then stripping these instances without affecting the backbone. The latter is the commonly accepted way of randomly generating k-SAT instances, which we also use to seed the first generation. For both algorithms we note that performing 200 generations is sufficient to evolve significantly more difficult to solve problem instances.

In Figure 7, we show the best fitness value at each improvement step averaged over 580 independent runs of the EA for zChaff and 1480 runs for Relsat. There is a smooth and monotonic increase in difficulty in both experiments. This result helps to make an important observation relevant for the results in the next sections; the smooth increase means we cannot sat any of the rapid changes and peaks in those results are because of jumps in the difficulty of problem instances.

4.3.2 Backbone Size and Number of Solutions

One of the most popular structures that is used to explain the difficulty of problem instances is the backbone. Although multiple definitions exist, also depending on the

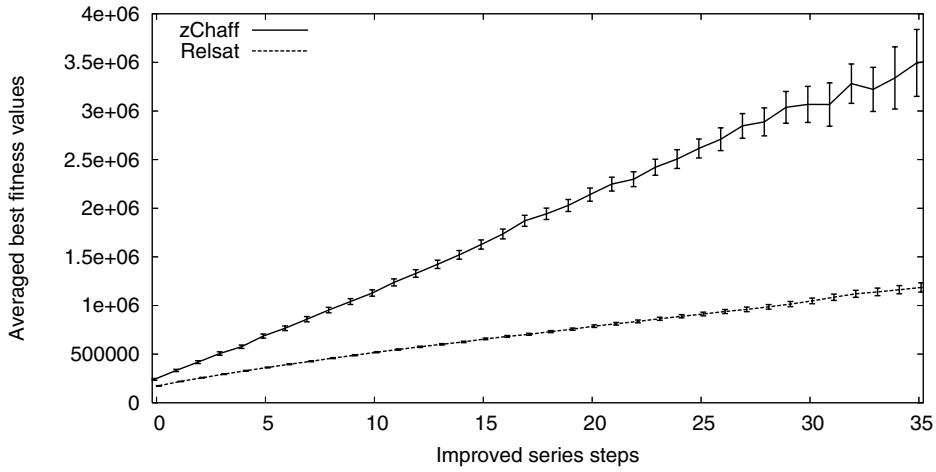


Figure 7: The increase in difficulty, i.e., the best fitness values, in terms of number of instantiations throughout the improved series for both zChaff and Relsat, averaged over respectively 580 and 1450 independent runs of the evolutionary algorithm. Results are presented with 95% confidence intervals.

type of constraint satisfaction problem (Boettcher et al., 2005), it was first defined for Boolean satisfiability and the standard definition is still used most often. The backbone of a problem instance $\langle U, W \rangle$ is the subset of variables $U' \subset U$ where for *every* solution to the problem, a variable $u \in U'$ always has the same truth assignment. There is a corresponding problem in physics, the spin glass model, from which the backbone property was first derived (Monasson et al., 1999).

The conjecture is that the size of the backbone correlates positively with problem difficulty in many combinatorial problems (Slaney and Walsh, 2001). However, recent empirical and theoretical aversive results were published. These respectively showed that almost no correlation exists (Kilby et al., 2005) and gave proof that backbone free problem instances can still be difficult to solve (Beacham, 2000).

In Figure 8, we show the size of the backbone throughout the improved series. The results depict the mean and 95% confidence intervals over 580 independent runs of the EA for zChaff and 1480 runs for Relsat. The latter algorithm requires more samples to reduce statistical variance. The results lack a clear trend, which indicates the evolutionary algorithm is not exploiting the size of the backbone in order to obtain difficult instances. We note that throughout one run the size of the backbone can vary considerably, but that on average it lies between 70% and 75% of the whole set of variables.

Closely related to the backbone is the number of solutions to a satisfiability problem. For larger backbones less freedom exists as only the non-backbone variables can have different truth values in two different solutions. The first conjecture on explaining constraint problem difficulty was founded on the idea that a small number of solutions would be hard to find (Smith, 1994; Smith and Dyer, 1996). Recent studies try to introduce special cases where problem instances have an induced number of solutions (Achlioptas et al., 2000; 2004) . Figure 9 shows the average number of solutions to problem instances over the generations. Although a slight decline may be observed in the first few generations, the number stays well over $5\,000$ solutions. At the end of

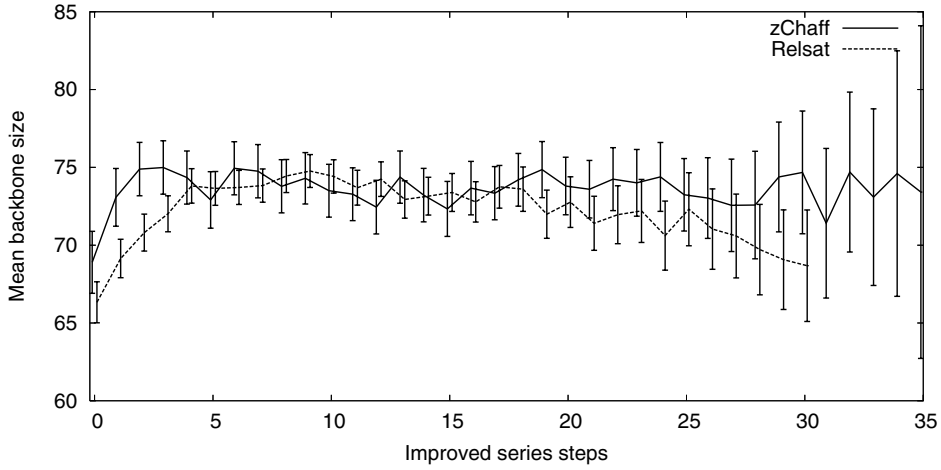


Figure 8: The average size of the backbone of instances throughout the improved series for both zChaff and Relsat, averaged over respectively 580 and 1450 independent runs of the evolutionary algorithm. Results are presented with 95% confidence intervals.

