65. Evolutionary Computation and Constraint Satisfaction

Jano I. van Hemert

In this chapter we will focus on the combination of *evolutionary computation* (EC) techniques and *constraint satisfaction problems* (CSPs). *Constraint programming* (CP) is another approach to deal with constraint satisfaction problems. In fact, it is an important prelude to the work covered here as it advocates itself as an alternative approach to programming [65.1]. The first step is to formulate a problem as a CSP such that techniques from CP, EC, combinations of the two, often referred to as hybrids [65.2, 3], or other approaches can be deployed to solve the problem. The formulation of a problem has an impact on its complexity in terms of effort required to either find a solution or that proof no solution exists. It is, therefore, vital to spend time on getting this right.

CP defines search as iterative steps over a search tree where nodes are partial solutions to the problem where not all variables are assigned values. The search then maintains a partial solution that satisfies all variables with assigned values. Instead, in EC algorithms sample a space of candidate solutions where for each sample point variables are all assigned values. None of these candidate solutions will satisfy all constraints in the problem until a solution is found. Such algorithms are often classified as Davis–Putnam–Logemann–Loveland (DPLL) algorithms, after the first backtracking algorithm for solving CSP [65.4].

Another major difference is that many constraint solvers from CP are sound, whereas EC solvers are not. A solver is sound if it always finds a solution if it exists. Furthermore, most constraint solvers from CP can easily be made complete, although this is often not a desired property for a constraint solver. A constraint solver is complete if it can find every solution to a problem.

65.1 Informal Introduction to CSP

For a formal definition please skip to the next section. A constraint satisfaction problem consists of a set of variables and each variable must be assigned one value from its finite set of values, called its domain. A set of constraints restricts certain simultaneous assignments. In most CSPs, the objective is to search for a simultaneous assignment of all the variables such that all constraints are satisfied, i. e., no forbidden si-

multaneous assignment from the set of constraints is used.

A famous example is the SEND MORE MONEY puzzle, where each letter must be replaced by a unique number such that the following sum holds [65.5]

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ = & M & O & N & E & Y. \end{array}$$

In this CSP, the variables are S, E, N, D, M, O, R, Y and the domains are $\{1, \ldots, 9\}$ for S, M and $\{0, \ldots, 9\}$ for E, N, D, O, R, Y. The constraint can be also written as $1000 \times S + 100 \times E + 10 \times N + D + 1000 \times M + 100 \times O + 10 \times R + E = 10\,000 \times M + 1000 \times O + 100 \times N + 10 \times E + Y$. Every CSP A can be rewritten into an another CSP B where a bijective mapping exists between the solutions of A and B, which follows from the reducibility theorem from complexity theory [65.6]. The solution to this CSP is the assignment $S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2$, which uniquely satisfies the constraint.

Other very well-known constraint satisfaction problems are map coloring, more commonly known as vertex coloring (Sect. 65.5.2), and the recreational game Sudoku, which is equivalent to completing a graph 9-coloring problem on a given specific graph with 81 vertices. A specific EC solution is provided by *Lewis* [65.7]. Quite a lot of constraint satisfaction problems exist; we will first look at CSP in general within the context of EC as problem solvers. Then we will discuss several specific constraint satisfaction problems and the particular EC approaches applied to these problems. Last, we will provide a brief overview on using EC for generating problem instances for CSP.

65.2 Formal Definitions

Slightly different, but equivalent, formal definitions of CSP exist. The most common definition is:

Definition 65.1 (Constraint Satisfaction Problem)
is a triple $\langle V, D, C \rangle$:

- V is an n-tuple of *variables* $V = \langle v_1, v_2, \ldots, v_n \rangle$,
- Each $v \in V$ has a corresponding m-tuple of values called its *domains*, $D_v = \langle d_1, d_2, \ldots, d_m \rangle$ of which it can be assigned one and
- $C = \langle C_1, \ldots, C_t \rangle$ is a t-tuple of *constraints* where each $c \in C$ *restricts* certain simultaneous variable assignments to occur.

The definition of a constraint is often reversed in the literature, where generic CSP is discussed in that constraints are defined as the set of assignments that are *allowed* rather than restricted. Note, in generic CSP literature, variables are often denoted with X, whereas in graph-oriented problem domains such as graph coloring and maximum clique, V is adopted.

Definition 65.2 (Solution to a CSP)
is an assignment of variables $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ such that for every constraint $c \in C$ on $x_{i_1}, \ldots, x_{i_m}$: $(d_{i_1}, \ldots, d_{i_m}) \notin c$.

In the context of one constraint c, we say an assignment of variables *satisfies* the constraint c if the assignment is in c or *violates* the constraint c if the assignment is *not* in c. A CSP can be *insoluble* – more commonly written as *insolvable*, which means every assignment of variables will violate at least one constraint.

A *constraint solver* is an algorithm that takes as input a CSP and produces as output either a solution or a proof that no solution exists or a notification of failure. The input is often referred to as a *problem instance*, as a CSP is often defined to cover a class of problems such as, 3-satisfiability. The output can be more than one solution, in fact it could be every solution. However, as EC techniques are based on sampling, in principle they cannot proof that every solution has been found, which is referred to as not complete. Moreover, they cannot proof no solution exists, which is referred to as not sound. Therefore, constraint solvers based on EC and other heuristic approaches often terminate after a certain criterion is met, e.g., a predefined elapsed time is reached in terms of the number of solutions evaluated, the computation time spent, or a certain convergence of the population reached.

We recommend the following books for further reading on constraint satisfaction. For the foundations of the problem and basic algorithms, *Tsang* [65.8]; for an introduction with comprehensive overview of constraint programming techniques, *Dechter* [65.9] and *Lecoutre* [65.10]; and for a more theoretical approach *Apt* [65.1] and *Chen* [65.11].

65.3 Solving CSP with Evolutionary Algorithms

In this chapter we will restrict ourself to covering the conceptual mapping required to solve a CSP with an evolutionary algorithm. This mapping will consist of choosing a representation for the problem and a corresponding fitness function to determine the quality of a solution. Once this mapping is complete, the evolutionary algorithm will require other components, such as appropriate variation operators, selection mechanisms, and a suitable initialization method for the population and termination criteria. All these, and other optional variants can be found elsewhere in the handbook.

We will explain the two most common mappings using the well-known n-queens on an $n \times n$-chessboard problem. These mappings are *direct encoding* and *indirect encoding*. First we introduce a conceptual definition of the problem.

The n-queens problem requires the placing of n queens on an $n \times n$ chessboard such that no queen attacks any of the other $n-1$ queens. Thus, a solution requires that no two queens share the same row, column, or diagonal. Several common formal definitions of the problem exist. The most common is to define n variables $\{q_1, \ldots, q_n\}$, where each variable q has a domain that consists of the row position the queen will be placed on in its corresponding unique column, i. e., $q \in \{1, \ldots, n\} \forall i = 1, \ldots, n$. The set of constraints consists of $q_i \neq q_j$ (i. e., not in the same row) and $|q_i - q_j| \neq |i - j| \forall i, j = 1, \ldots, n$ (i. e., not in the same diagonal).

The n-queens problem is no longer considered a challenging problem as it has a structure that can be exploited to solve very large problems of over 9 million queens by repeating a pattern [65.12]. It is, however, an excellent problem for explaining characteristics of constraint satisfaction problems and their solvers due to the simple 2-D spatial nature of the problem. For instance, to explain symmetry in CSP, the 8-queens problem can be used to show it has 12 unique solutions, as shown in Fig. 65.1 out of the 92 distinct solutions when removing variants due to rotational and reflection symmetry.

65.3.1 Direct Encoding

With a direct encoding the genotype consists of a vector $\vec{g}$ where each element corresponds uniquely to one variable of the CSP; an element g_i contains values directly from the domain of its corresponding variable D_i. A wide variety of genetic operators both for mutation and recombination are applicable to this encoding and can be found in [65.13]. Most of these operators will be called discrete or mixed-integer operations.

The genotype is mapped to the phenotype by taking into consideration the constraints; it requires a measurement for determining the quality of candidate solutions. Thus, we need to introduce a fitness function. The most common fitness function takes the sum of all constraints violated by a candidate solution

$$\text{fitness}(\vec{g}) = \sum_{c \in C} \text{violated}(c) \,,$$

$$\text{where violated}(c) = \begin{cases} 1 & \text{if } c \text{ violated by } \vec{g} \\ 0 & \text{if } c \text{ satisfied by } \vec{g} \end{cases} .$$

The fitness should be minimized and once it reaches zero, a solution has been found.

65.3.2 Indirect Encoding

With an indirect encoding the genotype first needs to be transformed into a full or partial assignment of the variables of the CSP. It is also referred to as local search depending on the level of sophistication; these transformations range from as simple as a greedy assignment all the way to sound search algorithms evaluating a small part of the CSP.

The most common approach for this representation takes as a genotype the permutation of variables of the CSP. Many genetic operators are designed to maintain a permutation and several are explained in the Handbook of Evolutionary Computation [65.13]. The permutation is the input to the local search and determines the order in which variables are processed; processing a variable involves trying to assign a value such that no constraint is violated and perhaps further steps if no value can be assigned without violating at least one constraint.