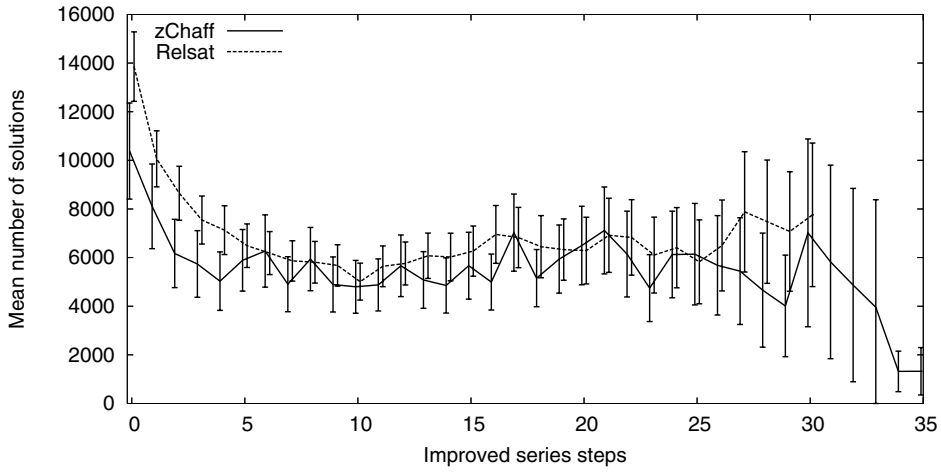


Figure 9: The average number of solutions to instances throughout the improved series for both zChaff and Relsat, averaged over respectively 580 and 1450 independent runs of the EA. It is presented with 95% confidence intervals.

the run the statistical variance is high due to a low number of long series of improved instances.

4.3.3 Variance in Variable Frequency

We examine the variance in the frequency of occurrence of variables in clauses, where we sum over both the negated and non-negated occurrence of a variable. Given 420 clauses of size 3, where we restrict clauses to contain only unique variables, a variable can occur between 0 and 420 times in a problem. When we observe problems created randomly, such as those used to initialise the evolutionary algorithm, most variable

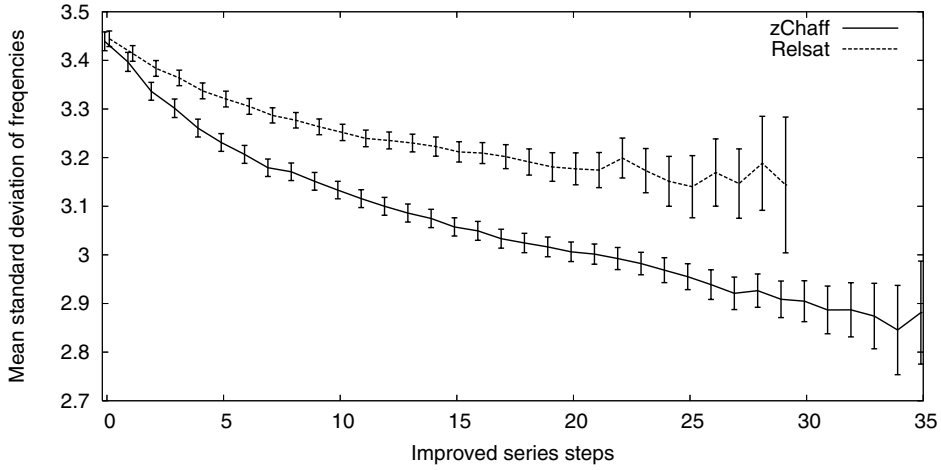


Figure 10: The standard deviation of the distribution of variables in evolved SAT problems per generation for zChaff and Relsat. At each generation the data is averaged over respectively 580 and 1450 independent runs of the EA. It is presented with 95% confidence intervals.

frequencies will be close to a normal distribution. Hence, a variation exists in the frequency in which variables occur in a problem.

In Figure 10, we show the average standard deviation of the frequency in which variables occur in the problem. We take the improved series for both algorithms (again 580 runs for zChaff and 1450 runs for Relsat). It is clear from the confidence intervals that the amount of variation in both measurements, especially at the start of the series, is small. A clear decreasing trend is witnessed for both algorithms. This indicates a mechanism used by the evolutionary algorithm to perform its search in terms of the structure of the problem. Moreover, we have studied the trend of the ratio of negative to non-negative clauses during the improved instances, where we found no significant deviation from the 50%-50% distribution at any point.

4.4 Conclusions on Boolean Satisfiability

Considering the latest results that no correlation exists for SAT between problem difficulty and backbone size, it is not surprising to find the lack of a clear trend in the backbone size in the improved series. We do note that on average over a whole run the backbone size is between 70% and 75%, but no clear relationship can be found with the difficulty of evolved problem instances. Also, no clear trend is present in the number of solutions.

Undoubtedly, a trend is visible in the variance in frequency of variable occurrence. This seems a reasonable candidate for the structural property that induces the increase in difficulty during evolution. First, it is a simple mechanism, which relies on transforming problem instances to form a narrow Gaussian distribution of the variables over all the clauses in a SAT problem. Second, in a previous study (Bayardo Jr and Schrag, 1997b) a significant rise in difficulty was achieved by creating SAT problems where the distribution of variables is close to a uniform distribution, rather than a normal distribution. The latter is obtained by selecting literals at random and is the most common way of creating random SAT, while the first randomly selects literals from a

pre-constructed bag of literals. This bag of literals is created such that each variable is represented equally. However, in that study, the clause size varied, and the peak in difficulty was found distant from the regular location. Here we show that SAT instances with less variance exist that are more difficult than random SAT and that coincide with the regular peak in difficulty.

5 Travelling Salesman

The travelling salesman problem (TSP) is well known to be NP-complete (Papadimitriou and Steiglitz, 1976). It is mostly studied in the form of an optimisation problem where the goal is to find the shortest Hamiltonian cycle in a given weighted graph (Lawler et al., 1985). Here we will restrict ourselves to the *symmetric* travelling salesman problem, i.e., distance$(x, y) =$ distance(y, x), with Euclidean distances in a two-dimensional space.

Over time, much study has been devoted to the development of better TSP solvers. Where "better" refers to algorithms being more efficient, more accurate, or both. It seems, while this development was in progress, most of the effort went into the construction of the algorithm, as opposed to studying the properties of travelling salesman problems. The work of (Cheeseman et al., 1991) forms an important counterexample, as it focuses on determining the phase transition properties of, among others, TSP in random graphs, by observing both the graph connectivity and the standard deviation of the cost matrix. Their conjecture, which has become popular, is that all NP-complete problems have at least one order parameter and that difficult to solve problem instances are clustered around a critical value of this order parameter.

The results presented here are a concise version of the ones presented in (van Hemert, 2005). While here we concentrate more on the success of the evolutionary algorithm, there the focus lies on the analysis of the evolved problem instances in relation to the algorithms.

5.1 Adaptations to the Methodology

A TSP instance is represented by a list of 100 (x, y) coordinates on a 400×400 grid. The list directly forms the chromosome representation with which the evolutionary algorithm works. For each of the thirty initial TSP instances, we create a list of 100 nodes, by uniform randomly selecting (x, y) coordinates on the grid.

Uniform crossover here consists of choosing for each node of the offspring with equal probability either the node from the first or the second parent. Mutation consists of replacing each one of its nodes, with a probability pm, with uniform randomly chosen (x, y) coordinates.

The number of generations until termination is set to 600 generations.

5.2 Problem Solvers

As for other constrained optimisation problems, we distinguish between two types of algorithms, complete algorithms and incomplete algorithms. The first are often based on a form of branch-and-bound, while the latter are equipped with one or several heuristics. In general, as complete algorithms will quickly become useless when the size of the problem is increased, the development of TSP solvers has shifted toward heuristic methods. One of the most renowned heuristic methods is Lin-Kernighan (Lin and Kernighan, 1973). Developed more than thirty years ago, it is still known for its success in efficiently finding near-optimal results.

The core of Lin-Kernighan, and its descending variants, consists of edge exchanges

in a tour. It is precisely this procedure that consumes more than 98% of the algorithm's run-time. Therefore, in order to measure the *search effort* of Lin-Kernighan-based algorithms we count the number of times an edge exchange occurs during a run. Thus, the measure of the time complexity is independent of the hardware, compiler and programming language used. In this study, we use two variants of the Lin-Kernighan algorithm, which are explained next.

Chained Lin-Kernighan (CLK) is a variant (Applegate et al., 2000) that aims to introduce more robustness in the resulting tour by chaining multiple runs of the Lin-Kernighan algorithm. Each run starts with a perturbed version of the final tour of the previous run. The length of the chain depends on the number of nodes in the TSP problem.

In (Papadimitriou, 1992), a proof is given demonstrating that local optimisation algorithms that are PLS-complete (Polynomial Local Search), can always be forced into performing an exponential number of steps with respect to the input size of the problem. In (Johnson and McGeoch, 1997), Lin-Kernighan was first reported to have difficulty on certain problem instances, which had the common property of being clustered. The reported instances consisted of partial graphs and the low performance was induced because the number of "hops" required to move the salesman between two clusters was set large enough to confuse the algorithm. We are using the symmetric TSP problem, where only full graphs exist and thus, every node can be reached from any other in one "hop".

As a reaction on the low performance reported in (Johnson and McGeoch, 1997), a new variant of Lin-Kernighan is proposed in (Neto, 1999), called *Lin-Kernighan with Cluster Compensation* (LK-CC). This variant aims to reduce the computational effort, while maintaining the quality of solutions produced for both clustered and non-clustered instances.

Cluster compensation works by calculating the cluster distance for nodes, which is a quick pre-processing step. The cluster distance between node v and w equals the minimum bottleneck cost of any path between v and w, where the bottleneck cost of a path is defined as the heaviest edge on that path. These values are then used in the guiding utility function of Lin-Kernighan to prune unfruitful regions, i.e., those involved with high bottlenecks, from the search space.

5.3 Experiments and Analysis

Each experiment consists of 190 independent runs with the evolutionary algorithm, each time producing the most difficult problem instance at the end of the run. The set of problem instances from an experiment is called *Algorithm:Evolved set*, where *Algorithm* is either CLK or LK-CC, depending on which problem solver was used in the experiment. We ignore here the series of improved instances.

The total set of problem instances used as the initial populations for the 190 runs is called *Random set*, and it contains $190 \times 30 = 5\,700$ unique problem instances, each of which is generated uniform randomly. This set of initial instances is the same for both Lin-Kernighan variants.