More advanced encodings may also include the ordering in which to consider values from each variable's domain. From constraint programming we know that the order in which variables and values are considered has a huge impact on the efficiency of search algorithms [65.14]; more often it is the search method that determines the order using a particular heuristics such as choosing the next vertex with the *maximum saturation degree*, as is used in DSatur [65.15]. The saturation degree for a vertex is defined as the total number of colors used for coloring its neighbors. The principle

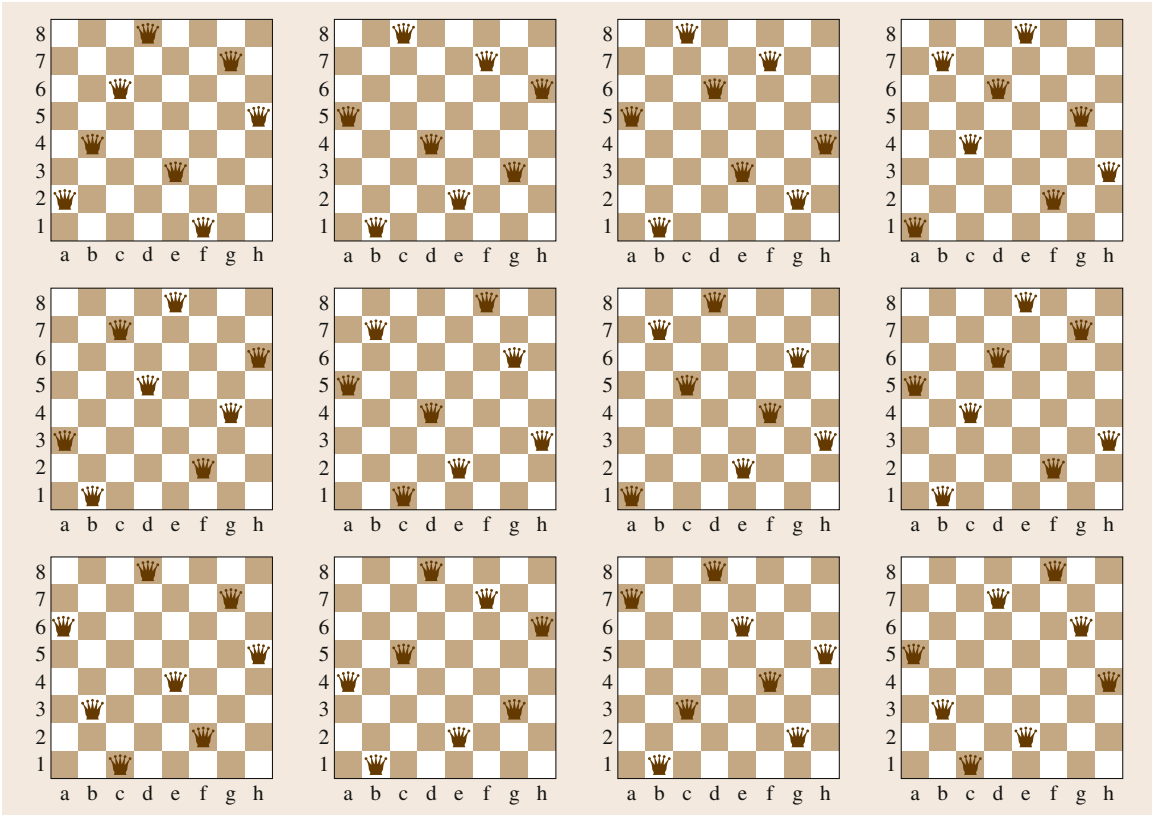

Fig. 65.1 The 12 unique solutions under symmetry via rotations and reflections for the 8-queens problem

has been used in many algorithms since its introduction in 1979.

The most common fitness function used with indirect encoding simply counts the number of unassigned variables after the local search terminates. Note that two different strategies will influence the resolution of this function. If the local search terminates after it first encounters a variable it cannot assign, then many candidate solutions will have the same fitness but can still be very different. On the other hand, terminating after all variables have been considered will give a richer landscape to consider but may incur more computational effort. See [65.16] for a comprehensive theoretical and empirical analysis of sampling in EC.

65.3.3 General Techniques to Improve Performance

Over the past two decades, many techniques were developed to improve the efficiency and/or the effectiveness of EC for solving constraint satisfaction problems. Only a handful of these techniques were evaluated on more than one problem. Hence, we cannot draw any general conclusions about the success of these techniques. Even worse is that many studies will show improvement only compared to their previous results or compare their results with an algorithm that has already been superseded in terms of performance by many other techniques. Often the set of competitor algorithms is chosen to fall within EC, which severely limits the strength of the competition. Therefore, we will discuss techniques for improving performance in the context of the problems they were developed for. Section 65.5 reviews several popular CSPs used for developing more efficient and effective evolutionary algorithms.

One approach that has been applied to several CSPs with varying success is that of assigning weights to constraints to allow biasing the search towards satisfying certain constraints; in the first experiments this approach was referred to as penalty functions [65.17]. Moreover, the search can be influenced dynamically

by adapting weights according to heuristics, such as increasing the weight of the constraint that has been satisfied the least number of times recently [65.18]. The origin of this idea can be found in the self-adaptation used in evolution strategies [65.19].

With penalty functions, the optimization objectives replacing the constraints are traditionally viewed as penalties for constraint violation, hence to be minimized [65.20]. There are two basic types of penalties:

1. Penalty for violated constraints
2. Penalty for wrongly instantiated variables.

Formally, let us assume that we have constraints c_i $(i = \{1, \ldots, m\})$ and variables v_j $(j = \{1, \ldots, n\})$. Let C^j be the set of constraints involving variable v_j. Then the penalties relative to the two options described above can be expressed as follows:

1. $f_1(s) = \sum_{i=1}^{m} w_i \times \chi(s, c_i)$, where

$$\chi(s, c_i) = \begin{cases} 1 & \text{if } s \text{ violates } c_i \\ 0 & \text{otherwise} \end{cases} ,$$

2. $f_2(s) = \sum_{j=1}^{n} w_j \times \chi(s, C^j)$, where

$$\chi(s, C^j) = \begin{cases} 1 & \text{if } s \text{ violates at least one } c \in C^j \\ 0 & \text{otherwise} , \end{cases}$$

where the w_i and w_j are weights that correspond to a constraint and a variable, respectively. These will be important later on, for now we assume all these weights equal to 1.

Obviously, for each of the above functions $f \in \{f_1, f_2\}$ and for each $s \in S$ we have that $\phi(s) = true$ if and only if $f(s) = 0$. For instance, in the graph 3-coloring problem the vertices of a given graph $G = (V, E)$, $E \subseteq V \times V$, have to be colored by three colors in such a way that no neighboring vertices, i. e., graph nodes connected by an edge, have the same color. This problem can be formalized by means of a CSP with $n = |V|$ variables, each with the same domain $D = \{1, 2, 3\}$. Furthermore, we have $m = |E|$ constraints, one for each edge $e = (k, l) \in E$, with $c_e(s) = true$ if and only if $s_k \neq s_l$. Then the corresponding CSP is $\langle S, \phi \rangle$, where $S = D^n$ and $\phi(s) = \bigwedge_{e \in E} c_e$. Using the constraint-oriented penalty function f_1 with $w_i = 1$ for all $i = \{1, \ldots, m\}$ we count the incorrect edges that connect two vertices with the same color. The variable-oriented penalty function f_2 with $w_i = 1$ for all $i = \{1, \ldots, m\}$ amounts to counting the incorrect vertices that have a neighbor with the same color.

Advantages of indirect encoding:

- Introduces in general, e.g., f_1, f_2 are problem-independent penalty functions
- Reduces problem to *simple* optimization
- Allows user preferences by weights.

Disadvantages of indirect encoding:

- Loss of information by packing everything in a single number
- In the case of constrained optimization (as opposed to CSP as we are handling here) f_1, f_2 are reported to be weak [65.21].

65.4 Performance Indicators

An understanding of the efficiency and effectiveness is vital when choosing which solver to use or when developing an algorithm to deal with a specific CSP. In this section we briefly explain measures for determining these properties in the context of solving CSP. However, these properties must be measured using a suite of benchmark instances and, as EAs are generally randomized algorithms, with multiple independent runs of the algorithm on each instance. Choosing an appropriate suite of benchmark instances is paramount to making decisions on which algorithm, parameter setting, or next algorithmic feature to add.

In a sense, the search for a good algorithm is in itself an optimization problem. The suite of benchmark instances represents only the problem, just like training data in a machine learning problem represents all data possibly encountered. Changing an algorithm and tuning its parameters on the same small suite of instances could lead to over-fitting [65.22, 23], which in turn means the algorithm will have a poorer performance in the general case. Therefore, the first step should be to characterize the problem well and have a good representation, e.g., spread, of the instances possibly encountered when deployed.

65.4.1 Efficiency

The time taken by an algorithm to provide a solution is an important factor. Even more so in situations where solutions are required in real time. Much research is devoted to speeding up algorithms, either by cleverly exploiting properties of the problem, by parallelization, or by balancing aspects of the quality of the solution.

The most common approach to measuring the efficiency of evolutionary algorithms is by counting the number of evaluations, i. e., the number of times the fitness function is executed. This approach has several drawbacks. First, the approach allows comparison only with algorithms that use the exact same fitness function and spend the most significant part of their time on computing that function. Second, the computational complexity of the evolutionary algorithm may not be dependent on the fitness function. For instance, with the indirect encoding described in Sect. 65.3.2, much computational effort will go into the local search, whereas the computation of the fitness is trivial.