5.3.1 Increase in Difficulty

In Figure 11, we show the amount of search effort required by Chain Lin-Kernighan to solve the sets of TSP instances corresponding to the different experiments, as well as to the Random set. Also, we compared these results to results reported in (van Hemert and Urquhart, 2004), where a specific TSP generator was used to create clustered in-

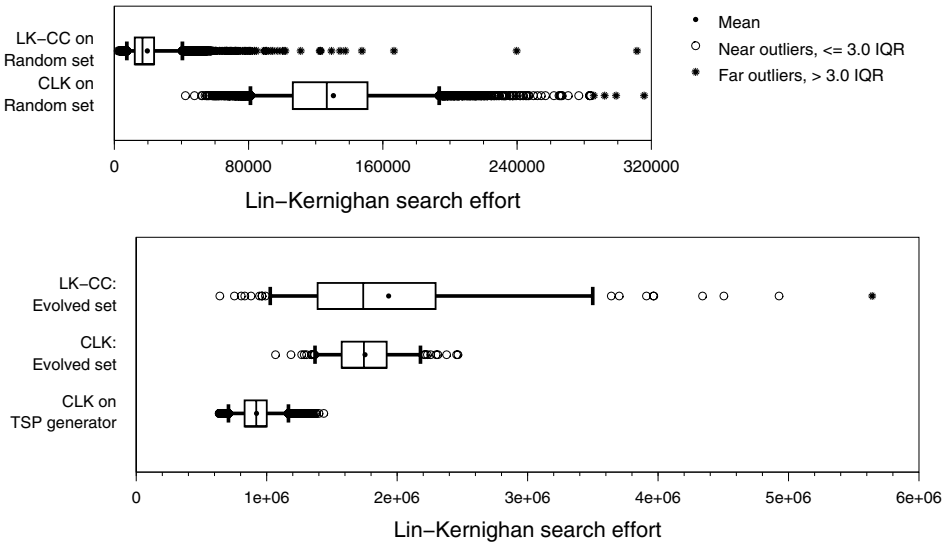


Figure 11: Box-and-whisker plots of the search effort required by CLK and LK-CC on the Random set (top), and CLK on the TSP generator and on the CLK:Evolved set (bottom) and by LK-CC on the LK-CC:Evolved set (bottom).

stances and then solved using the Chained Lin-Kernighan variant. This set contains the 50 most difficult to solve instances from the $50\,000$ produced in those experiments and it is called *TSP generator*.

In Figure 11, we notice that the mean and median difficulty of the instances in the CLK:Evolved set is higher than those created with the TSP generator. Also, as the 5/95 percentile ranges are not overlapping, we have a high confidence of the correctness of the difference in difficulty. This confidence is even higher when comparing with the randomly generated instances, which are shown separate in the upper plot as they require a much smaller scale.

When comparing the difficulty of CLK and LK-CC for both the Random set and the Evolved sets in Figure 11, we find a remarkable difference in the amount of variation in the results of both algorithms. CLK has much more variation with the Random set than LK-CC. However, for the evolved sets, the opposite is true. We also mention that for the Random set, LK-CC is significantly faster than CLK, while difference in speed for the evolved sets is negligible.

5.3.2 Discrepancy with the Optimum

We count the number of times the optimum was found by both algorithms for the Random set and for the corresponding Evolved sets. These optima are calculated using Concorde's (Applegate et al., 1999) branch-and-cut approach to create an LP-representation of the TSP instance, which is then solved using Qsopt (Applegate et al., 2004). We show the average discrepancy between optimal tour length and the length of the tour produced by one of the problem solvers.

For the Random set, CLK has an average discrepancy of 0.004% (stdev: 0.024), and it finds the best tour for 95.8% of the set. For the same set of instances, LK-CC has an average discrepancy of 2.080% (stdev: 1.419), and it finds the best tour for 6.26% of the

set. This is an enormous difference in effectiveness.

A similar picture presents itself for the Evolved sets. Here, CLK has an average discrepancy of 0.030% (stdev: 0.098), and find the best tour for 84.7% of the CLK:Evolved set. LK-CC has an average discrepancy of 2.58% (stdev: 1.666), and finds the best tour for 4.74% of the LK-CC:Evolved set.

We run each variant on the problem instances in the set evolved for the other variant. Table 2 clearly shows that a set evolved for one algorithm is much less difficult for the other algorithm. However, each variant needs significantly more search effort for the alternative Evolved set than for the Random set. This indicates that some properties of difficulty are shared between the algorithms.

Table 2: Mean and standard deviation, in brackets, of the search effort in edge exchanges required by both algorithms on the Random set and both Evolved sets.

	CLK		CC-LK	
CLK:Evolved set	1 753 790	(251 239)	207 822	(155 533)
CC-LK:Evolved set	268 544	(71 796)	1 934 790	(799 544)
Random set	130 539	(34 452)	19 660	(12 944)

5.3.3 Clustering Properties of Problem Sets

To get a quantifiable measure for the amount of clusters in TSP instances we use the clustering algorithm GDBSCAN (Sander et al., 1998). This algorithm uses no stochastic process, assumes no shape of clusters, and works without a predefined number of clusters. This makes it an ideal candidate to cluster 2-dimensional spatial data, as the method suffers the least amount of bias possible. It works by using an arbitrary neighbourhood function, which in this case is the minimum Euclidean distance. It determines clusters based on their density by first seeding clusters and then iteratively collecting objects that adhere to the neighbourhood function. The neighbourhood function here is a spatial index, which results in a run-time complexity of $O(n \log n)$.

Clustering algorithms impose a large responsibility on their users, as every clustering algorithm depends on at least one parameter to help it define what a cluster is. Two common parameters are the number of clusters, e.g., for variants of k-means, and distance measurements to decide when two points are near enough to consider them part of the same cluster. The setting of either of these parameters greatly affects the resulting clustering. To get a more unbiased result on the number of clusters in a set of points we need a more robust method.

To get a more robust result for the number of clusters found in the different sets of TSP instances we repeat the following procedure for each instance in the set. Using the set $\{10, 11, 12, \ldots, 80\}$ of settings for the minimum Euclidean distance parameter for GDBSCAN, we cluster the TSP instance for every parameter setting. We count the number of occurrences of each number of clusters found. Then we average these results over all the TSP instances in the set. The set of minimum Euclidean distance parameters is chosen such that it includes both the peak and the smallest number of clusters for each problem instance.

We use the above procedure to quantify the clustering of problem instances in the Random set and the two evolved sets. Figure 12 shows that for the Random set and the LK-CC:Evolved set, the average number of clusters found does not differ by

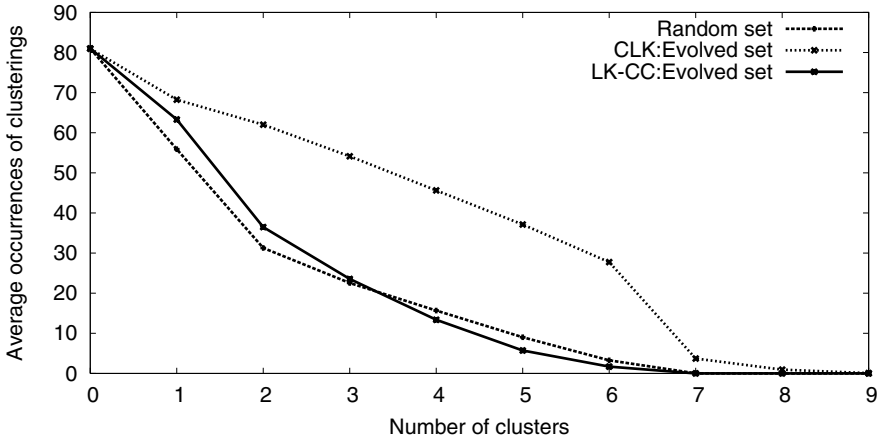


Figure 12: Average amount of clusters found for problem instances of the Random set and for problem instances evolved against CLK and LK-CC.

much. Instead, the problem instances in the CLK:Evolved set contain consistently more clusters. The largest difference is found for 2–6 clusters.

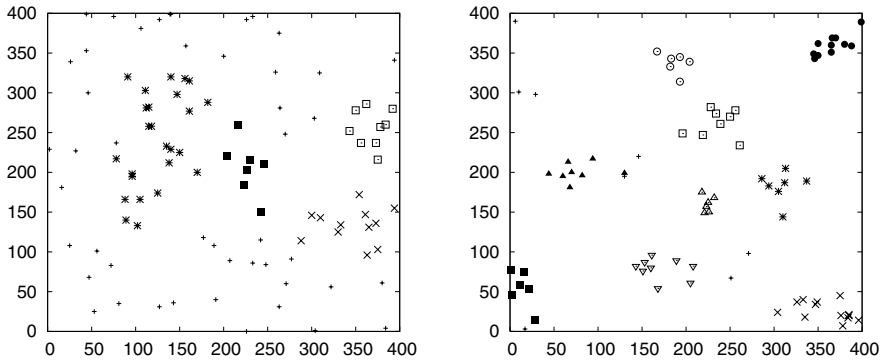
As two-dimensional TSP problems easily allow us to perform a visual inspection, we include in Figure 13 two problem instances. The first is a uniform randomly generated instance, while the second is the best result after 600 generations when evolving against the CLK algorithm. Both instances were clustered using the GDBSCAN algorithm using the same parameters (eps=41, and m=5). We clearly observe both a decrease in noise (46 points become 8 points) and an increase in clusters (5 clusters become 9 clusters).

5.3.4 Distribution of Pair-Wise Distances

In Figure 14, we show the average number of occurrences for distances between pairs of nodes. Every TSP instance contains $\binom{100}{2}$ pairs of nodes on the account that it forms a full graph. The distribution of these pair-wise distances mostly resembles a skewed Gaussian distribution. The main exception consists of the cut-off at short segments lengths. These very short distances, smaller than about 4, occur rarely when 100 nodes are distributed over a 400×400 space.

For the Chained Lin-Kernighan we notice a change in the distribution; compared with the Random set, both the number of short segments and the number of long segments increases. Also, the number of medium length occurrences is less than for the Random set. This forms more evidence for the introduction of clusters in the problem instances. Although this analysis does not provide us with the amount of clusters, it does give us an unbiased view on the existence of clusters, as it is both independent of the TSP algorithms and any clustering algorithm.

Figure 14 shows the distribution of pair-wise distances for problem instances evolved against the LK-CC algorithm. While we notice an increase in shorter distances, this is matched by an equal decrease in longer distances. Thus, the total layout of the nodes becomes more huddled together, although no clear clustering is present.



(a) Uniform randomly generated (b) Evolved against CLK (600 generations)

Figure 13: Two typical examples of the Euclidean TSP problem, both clustered using GDBSCAN (eps=41,m=5). The small crosses represent noise, while other points show which cluster they belong to.