Another common approach is to measure time spent as reported by the operating system. This has even more drawbacks as the reported numbers will depend on the computer programming language used for implementing the algorithm, the compiler and its setting for translating the implementation into machine code, the architecture of the computer for executing the machine code, and the operating system for hosting the execution environment. Variations of these will have an affect on the reported results and, moreover, as these environments themselves change over time, future studies will find it hard to reproduce results accurately or even create meaningful comparisons to reported results.

A more meaningful solution is to count all the atomic operations that are directly related to the problem. The operations that must be included should be those that in theory increase exponentially in numbers with larger problems, as CSP fall under the class of nonpolynomial deterministic problems. The most common operation will be a *conflict check*; this is also referred to as a constraint check, but in the strictest sense, a constraint check consists of multiple conflict checks [65.8]. For example, when solving the n-queens problem, every time the algorithm checks $q_i \neq q_j$ for any q_i and q_j, this should be recorded as one check. The same procedure should be followed for the constraint concerning diagonal attacks $|q_i - q_j| \neq |i - j|$. The sum of all checks when the algorithm terminates is the computational effort spent.

By reporting the number of conflict checks we assure future studies can compare with current results as this measurement will not be affected by future changes in hardware and software environments. We are measuring a property of the algorithm here as opposed to a property of one implementation of the algorithm running in one particular environment.

It is important to note that there are subtle differences in the reporting used in different studies. Some studies report the average number of operations over all independent runs, including runs that are unsuccessful, i. e., where no solution was found. Other studies report the average number of operations to a solution, where only the runs that yield a solution are taken into account. The former method will produce higher averages than the latter if the success rate is less than 1.

65.4.2 Effectiveness

Efficiency is only one aspect of which to measure the success of a constraint solver. The other most important aspect is that of effectiveness, which measures how successful an algorithm is in finding or approximating a solution. The easiest and most commonly used measurement is that of the *success rate*, which is defined for an experiment as the number of runs in which an algorithm finds a solution divided by the total of number of runs of the same algorithm in that experiment. As no prior knowledge is required about whether problem instances are insolvable, this measurement is straightforward to implement.

Another popular measurement in combinatorial optimization is *distance to the optimal solution*. This measurement poses two challenges in the context of constraint satisfaction. Unlike a combinatorial optimization problem, which has the function to optimize, a CSP has no such function. As an alternative we could use the fitness function, but that is not an inherent property of the problem. Also, we often do not know whether a CSP has a solution and when it does not, then we do not know the optimal fitness function. Distance to the optimal solution is rarely used when solving CSP due to these impracticalities.

65.5 Specific Constraint Satisfaction Problems

Many specific constraint satisfaction problems have been addressed in the literature. A full overview of these would not provide much benefit, as the most likely scenario is that one is looking for papers that provide descriptions of algorithms and results with those algorithms on a certain problem. The exceptions to this are several problems that in the literature are used to drive the development of algorithms in terms of efficiency and effectiveness. These *core* problems are used over and over to test whether new algorithms are better than existing algorithms.

Several reasons exist for the choice of these problems. Their compact definition means that the problem is easy to replicate by everyone and quick to introduce in papers. The most popular problems were used in the 1970s when the theory on non-polynomial deterministic problems was developed, which were consequently seen as important intelligent building blocks. Also, test sets and later problem generators were released in the public domain, thereby providing easy access to test suites.

We will use several of these *core* problems to describe the progress of development in evolutionary computation for constraint satisfaction problems. For each problem we will provide a quick introduction, a justification of its importance in terms of practical applications, and a set of pointers to problem suites before describing the approaches used.

65.5.1 Boolean Satisfiability Problem

Given a Boolean formula ϕ determine whether an assignment of the variables in ϕ exists that makes it `TRUE`. It is often referred to as satisfiability and abbreviated to SAT [65.24]. In SAT variables are often referred to as literals. Most often the problem is studied in conjunctive normal form (CNF) where ϕ is a conjunction of clauses where each clause is a disjunction of variables. Every SAT problem can be reduced to a 3-CNF-SAT (three variables/clause-conjunctive normal form-satisfiability) [65.25], where each clause has three literals.

3-CNF-SAT was the first problem to be shown to be NP-complete [65.26]. It serves as an important basis to proving that other problems are NP-complete, such as the maximal clique problem. Such a proof involves a polynomial-time reduction from 3-CNF-SAT to the other problem [65.6].

The following is an example of 3-CNF-SAT:

- $\phi = (x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_1 \vee \neg x_6) \wedge (x_3 \vee x_2 \vee \neg x_5)$
- A solution: $x_1 = 1$, $x_2 = 0$, $x_3 = 1$, $x_4 = 0$, $x_5 = 0$, $x_6 = 0$.

Important practical applications of SAT are model checking [65.27], for example, in mathematical proof planning [65.28], generic planning problems, especially using the planning domain definition language (PDDL) [65.29], test pattern generation [65.30], and haplotyping in the scientific field of bioinformatics [65.31].

As far as the development of efficient and effective CSP solvers go, SAT is the most active field. It has an annual conference – The International Conference on Theory and Applications of Satisfiability Testing, which also hosts an annual competition to determine the current best solvers. The latter also ensures that new problem instances are continuously added, which prevents what is called *overfitting* [65.32] of the solvers to an existing set of problem instances.

The general approach to solve satisfiability with EC is to directly represent the variables in ϕ and assign these either `TRUE` or `FALSE`, i. e., these form the domain. The fitness function used is the number of clauses violated, which should be minimized.

The earliest evolutionary algorithm for SAT was reported in 1994 by [65.33] and was soon followed by the work of *Gottlieb* and *Voss* [65.34, 35], who were looking to improve its performance. Soon after, independent efforts led to parallelized algorithms [65.36, 37]. In 2000, the first adaptive evolutionary algorithms were applied [65.38], which was 3 years after they were applied to graph coloring (Sect. 65.5.2).

The introduction of hybrid evolutionary algorithms with local search created a real boost of research activity [65.39–43]. However, a major issue remains with research on solving satisfiability with EC, as all studies include only local search and evolutionary algorithms without comparing to the state-of-art DPLL and heuristic solvers from the annual satisfiability community. This holds true even for recent studies such as [65.44]. Due to this major gap between the two communities of EC and CP, we do not comment on the comparison in terms of effectiveness and efficiency.

New research [65.45] focusses on using EC to evolve parameter settings for existing sound SAT

solvers, mostly ones based on the Davis–Putnam–Logemann–Loveland algorithm [65.46]. All modern SAT solvers have many parameters to tune how the search is organized. These parameters are often tuned manually, which allows for only a small exploration. Using EC, a much larger space can be explored in order to create fast SAT solvers for a given benchmark.

65.5.2 Graph Coloring

Graph coloring has several variants. The most commonly used definition is that of graph k-coloring, also known as the vertex coloring problem. Given a graph of vertices and edges $\langle V, E\rangle$ the goal is to find a coloring of the vertices V of the graph such that no two adjacent vertices have the same coloring. If $c(v)$ provides the color assigned to v, then $\forall v, w \in V : c(v) \neq c(w)$ *iff* $(v, w) \in E$. The objective is to make use of k or less colors. The problem is known to be NP-complete for $k \geq 3$ and to be decidable in linear time for $k \leq 2$.

Graph coloring is an abstract problem that lies at the core of many applications. Well-known applications are scheduling, most specifically timetabling [65.47], register allocation in compilers [65.48], and frequency assignment in wireless communication [65.49]. It is a well-studied problem as is shown by the number of entries in the best-kept bibliography source until April 2010 with over 450 publications contributing to vertex coloring [65.50].

The Second DIMACS Implementation Challenge in 1992–1993 focused on maximum clique, graph coloring, and satisfiability. The challenge provided not only a standard format for graph k-coloring problem instances, but also provided a set of problem instances that is still popular today. Soon after, in 1994, *Culberson* and *Luo* [65.51] created a problem instance generator, which can create problem instances with a known k and various other properties. Several other generators exist with specific goals, such as to hide cliques [65.52], to create register-interference graphs [65.53], and to create timetabling problems (Sect. 65.5.4).

The most straightforward approach to solving graph k-coloring with EC is to represent a genome as a vector of all variables of the problem. This vector can then undergo genetic operators suitable for integer representations. The fitness function is simply the number of violated constraints, which should be minimized until a solution is found when the fitness is equal to zero. Unfortunately, this approach leads to algorithms that are inefficient and ineffective [65.54].

To make EC more efficient and effective for solving graph k-coloring, new algorithms have been developed; these broadly fall into two categories. The first category consists of adding mechanisms that prevent the stagnation of search due to premature convergence. The second category consists of alternative representations that make use of decoders to map genotypes to phenotypes. The two categories are not mutually exclusive, and studies have included algorithms that combine mechanisms from both categories.

The earliest work on solving graph k-coloring with EC includes the following. *Fleurent* and *Ferland* successfully considered various hybrid evolutionary algorithms [65.55] with Tabu search and extended their work into a general implementation of heuristic search methods in [65.56]. *Von Laszewski* looked at structured operators and used adaption to improve the convergence rate of a genetic algorithm [65.57]. Davis designed an algorithm [65.58] to maximize the total of weights of nodes in a graph colored with a fixed number of colors. *Coll* et al. [65.59] discussed graph coloring and crossover operators in a more general context.