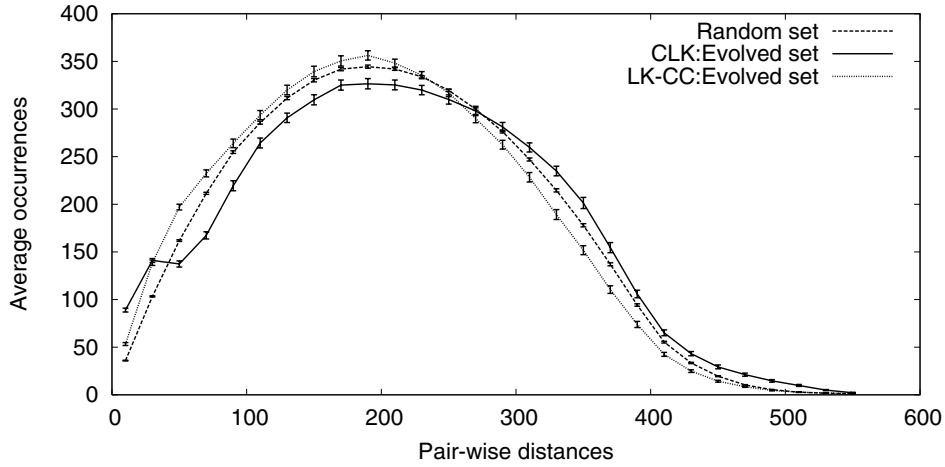


Figure 14: Distribution of distances over all pairs of nodes in randomly generated and evolved problem instances after 600 generations (CLK and LK-CC), 95% confidence intervals included, most of which are small.

5.4 Conclusions on Travelling Salesman

We have applied the methodology to evolve difficult to solve travelling salesman problem instances. The method was used to create a set of problem instances for two well known variants of the Lin-Kernighan heuristic. These sets provided far more difficult problem instances than problem instances generated uniform randomly. Moreover, for the Chained Lin-Kernighan variant, the problem instances are significantly more difficult than those created with a specialised TSP generator. Through analysis of the sets of evolved problem instances we show that these instances adhere to structural properties that directly affect the weak points of the corresponding algorithm.

Problem instances which were difficult for Chained Lin-Kernighan seem to contain clusters. When comparing with the instances of the TSP generator, these contained on average more clusters (10 clusters) than the evolved ones (2–6 clusters). Thus, this leads us to the conjecture that clusters alone are not a sufficient property to induce difficulty for CLK. The position of the clusters and distribution of cities over clusters, as well as the distribution of nodes not belonging to clusters, can have considerable influence.

The goal of the author of LK-CC is to provide a TSP solver where its efficiency and effectiveness are not influenced by the structure of the problem (Neto, 1999). Problem instances evolved in our study, which are difficult to solve for Lin-Kernighan with Cluster Compensation, tend to be condense and contain random layouts. The algorithm suffers from high variation in the amount of search effort required, therefore depending heavily on a lucky setting of the random seed. Furthermore, its effectiveness is much lower than that of CLK, as the length of its tours are on average, further away from the optimum. Thus, it seems that to live up its goals, LK-CC is losing on both performance and robustness.

6 Conclusions

We have provided a general methodology whereby an evolutionary algorithm is used to search for difficult to solve problem instances. Thus, we contribute a general stress testing technique that enables researchers to locate weaknesses in their combinatorial optimisation algorithms. Stress testing is an important part of the process of developing algorithms, as it allows developers to seek out weaknesses that need attention. Of course, the problem instances that arise from this process are most relevant to the algorithm that was under testing, however, as we observed on the travelling salesman problem, it may be the case that such instances prove difficult to solve for other algorithms. As a result, the structural properties extracted may be of use to understand the weakness of those algorithms.

We have shown that this methodology is successful in three major problem domains of combinatorics because it can locate problem instances more difficult than found in popular benchmark suites when keeping the number of basic features, such as the size, the same. Moreover, by analysing the instances obtained this way we try to explain their difficulty in terms of structural properties, thereby exposing weaknesses of the corresponding algorithms. By expressing the structures of these weaknesses developers can try to address them directly. The following summarises the important conclusions and properties for each major domain.

- We can consistently produce binary constraint satisfaction problem instances that turn out to be extremely hard to prove unsolvable. An analysis of these problem instances provides a new structural property that correlates well with the search effort required by two well known search algorithms that are both sound and com-

plete. This property is based on the minimal unsolvable sub-problems of a given unsolvable problem, where we count the number of solutions to the largest solvable sub-problems of these unsolvable sub-problems. A clear trend exists where the number of solutions increases with increasingly more difficult problem instances.

- The solvable problem instances found for the Boolean satisfiability problem are significantly more difficult to solve than those in two existing benchmarks suits. By observing the size of the backbone and the number of solutions we show that these have no influence on the increase in difficulty throughout the run of the evolutionary algorithm. On the other hand, there exists a clear trend in the average standard deviation in variable frequency, which provides a reasonable explanation for the increase in difficulty when paired with findings in a previous study.
- In the domain of travelling salesman we have used the evolved problem instances to show the difference in the structure of pathological cases for two modern variants from the well known Lin-Kernighan heuristic. For one of these variants, this structure consists of partial clustering of the weighted graph. In turn, for the other variant, the degeneration of efficiency and effectiveness is due to small variations, making it highly susceptible to small changes in the problem space.

The results obtained from the analyses in this study may be of interest to the corresponding developers in order to make new improvements, which would help tackle problem instances that exhibit the properties that make the evolved problem instances difficult to solve. They may also be of interest to those using these problem solvers, where the choice might be to steer away from a problem solver if one needs to solve problems that exhibit such properties.

The potential of the methodology as a whole lies in its automated way of locating pathological benchmark cases. In a similar way as shown here, these cases may lead to insights into the algorithm when analysed for common structures. In this study, the technique is used to maximise the search effort of one algorithm. Other tasks and variations for it to perform may include multi-objective goals. In this context, an example is to evolve problem instances that contain a small backbone, that are solvable, and that are difficult to solve. Another interesting area is to evolve problem instances that are difficult for a set of problem solvers. Last, we would like to see this methodology applied to many more problem domains and problem solvers.

Acknowledgments

The author started this study at the National Research Institute for Mathematics and Computer Science (CWI), Amsterdam, NL. He continued it at Napier University and the University of Edinburgh, Edinburgh, UK while financially supported through a TALENT-Stipendium awarded to the author by the Netherlands Organization for Scientific Research (NWO). The author wishes to thank Conor Ryan at the University of Limerick, IE, for making available their computing facilities. The author expresses his gratitude to Mark Collins for the long discussions on the difficulty of Boolean satisfiability. Last, he thanks the reviewers for their detailed and constructive comments.

References

Achlioptas, D., Gomes, C., Kautz, H., and Selman, B. (2000). Generating satisfiable problem instances. In *Proceedings of the Seventeenth National Conference on Artificial*

Intelligence and The Twelfth Annual Conference on Innovative Applications of Artificial Intelligence.

Achlioptas, D., Jia, H., and Moore, C. (2004). Hiding satisfying assignments: Two are better than one. In *AAAI*, pages 131–136. Extended version.

Achlioptas, D., Kirousis, L., Kranakis, E., Krizanc, D., Molloy, M., and Stamatiou, Y. (2001). Random constraint satisfaction: A more accurate picture. *Constraints*, 4(6):329–344.

Applegate, D., Bixby, R., Chvátal, V., and Cook, W. (1999). Finding tours in the TSP. Technical Report 99885, Research Institute for Discrete Mathematics, Universität Bonn.

Applegate, D., Cook, W., Dash, S., and Mevenkamp, M. (2004). Qsopt linear programming solver. http://www.isye.gatech.edu/~wcook/qsopt/.

Applegate, D., Cook, W., and Rohe, A. (2000). Chained lin-kernighan for large travelling salesman problems. http://www.citeseer.com/applegate99chained.html.

Bacchus, F. and van Beek, P. (1998). On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the 15th International Conference on Artificial Intelligence*, pages 311–318. Morgan Kaufmann.

Bäck, T., Fogel, D., and Michalewicz, Z., editors (1997). *Handbook of Evolutionary Computation*. Institute of Physics Publishing Ltd, Bristol and Oxford University Press, New York.

Bayardo, R. (2005). Relsat. Version 2.00 at http://www.almaden.ibm.com/cs/people/bayardo/resources.html.

Bayardo Jr, R. and Pehoushek, J. (2000). Counting models using connected components. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*.

Bayardo Jr, R. and Schrag, R. (1997a). Using csp look-back techniques to solve real world sat instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 203–208.

Bayardo Jr, R. J. and Schrag, R. (1997b). Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208.

Beacham, A. (2000). The complexity of problems without backbones. Master's thesis, University of Alberta.

Boettcher, S., Istrate, G., and Percus, A. (2005). Spines of random constraint satisfaction problems: Definition and impact on computational complexity. In *Eighth International Symposium on Artificial Intelligence and Mathematics*. Extended version.

Cheeseman, P., Kenefsky, B., and Taylor, W. M. (1991). Where the really hard problems are. In *Proceedings of the IJCAI'91*, pages 331–337.

Cohn, A. G., editor (1994). *Proceedings of the European Conference on Artificial Intelligence*. Wiley.

Cook, S. (1971). The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*, pages 151–158.

Cook, S. and Mitchell, D. (1997). Finding hard instances of the satisfiability problem: A survey. In Du, D., Gu, J., and Pardalos, P., editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–17. AMS.

Corno, F., Sanchez, E., Sonza Reorda, M., and Squillero, G. (2004). Automatic test program generation - a case study. *IEEE Design & Test*, 21(2):102–109.

Cotta, C. and Moscato, P. (2003). A mixed evolutionary-statistical analysis of an algorithm's complexity. *Applied Mathematics Letters*, 16:41–47.

Culberson, J. and Gent, I. (1999). Well out of reach: why hard problems are hard. Technical report, APES Research Group.

Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the ACM*, 7(201–215).

Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312.

Du, D., Gu, J., and Pardalos, P., editors (1997). *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS.

Fu, Z. (2004). zChaff. Version 2004.11.15 from http://www.princeton.edu/~chaff/zchaff.html.

Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freedman and Co.

Gent, I., MacIntyre, E., Prosser, P., Smith, B., and Walsh, T. (1996a). An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Principles and Practice of Constraint Programming*, pages 179–193. Springer LNCS 1118.

Gent, I. and Walsh, T. (1996). The satisfiability constraint gap. *Artificial Intelligence*, 81(1-2):59–80.

Gent, I. P., MacIntyre, E., Prosser, P., and Walsh, T. (1996b). The constrainedness of search. In *Proceedings of the AAAI-96*, pages 246–252.

Golomb, S. and Baumert, L. (1965). Backtrack programming. *ACM*, 12(4):516–524.

Haralick, R. and Elliot, G. (1980). Increasing tree search efficiency for constraint-satisfaction problems. *Artificial Intelligence*, 14(3):263–313.

Hayes, B. (1997). Can't get no satisfaction. *American Scientist*, 85(2):108–112.

Johnson, D. and McGeoch, L. (1997). The traveling salesman problem: a case study. In Aarts, E. and Lenstra, J., editors, *Local Search in Combinatorial Optimization*, chapter 8, pages 215–310. John Wiley & Sons, Inc.

Kautz, H. and Selman, B. (1992). Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363. John Wiley & Sons, Inc.

Kilby, P., Slaney, J., Thiébaux, S., and Walsh, T. (2005). Backbones and backdoors in satisfiability. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pp 1368–1373.

Kratica, J., Ljubić, I., and Tošic, D. (2003). A genetic algorithm for the index selection problem. In et al., G. R., editor, *Applications of Evolutionary Computation*, volume 2611 of *LNCS*, pages 281–291. Springer-Verlag.