Juhos and *van Hemert* introduced several heuristics [65.60, 61] for guiding the search of an evolutionary algorithm. All these heuristics depend on their novel representation that collapses the graph by combining nodes assigned with the same color into one hypernode, which speeds up further constraint checking as edges are merged into hyperedges [65.62]. This representation benefits both complete and heuristic methods.

Moreover, as shown in the results in Fig. 65.2, the evolutionary algorithms developed by Juhos and van Hemert are able to outperform a complete method (Backtracking-DSatur) on very difficult problem instances where the chromatic number is 10 or 20. These algorithms are unable to compete with the complete method for smaller chromatic numbers of 3 and 5.

65.5.3 Binary Constraint Satisfaction Problems

A *binary constraint satisfaction problem* (BINCSP) is a CSP where every constraint $c \in C$ restricts at most two variables [65.63]. Often, network graphs are used to visualize (CSP) instances. In Fig. 65.3, we provide an example of a restricting hypergraph of a BINCSP. It consists of three variables $V = \{v_1, v_2, v_3\}$, all of which have domain $D = \{a, b\}$. In a hypergraph every vertex corresponds to a possible variable assignment, i. e., $\langle v, d\rangle$, where $v \in V$ and $d \in D_v$. Every edge indicates the variable assignments that are forbidden by the

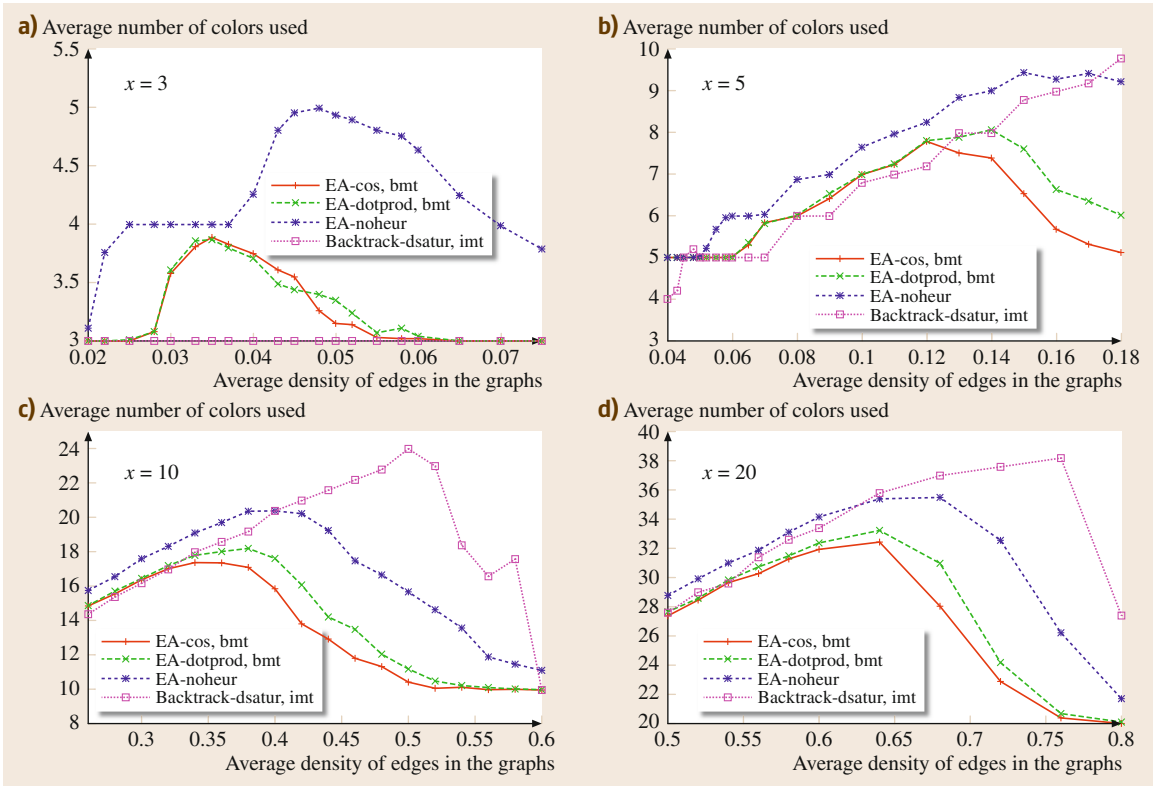

Fig. 65.2a–d Results of several evolutionary algorithms against the complete method Backtracking-DSatur; average minimum number of colors used through the phase transition

set of constraints C. In the example, we show all the edges that correspond to the following set of forbidden value pairs $C = \{ \{\langle v_1, a\rangle, \langle v_2, a\rangle\}, \{\langle v_1, a\rangle, \langle v_3, b\rangle\}, \{\langle v_1, b\rangle, \langle v_2, a\rangle\}, \{\langle v_1, b\rangle, \langle v_2, b\rangle\}, \{\langle v_1, b\rangle, \langle v_3, a\rangle\}, \{\langle v_1, b\rangle, \langle v_3, b\rangle\}, \{\langle v_2, a\rangle, \langle v_3, a\rangle\}, \{\langle v_2, a\rangle, \langle v_3, b\rangle\} \}$.

For problem instances, studies on BINCSP generally create large sets of instances using one of many problem instance generators. Several models to randomly create BINCSPs have been designed and analyzed [65.63–65]. All of these incorporate a set of parameters that may be used to control the size and difficulty of the problems. Often, these parameters can be used to create a set of problems that go through a *phase transition*. That is, we order the set on the parameters and observe how the algorithms behave when we move through the parameter space. In most constraint satisfaction problems we observe that the performance drops gradually until it reaches a minimum, after which it rises again. Most researchers test their algorithms in the region where the minimum is reached. Here the set of most difficult to solve problem instances is found. We will discuss these methods next.

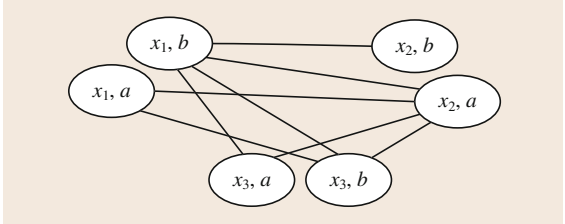

Fig. 65.3 Example of a $|V|$-partite hypergraph of a (BINCSP) with one solution: $\{\langle v_1, a\rangle, \langle v_2, b\rangle, \langle v_3, a\rangle\}$

The model most often used in empirical research on binary constraint satisfaction problems is one that uses four parameters to control, to some degree, the difficulty of an instance. By varying these global parameters one can characterize instances that are more likely to be either more or less difficult to solve. These parameters are: the number of variables $n = |V|$, the size of each variable's domain $m = |D_{v_1}| = |D_{v_2}| = \cdots = |D_{v_n}|$, the density of constraints p_1, and the average tightness of

all the constraints p_2. There are two ways of looking at parameters p_1 and p_2. We will use the following definitions.

Definition 65.3 (Density)
The *density* of a BINCSP is the ratio between the maximum number of constraints $\binom{n}{2}$ and the actual number of constraints $|C|$,

$$p_1 = \frac{|C|}{\binom{n}{2}} .$$

Definition 65.4 (Tightness)
The *tightness* of a constraint $c \subset C$ over the variables $v, w \in V$ of a BINCSP $\langle V, D, C\rangle$ is the ratio between the total number of forbidden variable assignments $|c|$ and the total number of combinations of variable assignments possible $m = |D_v||D_w|$,

$$p_2(c) = \frac{|c|}{m^2} .$$

Definition 65.5 (Average Tightness)
The *average tightness* of a BINCSP $\langle V, D, C\rangle$ is the sum of the tightness over all constraints divided by the number of constraints,

$$\overline{p_2} = \frac{\sum_{c\in C} p_2(c)}{|C|} .$$

These definitions give the density and tightness in terms of a ratio, or in other words, as the percentages of the maximum. Another way of looking at these two properties uses probabilities [65.66]. We could define the density of a BINCSP as the probability that a constraint exists between two variables. The tightness can be alternatively defined in an analogous way, as the probability that a conflict exists between two instantiations of two variables. The differences in these viewpoints becomes apparent in the different implementations of algorithms that generate BINCSPs, as with uniform generation the ratio in an instance is determined beforehand, while with probability the ratio will vary according to a normal distribution. When comparing studies it is important to know when probabilities are used whether the results reported are against the probability set or the actual measured ratio in the whole instance.

Table 65.1 Different models for the general method for generating binary constraint satisfaction problems

		Nogoods	
		Probability	**Uniform**
Constraints	Probability	Model A	Model C
	Uniform	Model D	Model B

The simplest way to empirically test the performance of an algorithm on solving CSPs is by generating instances using different settings for the four main parameters, n, m, p_1, and $\overline{p_2}$. However, there are two ways of choosing where to put constraints in a constraint network. We can choose the number of constraints we want to have beforehand and then uniformly distribute them in the constraint network. Alternatively, we can choose for each *possible* edge in the constraint network with the probability p_1 if this edge is inserted, i. e., a constraint is added. We will call the first model the *uniform model* and the second the *probability model*. The same categorization holds for nogoods. Given a constraint we can either distribute $\overline{p_2}m^2$ nogoods uniformly or with probability $\overline{p_2}$ decide which value pairs become nogoods. Now we can define four different models and we will name them according to the models in [65.63, 65]. The models are shown in Table 65.1.