Kwan, A., Tsang, E., and Borrett, J. (1998). Predicting phase transitions of binary constraint satisfaction problems with constraint graph information. *Intelligent Data Analysis*, 268(1–4):45–62.

Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G., and Shmoys, D. B. (1985). *The Traveling Salesman Problem*. John Wiley & Sons, Chichester.

Le Berre, D. and Simon, L. (2005). SAT competitions. http;//www.satcompetition.org.

Lenstra, J. and Rinnooy Kan, A. (1981). Complexity of vehicle routing and scheduling problems. *Networks*, 11:221–227.

Lin, S. and Kernighan, B. (1973). An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516.

Mammen, D. and Hogg, T. (1997). A new look at the easy-hard-easy pattern of combinatorial search difficulty. *Journal of Artificial Intelligence Research*, 7:47–66.

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E. (1953). Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, pages 1087–1091.

Mitchell, D., Selman, B., and Levesque, H. (1992). Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465.

Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., and Troyansky, L. (1999). Determining computational complexity from characteristic phase transitions. *Nature*, 400:133–137.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535.

Neto, D. (1999). *Efficient Cluster Compensation for Lin-Kernighan Heuristics*. PhD thesis, Computer Science, University of Toronto.

Palmer, E. M. (1985). *Graphical Evolution*. John-Wiley & Sons, New York.

Papadimitriou, C. (1992). The complexity of the Lin-Kernighan heuristic for the traveling salesman problem. *SIAM Journal of Computing*, 21(3):450–465.

Papadimitriou, C. (1994). *Computational Complexity*. Addison-Wesley.

Papadimitriou, C. and Steiglitz, K. (1976). Some complexity results for the traveling salesman problem. In *STOC '76: Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM Press.

Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299.

Prosser, P. (1994). Binary constraint satisfaction problems: Some are harder than others. In (Cohn, 1994), pages 95–99.

Prosser, P. (1996). An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109.

Rossi, F., Petrie, C., and Dhar, V. (1990). On the equivalence of constraint satisfaction problems. In Aiello, L. C., editor, *ECAI'90: Proceedings of the 9th European Conference on Artificial Intelligence*, pages 550–556, Stockholm. Pitman.

Rudnick, E., Patel, J., Greenstein, G., and Niermann, T. (1994). Sequential circuit test generation in a genetic algorithm framework. In *DAC '94: Proceedings of the 31st Annual Conference on Design Automation*, pages 698–704. ACM Press.

Saab, D., Saab, U., and Abraham, J. (1992). CRIS: a test cultivation program for sequential VLSI circuits. In *Proc. 1992 IEEE/ACM International Conference on Computer-Aided Design*, pages 216–219.

Sander, J., Ester, M., Kriegel, H.-P., and Xu, X. (1998). Density-based clustering in spatial databases: The algorithm GDBSCAN and its applications. *Data Mining and Knowledge Discovery*, 2(2):169–194.

Selman, B., Mitchell, D., and Levesque, H. (1996). Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29.

Singer, J., Gent, I., and Smaill, A. (2000). Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research*, 12:235–270.

Slaney, J. and Walsh, T. (2001). Backbones in optimization and approximation. In Nebel, B., editor, *Proceedings of the Seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, pages 254–259, San Francisco, CA. Morgan Kaufmann Publishers, Inc.

Smith, B. (1994). Phase transition and the mushy region in constraint satisfaction problems. In (Cohn, 1994), pages 100–104.

Smith, B. M. and Dyer, M. E. (1996). Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1–2):155–181.

Smith, B. M. and Grant, S. A. (1997). Modelling exceptionally hard constraint satisfaction problems. In Smolka, G., editor, *Principles and Practice of Constraint Programming (CP97)*, pages 182–195. Springer-Verlag.

Srinivas, M. and Patnaik, L. (1993). A simulation-based test generation scheme using genetic algorithms. In *Proceedings International Conference VLSI Design*, pages 132–135.

Tsang, E. (1993). *Foundations of Constraint Satisfaction*. Academic Press.

van Beek, P. (2005). A 'C' library of routines for solving binary constraint satisfaction problems. Available from http://ai.uwaterloo.ca/~vanbeek/software/software.html.

van Hemert, J. (2003a). Evolving binary constraint satisfaction problem instances that are difficult to solve. In *Proceedings of the IEEE 2003 Congress on Evolutionary Computation*, pages 1267–1273. IEEE Press.

van Hemert, J. (2003b). RandomCSP. Version 1.8.0 freely available from http://freshmeat.net/projects/randomcsp/.

van Hemert, J. (2005). Property analysis of symmetric travelling salesman problem instances acquired through evolution. In Raidl, G. and Gottlieb, J., editors, *Evolutionary Computation in Combinatorial Optimization*, Springer Lecture Notes on Computer Science, pages 122–131. Springer-Verlag, Berlin.

van Hemert, J. and Urquhart, N. (2004). Phase transition properties of clustered travelling salesman problem instances generated with evolutionary computation. In Yao, X., Burke, E., Lozano, J. A., Smith, J., Merelo-Guervós, J. J., Bullinaria, J. A., Rowe, J., Kabán, P. T. A., and Schwefel, H.-P., editors, *Parallel Problem Solving from Nature (PPSN VIII)*, volume 3242 of *LNCS*, pages 150–159, Birmingham, UK. Springer-Verlag.

Walsh, T. (2000). SAT *v* CSP. In *Proceedings of the 6th Conference on Principles and Practice of Constraint Programming (CP 2000)*, pages 441–456.

Williams, C. and Hogg, T. (1994). Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70:73–117.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

Zhang, H. (1997). SATO: An efficient propositional prover. In *CADE-14: Proceedings of the 14th International Conference on Automated Deduction*, pages 272–275. Springer-Verlag.