Definition 65.6 (Parameter Vector of a BINCSP)
A *parameter vector* of a binary constraint satisfaction problem (BINCSP) with n variables and m as each variable's domain size is a 4-tuple $\langle n, m, p_1, \overline{p_2}\rangle$ of four parameters: the number of variables n, the domain size of each variable m, the density p_1, and the average tightness $\overline{p_2}$.

We can also characterize a set of binary constraints satisfaction problems using the parameter vector as a set B of BINCSP instances where

$$\forall \langle n, m, p_1, \overline{p_2}\rangle, \langle n', m', p_1', \overline{p_2}'\rangle \in B \\ \Leftrightarrow n = n' \wedge m = m' \wedge p_1 = p_1' \wedge \overline{p_2} = \overline{p_2}' .$$

Such a set we call a *suite of problem instances*.

Achlioptas et al. proves in [65.64] that as the number of variables becomes large almost all instances created by Models A–D become unsolvable. The reason lies in the existence of *flawed variables*. Whenever a variable v is involved in a constraint and has all its values incompatible with a value of an adjacent variable w, this variable is called *flawed*. In terms of compound labels using the constraint c over variables v and w this is

written as,

$$\forall v \in D_v : \nexists w \in D_w : \\ satisfies((\langle v, v\rangle, \langle w, w\rangle), c) \wedge c \in C \, .$$

When the number of variables is increased without changing the other parameters, the number of flawed variables will increase, thus making it easy to prove instances have no solution. To overcome the problems a new model is proposed [65.64]:

Definition 65.7 (Model E)
The graph C^{IT} is a random n-partite graph with m nodes in each part that is constructed by uniformly, independently, and with repetitions selecting $p_e\binom{n}{2}m^2$ edges out of the $\binom{n}{2}m^2$ possible ones.

The idea behind this model is that the difficulty is controlled by the tightness and not influenced by the structure of the constraint network. The parameter p_e is responsible for the average tightness of the BINCSP. However, it is not the same parameter as the average tightness $\overline{p_2}$. Because we allow repetitions in the process we end up with an average tightness smaller than or at most equal to p_e.

Parameter p_e also influences the value of p_1. In [65.65] we find the proof that using Model E with fairly small values ($p_e < 0.05$) will result in a fully connected constraint network ($p_1 = 1$). This is seen as a flaw in Model E, as many problems do not require a fully connected constraint network. This has led to yet another model.

MacIntyre et al. propose a more generalized version of Model E called *Model F* [65.65]. This model starts out the same way as Model E by generating $p_1\overline{p_2}m\binom{n}{2}$ nogoods. Afterwards, a constraint network is generated with exactly $p_1\binom{n}{2}$ edges in the uniform way. All nogoods that are not in a constraint in the constraint network are removed from the problem instance. Model E is the special case of Model F where $p_1 = 1$. The benefit of Model F is the ability to generate problems where $p_1 < 1$, which is more realistic towards real-world problems.

Craenen et al. [65.67] present the largest comparison study of EC and CP approaches for the BINCSP. In this study they compare the success rate and average number of conflict checks to a solution of 11 evolutionary algorithms. The best four evolutionary algorithms are compared with forward checking with conflict-directed backjumping [65.68], and the authors concluded the latter has a superior performance on every problem instance in the benchmark.

The following heuristic approaches are included in the study. In [65.69, 70], *Eiben* et al. propose to incorporate existing CSP heuristics into genetic operators. A study on the performance of these heuristic-based operators when solving binary CSPs was published in [65.71]. Two heuristic-based genetic operators are specified: an asexual operator that transforms one individual into a new one and a multi-parent operator that generates one offspring using a number of parents. In [65.72–74], *Riff-Rojas* introduced an EA for solving CSPs that uses information about the constraint network in the fitness function and in the genetic operators (crossover and mutation). The fitness function is based on the notion of the *error evaluation* of a constraint. *Marchiori* et al. introduced and investigated EAs for solving CSPs based on pre-processing and post-processing techniques [65.75–77]. Included in the comparison is the variant form [65.75, 78] that transforms constraints into a canonical form in such a way that there is only one single (type of) primitive constraint; we call this algorithm glass-box. This approach is used in constraint programming, where CSPs are given in implicit form by means of formulas of a given specification language. In [65.79, 80] *Handa* et al. formulate a coevolutionary algorithm where a population of schemata are parasitic on the host population. Schemata in this algorithm are individuals where a portion of variables in the individual has values while all other variables have do-not-care symbols represented by asterisks.

The following approaches with emphasis on adaptive features are included in the comparison; a co-evolutionary approach invented by *Paredis* and evaluated on different problems, such as neural net learning [65.81], constraint satisfaction [65.81, 82], and searching for cellular automata that solve the density classification task [65.83]. Furthermore, results on the performance of the co-evolutionary approach when facing the task of solving binary CSPs are reported in [65.84, 85]. In the co-evolutionary approach for CSPs two populations evolve according to a predator-prey model: a population of candidate solutions and a population of constraints. In the approach proposed by *Dozier* et al. in [65.86] and further refined and applied in [65.87–89], information about the constraints is incorporated both in the genetic operators and in the fitness function. In the microgenetic iterative descent algorithm the fitness function is adaptive and employs *Morris*' breakout creating

mechanism [65.90] to escape from local optima. The stepwise adaptation of weights mechanism was introduced by *Eiben* and *van der Hauw* [65.91, 92] as an improved version of the weight adaptation mechanism of *Eiben* et al. [65.93, 94]. The approach has been studied in several comparisons and often proved to be a robust technique for solving several specific CSPs [65.95–97]. A comprehensive study of different parameters and genetic operators can be found in [65.98]. The basic idea is that constraints that are not satisfied or variables causing constraint violations after a certain number of steps must be hard, thus must be given a high weight (penalty) in the fitness function.

65.5.4 Examination Timetabling

Examination timetabling has been studied for many years as it is a common problem in many organizations. Already in 1986, *Carter* gave an extended survey of work on automated timetabling [65.99]. He is also responsible for providing problem instances, which are still available and popular today [65.100], although a more diverse benchmark is used in the annual timetabling competition [65.101]. *Burke* et al. provide the most extensive recent surveys of automated timetabling in [65.102, 103]. Examination timetabling is just one of many problems under the topic of timetabling [65.104].

Timetabling as a problem has many different definitions due to different kinds of constraints and objectives. The definition that is most relevant for constraint satisfaction is often referred to as examination timetabling. The most abstract definition simply consists of a matrix C where $C_{i,j} = 1$ if exam i conflicts with exam j by having common students that must take both exams, $C_{i,j} = 0$ otherwise. This definition is equivalent to a graph coloring problem if the objective is to minimize the number of exam slots required, where the number of slots equals the number of colors required for coloring the graph with incidence matrix C. Hence, an appropriate approach to performance testing is via graph coloring instances based on examination timetabling, such as the problem instances labeled SCH (school) in the graph coloring instances suite provided by *Lewandowski* [65.105].

Many problem instances and problem instance generators exist. Infrequently, an International Timetabling Competition is organized by The International Series of Conferences on the Practice and Theory of Automated Timetabling. At each event, another definition of timetabling problems is tackled. The differences between definitions are in the objectives and the soft and hard constraints used. Hard constraints are treated the same as in constraint satisfaction, whereas soft constraints may be violated but will either incur an additional penalty on the objective function or be used to prioritize solutions otherwise, for instance, using a Pareto front. *Corne* et al. [65.106] identified five categories of constraints, unary, binary, capacity, event spread, and agent preference.

Three approaches exist to solving timetabling problems. The first approach is called *one-stage optimization*. It aggregates all types of constraints of one problem, often by summation, into one objective function where each type is assigned a weight. The advantage is that, in principle, the approach can be applied to any set of constraints. In practice, it may prove difficult to optimize such a function. Representations of the problem fall into the two main categories direct encoding (Sect. 65.3.1) [65.107] and indirect encoding (Sect. 65.3.2) [65.106, 108].

The second approach is called *two-stage optimization*. It first solves the problem of finding a feasible solution where all the hard constraints are satisfied. In the second stage it searches within the space set with these hard constraints and optimizes only against the soft constraints. The benefits are that during search we do not have to distinguish between feasible and infeasible constraints and, therefore, are not in danger of the search wandering off into an infeasible part of the search space. *Thompson* and *Dowsland* [65.109] were the first to report on this approach using simulated annealing, closely followed by the first EA by *Yu* and *Sung* [65.110].

The third approach uses *relaxation of constraints*. Typically, relaxation in timetabling is achieved by not assigning events to slots or by adding additional time slots. An early example of an EA is by *Burke* et al. [65.111], where an indirect encoding is used and additional time slots are used to relax the problem.

65.6 Creating Rather than Solving Problems

So far we have covered evolutionary computation for solving CSP. A contrasting idea proposed first for constraint satisfaction in [65.112] is to use evolutionary computation to generate problem instances. Such an approach allows a search for problem instances that adhere to certain properties as long as these can be measured efficiently by a fitness function.

A straightforward use for such an approach is to evolve problem instances that are difficult to solve for a particular algorithm. By measuring the efficiency of an algorithm to solve instances of a certain problem we can then change the instances with the aim of decreasing the efficiency. Measurements for efficiency of EC for CSP are discussed in Sect. 65.4.1. It is important to note that the algorithm we are evolving problem instances for can be of any kind, as long as we can execute it on problem instances generated and we can measure its efficiency.

Such hard problem instances identify the weak spots in the algorithm that tries to solve it. Moreover, if we can characterize a set of problem instances where all members of the set are hard for an algorithm, then we can use that characterization to decide what algorithm is suitable for solving a new problem instance. That is, if the work required to obtain the characteristics of one instance takes less effort than solving the actual problem instance itself [65.113].

65.6.1 Evolving Binary Constraint Satisfaction Problem Instances

The first application to constrained problems was for the binary constraint satisfaction problem (Sect. 65.5.3), where problem instances are represented as a binary vector with each element corresponding to the element of a conflict matrix between two variables [65.114]. Even the small instances investigated in the study led to large vectors, i. e., with 15 variables each with a domain of size 15, the corresponding vector has $\binom{15}{2} \cdot 15^2 = 23\,625$ elements. Results with problem instances of this size show problem instances can be created that are far more difficult to solve than when creating a much larger set of randomly generated instances [65.112]. Furthermore, analysis of these instances provides an insight as to what structure is responsible for making instances difficult for the algorithm; two well-known algorithms from constraint programming were tested: chronological backtracking [65.115] and forward checking with conflict-directed backjumping [65.116].

65.6.2 Evolving Boolean Satisfiability Problem Instances

In [65.114] an evolutionary algorithm is used to evolve solvable Boolean satisfiability problem instances that are in conjunctive normal form and have three variables per clause. A 3-SAT problem is represented by a list of 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 3, hence there are 1 313 400 unique clauses. This representation has strong advantages over a simple *one gene for every literal* approach. 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.

Two problem solvers are used from the annual SAT competition [65.117]; both are based on the *Davis–Putnum* procedure [65.4]. zChaff [65.118] is based on Chaff [65.119], a SAT solver that employs a particularly efficient implementation of Boolean constraint propagation and a novel low overhead decision strategy. Relsat [65.120] is explained in [65.121, 122]. In both solvers, the number of states of instantiations are enumerated to determine the search effort required.

The change of certain structural properties over the duration of evolution was analyzed. Two established properties were used: the number of solutions [65.123, 124] and the backbone size [65.125]. No clear relationship was identified with these properties.

However, a new relationship was identified: when problem instances are becoming more difficult to solve, the variance in the frequency in variable usage decreases. In other words, the distribution of variables throughout the instances is more uniform when problems are more difficult to solve.

65.6.3 Further Investigations

The application of evolutionary computation in problem generation is widespread. *Smith-Miles* and

Lopes [65.126] provide an extensive review in terms of measuring instance difficulty in combinatorial optimization problems, which also discusses studies that evolve problem instances for constrained optimization as well as for constraint satisfaction problems.

The maximization of the effort required to solve a problem instance highlights only one aspect of the problem difficulty. Another aspect that looks at the effectiveness is to maximize the distance a solver is able to reach to the optimal solution. To compute this distance, we require the fitness of the optimal solution a priori. Note, however, we do not need to know what the optimal solution is, only its fitness. Another approach is to directly compare solvers by maximizing the difference in some aspect, e.g., efficiency or effectiveness, between two solvers.

65.7 Conclusions and Future Directions

Research on solving constraint satisfaction problems with evolutionary computation has produced a rich set of research papers that contribute solvers, insights into solvers and their performance, and heuristic subroutines. One major flaw in this research has remained consistent over the past 20 years: most studies compare performance results only to other evolutionary or closely associated techniques. Even recent studies, such as [65.127–129], restrict themselves to comparing only results from other heuristic methods or have not included alternative techniques at all.

Many studies report on the promising performance of a particular evolutionary algorithm over another existing heuristic technique. The few systematic studies that do compare evolutionary and constraint programming techniques conclude that constraint programming is superior in terms of efficiency [65.60, 67]. Also, constraint programming techniques are generally sound and, therefore, given sufficient time, always find a solution or proof that none exists. Hence, these solvers are more effective unless they are bounded by time. Recent efforts have shown success in speeding up modern DPLL-based techniques using heuristics for guiding the search [65.130, 131].

In Sect. 65.5 we reviewed many techniques that were developed and studied for the purpose of improving EC in terms of efficiency and effectiveness. The vast majority of these techniques was applied to one problem only. A huge benefit would come from studies that show the success of a technique across several CSPs. Such studies would be especially opportune for the SAT problem, which is still the most actively used CSP for benchmarking algorithms [65.132].

References

65.1 K. Apt: *Principles of Constraint Programming* (Cambridge Univ. Press, Cambridge 2003)

65.2 B.G.W. Craenen, A.E. Eiben: Hybrid evolutionary algorithms for constraint satisfaction problems: Memetic overkill?, 2005 IEEE Congr. Evol. Comput., Vol. 3 (2005) pp. 1922–1928

65.3 R. Kibria, Y. Li: Optimizing the initialization of dynamic decision heuristics in DPLL SAT solvers using genetic programming, Lect. Notes Comput. Sci. **3905**, 331–340 (2006)

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

65.5 H.E. Dudeney: Cryptarithm, Strand Mag. **68**, 97 and 214 (1924)

65.6 M.R. Garey, D.S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W.H. Freeman, San Francisco 1979)

65.7 R. Lewis: Metaheuristics can solve Sudoku puzzles, J. Heuristics **13**, 387–401 (2007)

65.8 E. Tsang: *Foundations of Constraint Satisfaction* (Academic, London 1993)

65.9 R. Dechter: *Constraint Processing* (Morgan Kaufmann, San Francisco 2003) pp. 1–481

65.10 C. Lecoutre: *Constraint Networks: Techniques and Algorithms* (Wiley, Hoboken 2009)

65.11 H. Chen: A rendezvous of logic, complexity, and algebra, ACM Comput. Surv. **42**(1), 2 (2009)

65.12 B. Bernhardsson: Explicit solutions to the *n*-queens problem for all *n*, SIGART Bull. **2**, 7 (1991)

65.13 T. Bäck, D. Fogel, Z. Michalewicz (Eds.): *Handbook of Evolutionary Computation* (Oxford Univ. Press, New York 1997)

65.14 F. Rossi, P. Van Beek, T. Walsh: *Handbook of Constraint Programming* (Elsevier, Amsterdam 2006)

65.15 D. Brélaz: New methods to color the vertices of a graph, Communications ACM **22**, 251–256 (1979)

65.16 D.B. Fogel: *Evolutionary Computation: Towards a New Philosophy of Machine Intelligence*, 2nd edn. (Wiley, Hoboken 1999)

65.17 A.E. Eiben, Z. Ruttkay: Self-adaptivity for constraint satisfaction: Learning penalty functions, Int. Conf. Evol. Comput. (1996) pp. 258–261

65.18 R. Hinterding, Z. Michalewicz, A.E. Eiben: Adaptation in evolutionary computation: A survey, Proc. 4th IEEE Conf. Evol. Comput. (1997) pp. 65–69

65.19 T. Bäck: Introduction to the special issue: Self-adaptation, Evol. Comput. **9**(2), 3–4 (2001)

65.20 T. Runnarson, X. Yao: Constrained evolutionary optimization – The penalty function approach. In: *Evolutionary Optimization*, ed. by R. Sarker, M. Mohammadian, X. Yao (Kluwer, Boston 2002) pp. 87–113, Chap. 4

65.21 J.T. Richardson, M.R. Palmer, G. Liepins, M. Hilliard: Some guidelines for genetic algorithms with penalty functions, Proc. 3rd Int. Conf. Genet. Algoritms. (1989) pp. 191–197

65.22 M.L. Braun, J.M. Buhmann: The noisy Euclidean traveling salesman problem and learning, Proc. 2001 Neural Inf. Process. Syst. Conf. (2002)

65.23 D. Whitley, J.P. Watson, A. Howe, L. Barbulescu: Testing, evaluation and performance of optimization and learning systems. In: *Adaptive Computing in Design and Manufacturer*, ed. by I.C. Parmee (Springer, Berlin, Heidelberg 2002) pp. 27–39

65.24 A. Biere, M. Heule, H. van Maaren, T. Walsh: *Handbook of Satisfiability* (IOS, Amsterdan 2009)

65.25 V. Malek: *Introduction to Mathematics of Satisfiability* (Chapman Hall, Boca Raton 2009)

65.26 S.A. Cook: The complexity of theorem-proving procedures, Proc. 3rd Annu. ACM Symp. Theory Comput. (1971) pp. 151–158

65.27 M. Utting, B. Legeard: *Practical Model-Based Testing: A Tools Approach* (Morgan Kaufmann, San Francisco 2007)

65.28 A. Bundy: A science of reasoning: extended abstract, Proc. 10th Int. Conf. Autom. Deduc. (1990) pp. 633–640

65.29 D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins: PDDL – The planning domain definition language, Tech. Rep. TR-98-003, Yale Center for Computational Vision and Control (1998)

65.30 R. Drechsler, S. Eggersglüß, G. Fey, D. Tille: *Test Pattern Generation using Boolean Proof Engines* (Springer, Berlin, Heidelberg 2009) pp. 1–192

65.31 D. He, A. Choi, K. Pipatsrisawat, A. Darwiche, E. Eskin: Optimal algorithms for haplotype assembly from whole-genome sequence data, Bioinformatics **26**(12), i183–i190 (2010)

65.32 I.V. Tetko, D.J. Livingstone, A.I. Luik: Neural network studies. 1. Comparison of overfitting and overtraining, J. Chem Inf. Comput. Sci. **35**, 826–833 (1995)

65.33 J.-K. Hao, R. Dorne: An empirical comparison of two evolutionary methods for satisfiability problems, Int. Conf. Evol. Comput. (1994) pp. 451–455

65.34 J. Gottlieb, N. Voss: Fitness functions and genetic operators for the satisfiability problem, Lect. Notes Comput. Sci. **1363**, 55–68 (1997)

65.35 J. Gottlieb, N. Voss: Improving the performance of evolutionary algorithms for the satisfiability problem by refining functions, Lect. Notes Comput. Sci. **1498**, 755–764 (1998)

65.36 G. Folino, C. Pizzuti, G. Spezzano: Solving the satisfiability problem by a parallel cellular genetic algorithm, Proc. 24th Euromicro Conf. (1998) pp. 715–722

65.37 N. Nemer-Preece, R.W. Wilkerson: Parallel genetic algorithm to solve the satisfiability problem, Proc. 1998 ACM Symp. Appl. Comput. (1998) pp. 23–28

65.38 C. Rossi, E. Marchiori, J.N. Kok: An adaptive evolutionary algorithm for the satisfiability problem, Proc. 2000 ACM Symp. Appl. Comput. (2000) pp. 463–469

65.39 J.-K. Hao, F. Lardeux, F. Saubion: Evolutionary computing for the satisfiability problem, Lect. Notes Comput. Sci. **2611**, 258–267 (2003)

65.40 M.E. Bachir Menai: An evolutionary local search method for incremental satisfiability, Lect. Notes Comput. Sci. **3249**, 143–156 (2004)

65.41 L. Aksoy, E.O. Günes: An evolutionary local search algorithm for the satisfiability problem, Lect. Notes Comput. Sci. **3949**, 185–193 (2005)

65.42 M.E. Bachir Menai, M. Batouche: Solving the maximum satisfiability problem using an evolutionary local search algorithm, Int. Arab J. Inf. Technol. **2**(2), 154–161 (2005)

65.43 P. Guo, W. Luo, Z. Li, H. Liang, X. Wang: Hybridizing evolutionary negative selection algorithm and local search for large-scale satisfiability problems, Lect. Notes Comput. Sci. **5821**, 248–257 (2009)

65.44 Y. Kilani: Comparing the performance of the genetic and local search algorithms for solving the satisfiability problems, Appl. Soft. Comput. **10**(1), 198–207 (2010)

65.45 R.H. Kibria: Soft Computing Approaches to DPLL SAT Solver Optimization, Ph.D. Thesis (TU Darmstadt, Darmstadt 2011)

65.46 M. Davis, G. Logemann, D. Loveland: A machine program for theorem-proving, Communications ACM **5**(7), 394–397 (1962)

65.47 R. Lewis, J. Thompson: On the application of graph colouring techniques in round-robin sports scheduling, Comput. Oper. Res. **38**, 190–204 (2011)

65.48 S.S. Muchnick: *Advanced Compiler Design and Implementation* (Morgan Kaufmann, San Fransisco 1997)

65.49 W.K. Hale: Frequency assignment: Theory and applications, Proc. IEEE **68**(12), 1497–1514 (1980)

65.50 J. Culberson: Graph Coloring Page (2010), available online at http://webdocs.cs.ualberta.ca/~joe/Coloring/

65.51 J.C. Culberson, F. Luo: Exploring the *k*-colorable landscape with iterated greedy. In: *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26, ed. by D.S. Johnson, M.A. Trick (American Mathematical Society, Providence 1996) pp. 245–284

65.52 M. Brockington, J.C. Culberson: Camouflaging independent sets in quasi-random graphs. In: *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26, ed. by D.S. Johnson, M.A. Trick (American Mathematical Society, Providence 1996) pp. 75–88

65.53 G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, P.W. Markstein: Register allocation via coloring, Comput. Lang. **6**(1), 47–57 (1981)

65.54 D.J.A. Welsh, M.B. Powell: An upper bound for the chromatic number of a graph and its application to timetabling problems, Comput. J. **10**(1), 85–86 (1967)

65.55 C. Fleurent, J. Ferland: Genetic and hybrid algorithms for graph coloring, Ann. Oper. Res. **63**(3), 437–461 (1996)

65.56 C. Fleurent, J.A. Ferland: Object-oriented implementation of heuristic search methods for graph coloring, maximum clique, and satisfiability. In: *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26, ed. by D.S. Johnson, M.A. Trick (American Mathematical Society, Providence 1996) pp. 619–652

65.57 G. von Laszewski: Intelligent structural operators for the k-way graph partitioning problem, Proc. 4th Int. Conf. Genet. Algorithms (1991) pp. 45–52

65.58 L. Davis: Order-based genetic algorihms and the graph coloring problem. In: *Handbook of Genetic Algorithms*, ed. by L. Davis (Van Nostrand Reinhold, New York 1991) pp. 72–90

65.59 P.E. Coll, G.A. Durán, P. Moscato: A discussion on some design principles for efficient crossover operators for graph coloring problems, An. XXVII Simp. Brasil. Pesqui. Oper. (1995)

65.60 I. Juhos, J.I. van Hemert: Contraction-based heuristics to improve the efficiency of algorithms solving the graph colouring problem. In: *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, ed. by C. Cotta, J.I. van Hemert (Springer, Berlin, Heidelberg 2008) pp. 167–184

65.61 I. Juhos, J.I. van Hemert: Graph colouring heuristics guided by higher order graph properties, Lect. Notes Comput. Sci. **4972**, 97–109 (2008)

65.62 I. Juhos, J.I. van Hemert: Increasing the efficiency of graph colouring algorithms with a representation based on vector operations, J. Softw. **1**(2), 24–33 (2006)

65.63 E.M. Palmer: *Graphical Evolution* (Wiley, New York 1985)

65.64 D. Achlioptas, L.M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy, Y.C. Stamatiou: Random constraint satisfaction: A more accurate picture, Lect. Notes Comput. Sci. **1330**, 107–120 (1997)

65.65 E. MacIntyre, P. Prosser, B.M. Smith, T. Walsh: Random constraint satisfaction: Theory meets practice. In: *Principles and Practice of Constraint Programming — CP98*, ed. by M. Maher, J.-F. Puget (Springer, Berlin, Heidelberg 1998) pp. 325–339

65.66 E. Freuder, R.J. Wallace: Partial constraint satisfaction, Artif. Intell. **65**, 363–376 (1992)

65.67 B.G.W. Craenen, A.E. Eiben, J.I. van Hemert: Comparing evolutionary algorithms on binary constraint satisfaction problems, IEEE Trans. Evol. Comput. **7**(5), 424–444 (2003)

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

65.69 A.E. Eiben, P.-E. Raué, Z. Ruttkay: Heuristic Genetic Algorithms for Constrained Problems, Part I: Principles, Tech. Rep. IR-337 (Vrije Universiteit Amsterdam 1993)

65.70 A.E. Eiben, P.-E. Raué, Z. Ruttkay: Solving constraint satisfaction problems using genetic algorithms, Proc. 1st IEEE Conf. Evol. Comput. (1994) pp. 542–547

65.71 B.G.W. Craenen, A.E. Eiben, E. Marchiori: Solving constraint satisfaction problems with heuristic-based evolutionary algorithms, Congr. Evol. Comput. (2000)

65.72 M.C. Riff-Rojas: Using the knowledge of the constraint network to design an evolutionary algorithm that solves CSP, Proc. 3rd IEEE Conf. Evol. Comput. (1996) pp. 279–284

65.73 M.C. Riff-Rojas: Evolutionary search guided by the constraint network to solve CSP, Proc. 4th IEEE Conf. Evol. Comput. (1997) pp. 337–348

65.74 M.-C. Riff-Rojas: A network-based adaptive evolutionary algorithm for constraint satisfaction problems. In: *Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization*, ed. by S. Voss (Kluwer, Boston 1998) pp. 325–339

65.75 E. Marchiori: Combining constraint processing and genetic algorithms for constraint satisfaction problems, Proc. 7th Int. Conf. Genet. Algorithms (1997) pp. 330–337

65.76 E. Marchiori, A. Steenbeek: A genetic local search algorithm for random binary constraint satisfaction problems, Proc. ACM Symp. Appl. Comput. (2000) pp. 458–462

65.77 B.G.W. Craenen, A.E. Eiben, E. Marchiori, A. Steenbeek: Combining local search and fitness function adaptation in a GA for solving binary constraint satisfaction problems, Proc. Genet. Evol. Comput. Conf. (2000)

65.78 P. van Hentenryck, V. Saraswat, Y. Deville: Constraint processing in cc(FD). In: *Constraint Programming: Basics and Trends*, ed. by A. Podelski (Springer, Berlin, Heidelberg 1995)

65.79 H. Handa, C.O. Katai, N. Baba, T. Sawaragi: Solving constraint satisfaction problems by using coevolutionary genetic algorithms, Proc. 5th IEEE Conf. Evol. Comput. (1998) pp. 21–26

65.80 H. Handa, N. Baba, O. Katai, T. Sawaragi, T. Horiuchi: Genetic algorithm involving coevolution mechanism to search for effective genetic information, Proc. 4th IEEE Conf. Evol. Comput. (1997)

65.81 J. Paredis: Co-evolutionary computation, Artif. Life **2**(4), 355–375 (1995)

65.82 J. Paredis: Coevolutionary constraint satisfaction, Lect. Notes Comput. Sci. **866**, 46–55 (1994)

65.83 J. Paredis: Coevolving cellular automata: Be aware of the red queen, Proc. 7th Int. Conf. Genet. Algorithms (1997)

65.84 A.E. Eiben, J.I. van Hemert, E. Marchiori, A.G. Steenbeek: Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function, Lect. Notes Comput. Sci. **1498**, 196–205 (1998)

65.85 J.I. van Hemert: Applying Adaptive Evolutionary Algorithms to Hard Problems, M.Sc. Thesis (Leiden University, Leiden 1998)

65.86 G. Dozier, J. Bowen, D. Bahler: Solving small and large constraint satisfaction problems using a heuristic-based microgenetic algorithm, Proc. 1st IEEE Conf. Evol. Comput. (1994) pp. 306–311

65.87 J. Bowen, G. Dozier: Solving constraint satisfaction problems using a genetic/systematic search hybride that realizes when to quit, Proc. 6th Int. Conf. Genet. Algorithms (Morgan Kaufmann, Burlington 1995) pp. 122–129

65.88 G. Dozier, J. Bowen, D. Bahler: Solving randomly generated constraint satisfaction problems using a micro-evolutionary hybrid that evolves a population of hill-climbers, Proc. 2nd IEEE Conf. Evol. Comput. (1995) pp. 614–619

65.89 P.J. Stuckey, V. Tam: Improving evolutionary algorithms for efficient constraint satisfaction, Int. J. Artif. Intell. Tools **8**(4), 363–384 (1999)

65.90 P. Morris: The breakout method for escaping from local minima, Proc. 11th Natl. Conf. Artif. Intell. (1993) pp. 40–45

65.91 A.E. Eiben, J.K. van der Hauw: Adaptive penalties for evolutionary graph-coloring, Lect. Notes Comput. Sci. **1363**, 95–106 (1998)

65.92 J.K. van der Hauw: Evaluating and Improving Steady State Evolutionary Algorithms on Constraint Satisfaction Problems, M.Sc. Thesis (Leiden University, Leiden 1996)

65.93 A.E. Eiben, P.-E. Raué, Z. Ruttkay: Constrained problems. In: *Practical Handbook of Genetic Algorithms*, ed. by L. Chambers (Taylor Francis, Boca Raton 1995) pp. 307–365

65.94 A.E. Eiben, Z. Ruttkay: Self-adaptivity for constraint satisfaction: Learning penalty functions, Proc. 3rd IEEE Conf. Evol. Comput. (1996) pp. 258–261

65.95 T. Bäck, A.E. Eiben, M.E. Vink: A superior evolutionary algorithm for 3-SAT, Lect. Notes Comput. Sci. **1477**, 125–136 (1998)

65.96 A.E. Eiben, J.K. van der Hauw, J.I. van Hemert: Graph coloring with adaptive evolutionary algorithms, J. Heuristics **4**(1), 25–46 (1998)

65.97 A.E. Eiben, J.I. van Hemert: SAW-ing EAs: Adapting the fitness function for solving constrained problems. In: *New Ideas in Optimization*, ed. by D. Corne, M. Dorigo, F. Glover (McGraw Hill, New York 1999) pp. 389–402

65.98 B.G.W. Craenen, A.E. Eiben: Stepwise adaption of weights with refinement and decay on constraint satisfaction problems, Proc. Genet. Evol. Comput. Conf. (2001) pp. 291–298

65.99 M.W. Carter: A survey of practical applications of examination timetabling algorithms, Oper. Res. **34**, 193–202 (1986)

65.100 M.W. Carter, G. Laporte, S.Y. Lee: Examination timetabling: Algorithmic strategies and application, J. Oper. Res. Soc. **47**(3), 373–383 (1996)

65.101 International Timetabling Competition 2011: available online at http://www.utwente.nl/ctit/itc2011/

65.102 E.K. Burke, S. Petrovic: Recent research directions in automated timetabling, Eur. J. Oper. Res. **140**(2), 266–280 (2002)

65.103 R. Qu, E.K. Burke, B. Mccollum, L.T. Merlot, S.Y. Lee: A survey of search methodologies and automated system development for examination timetabling, J. Sched. **12**, 55–89 (2009)

65.104 E.K. Burke, D. Corne, B. Paechter, P. Ross (Eds.): *Proc. 1st Int. Conf. Pract. Theory Autom. Timetabling* (Napier University, Edinburgh 1995)

65.105 G. Lewandowski: *Course scheduling: Metrics, Models, and Methods* (Xavier University, Cincinnati 1996)

65.106 D. Corne, P. Ross, H.-L. Fang: Evolving timetables. In: *Practical Handbook of Genetic Algorithms: Applications*, Vol. I, ed. by L. Chambers (Taylor Francis, Boca Raton 1995) pp. 219–276

65.107 A. Colorni, M. Dorigo, V. Maniezzo: Metaheuristics for high school timetabling, Comput. Optim. Appl. **9**(3), 275–298 (1998)

65.108 M.P. Carrasco, M.V. Pato: A multiobjective genetic algorithm for the class/teacher timetabling prob-

lem, Proc. 3rd Int. Conf. Pract. Theory Autom. Timetabling (2001) pp. 3–17
65.109 J.M. Thompson, K.A. Dowsland: A robust simulated annealing based examination timetabling system, Comput. Oper. Res. **25**, 637–648 (1998)
65.110 E. Yu, K.-S. Sung: A genetic algorithm for a university weekly courses timetabling problem, Int. Trans. Oper. Res. **9**(6), 703–717 (2002)
65.111 E.K. Burke, D. Elliman, R.F. Weare: A hybrid genetic algorithm for highly constrained timetabling problems, Proc. 6th Int. Conf. Genet. Algorithms (1995) pp. 605–610
65.112 J.I. van Hemert: Evolving binary constraint satisfaction problem instances that are difficult to solve, Proc. IEEE 2003 Congr. Evol. Comput. (New York) (2003) pp. 1267–1273
65.113 K. Smith-Miles, J.I. van Hemert: Discovering the suitability of optimisation algorithms by learning from evolved instances, Ann. Math. Artif. Intell. **61**(2), 87–104 (2011)
65.114 J.I. van Hemert: Evolving combinatorial problem instances that are difficult to solve, Evol. Comput. **14**(4), 433–462 (2006)
65.115 S.W. Golomb, L.D. Baumert: Backtrack programming, Journal ACM **12**(4), 516–524 (1965)
65.116 P. Prosser: Hybrid algorithms for the constraint satisfaction problem, Comput. Intell. **9**(3), 268–299 (1993)
65.117 D. Le Berre, L. Simon: *Sat Competitions* http://www.satcompetition.org. (2005)
65.118 Z. Fu: *zChaff* (Princeton University) Version 2004.11.15 http://www.princeton.edu/~chaff/zchaff.html (2004)
65.119 M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik: Chaff: Engineering an efficient SAT solver, Proc. 38th Design Autom. Conf. (2001) pp. 530–535
65.120 R. Bayardo: *Relsat. Version 2.00* (IBM, San Jose 2005), available online at http://www.almaden.ibm.com/cs/people/bayardo/resources.html
65.121 R. Bayardo, R.C. Schrag: Using CSP look-back techniques to solve real world SAT instances, Proc. 14th Natl. Conf. Artif. Intell. (1997) pp. 203–208
65.122 R. Bayardo, J. Pehoushek: Counting models using connected components, Proc. 17th Natl. Conf. Artif. Intell. (2000)
65.123 D. Achlioptas, C.P. Gomes, H.A. Kautz, B. Selman: Generating satisfiable problem instances, Proc. 17th Natl. Conf. Artif. Intell. 12th Conf. Innov. Appl. Artif. Intell. (2000) pp. 256–261
65.124 D. Achlioptas, H. Jia, C. Moore: Hiding satisfying assignments: Two are better than one, J. Artif. Intell. Res. **24**, 623–639 (2005)
65.125 S. Boettcher, G. Istrate, A.G. Percus: Spines of random constraint satisfaction problems: Definition and impact on computational complexity, 8th Int. Symp. Artif. Intell. Math. (2005), extended version
65.126 K. Smith-Miles, L. Lopes: Review: Measuring instance difficulty for combinatorial optimization problems, Comput. Oper. Res. **39**, 875–889 (2012)
65.127 R. Abbasian, M. Mouhoub: An efficient hierarchical parallel genetic algorithm for graph coloring problem, Proc. 13th Annu. Conf. Genet. Evol. Comput. (2011) pp. 521–528
65.128 D.C. Porumbel, J.-K. Hao, P. Kuntz: An evolutionary approach with diversity guarantee and well-informed grouping recombination for graph coloring, Comput. Oper. Res. **37**(10), 1822–1832 (2010)
65.129 M. Mouhoub, B. Jafari: Heuristic techniques for variable and value ordering in CSPs, Proc. 13th Annu. Conf. Genet. Evol. Comput. (2011) pp. 457–464
65.130 J. Chen: Building a hybrid sat solver via conflict-driven, look-ahead and XOR reasoning techniques, Lect. Notes Comput. Sci. **5584**, 298–311 (2009)
65.131 A. Balint, M. Henn, O. Gableske: A novel approach to combine a SLS- and a DPLL-solver for the satisfiability problem, Lect. Notes Comput. Sci. **5584**, 284–297 (2009)
65.132 O. Kullmann (Ed.): *Theory and Applications of Satisfiability Testing – SAT 2009*, Lecture Notes in Computer Science, Vol. 558 (Springer, Berlin, Heidelberg 2009)