1st International Workshop on Local Search Techniques in Constraint Satisfaction

Toronto, Canada
September 27, 2004

Workshop Organisers

Justin Pearson, Uppsala University
Magnus Ågren, Uppsala University
Markus Bohlin, SICS

Held in conjunction with

CP 2004
Tenth International Conference on
Principles and Practice of
Constraint Programming

September 27 - October 1, 2004

Toronto Marriott Downtown Eaton Centre

Toronto, Canada

Overview

Local search is an efficient method for solving hard combinatorial (optimisation) problems. Instances that are still infeasible for global search may be solvable using local search techniques. Traditionally, the implementation of local search algorithms was error-prone and time-consuming since this usually meant application-tailored algorithms in some low-level imperative language. In recent years however, following the spirit of constraint programming, systems providing high-level modelling abstractions together with efficient solving mechanisms have evolved. These systems provide general frameworks for problem solving using neighbourhood techniques, resulting in a more declarative solving approach for the user. Local search and constraint satisfaction is an active area of research that attracts and unites many different fields such as computer science and mathematics.

The First International Workshop on Local Search Techniques in Constraint Satisfaction, LSCS '04, is held at the Tenth International Conference on Principles and Practice of Constraint Programming, CP '04, at Toronto Marriott Downtown Eaton Centre, Toronto, Canada. It brings together recent research on local search and its integration with constraint programming. The workshop focuses on any aspect concerning local search. This includes, but is not limited to, new search heuristics, meta-heuristics, techniques for (incremental) cost calculations, local search systems, comparing or combining global and local search, and applications solved using neighbourhood techniques.

The workshop organisers would like to thank all the people that attended the workshop as well as the contributing authors and the rest of the programme committee.

Workshop Organisers

Justin Pearson and **Magnus Ågren**, Department of Information Technology, Uppsala University, Box 337, S-751 05 Uppsala, Sweden.
Markus Bohlin Swedish Institute of Computer Science (SICS AB), IDt/MdH, Box 883, S-721 23 Västerås, Sweden.

Program Committee

Gilles Pesant, École Polytechnique de Montréal, Montréal, Canada
Steven Prestwich, University College, Cork, Ireland
Meinolf Sellmann, Brown University, Providence, RI, USA
Pascal Van Hentenryck, Brown University, Providence, RI, USA

Contents

Iterative Forward Search: Combining Local Search with Maintaining Arc Consistency and a Conflict-based Statistics

Tomáš Müller[1], Roman Barták[1] and Hana Rudová[2]

[1] Faculty of Mathematics and Physics, Charles University
Malostranské nám. 2/25, Prague, Czech Republic
{muller|bartak}@ktiml.mff.cuni.cz
[2] Faculty of Informatics, Masaryk University
Botanická 68a, Brno 602 00, Czech Republic
hanka@fi.muni.cz

Abstract. The paper presents an iterative forward search framework for solving constraint satisfaction and optimization problems. This framework combines ideas of local search, namely improving a solution by local steps, with principles of depth-first search, in particular extending a partial feasible assignment towards a solution. Within this framework, we also propose and study a conflict-based statistics and explanation-based arc consistency maintenance. To show the versatility of the proposed framework, the dynamic backtracking algorithm with maintaining arc consistency is presented as a special instance of the iterative forward search framework. The presented techniques are compared on random constraint satisfaction problems and a real-life lecture timetabling problem.

1 Introduction

Many real-life industrial and engineering problems can be modeled as finite constraint satisfaction problems (CSP). A CSP consists of a set of variables associated with finite domains and a set of constraints restricting the values that the variables can simultaneously take. In a complete solution of a CSP, a value is assigned to every variable from the variable's domain, in such a way that every constraint is satisfied. Algorithms for solving CSPs usually fall into one of two main families: systematic search algorithms and local search algorithms.

Most algorithms for solving CSPs search systematically through the possible assignments of values to variables. Such algorithms are guaranteed to find a solution, if one exists, or to prove that the problem has no solution. They start from an empty solution (no variable is assigned) that is extended towards a complete solution satisfying all the constraints in the problem. Backtracking occurs when a dead-end is reached. The biggest problem of such backtrack-based algorithms is that they typically make early mistakes in the search, i.e., a wrong early assignment can cause a whole subtree to be explored with no success. There

are several ways of improving standard chronological backtracking. Look-back enhancements exploit information about the search which has already been performed, e.g., backmarking or backjumping [6]. Look-ahead enhancements exploit information about the remaining search space via filtering techniques (e.g., via maintaining arc consistency described in [1, 2]) or variable and value ordering heuristics [13]. The last group of enhancements is trying to refine the search tree during the search process, e.g., dynamic backtracking [8].

Local search algorithms [13] (e.g., min-conflict [14] or tabu search [7]) perform an incomplete exploration of the search space by repairing an infeasible complete assignment. Unlike systematic search algorithms, local search algorithms move from one complete (but infeasible) assignment to another, typically in a non-deterministic manner, guided by heuristics. In general, local search algorithms are incomplete, they do not guarantee finding a complete solution satisfying all the constraints. However, these algorithms may be far more efficient (wrt. response time) than systematic ones in finding a solution. For optimization problems, they can reach a far better quality in a given time frame.

In this paper, we present an iterative forward search (IFS) framework (based on our earlier work published in [15]) to solve CSPs. This framework is close to local search methods; however, it maintains a partial feasible solution as opposed to the complete conflicting assignment characteristic of local search. Similarly to local search, we process local changes in the solution. We also describe how to extend the presented algorithm with dynamic maintenance of arc consistency and conflict-based statistics. New conflict-based statistics is proposed to improve the quality of the final solution. Conflicts during the search are memorized and their potential repetition is minimized. The presented framework is very close to local search algorithms, but unlike them, it can be easily refined into a dynamic backtracking algorithm by enforcement of several basic rules.

There are several other approaches which try to combine local search methods together with backtracking based algorithms. For example, the decision repair algorithm presented in [11] repeatedly extends a set of assignments (called decisions) satisfying all the constraints, like in backtrack-based algorithms. It performs a local search to repair these assignments when a dead-end is reached (i.e., these decisions become inconsistent). After these decisions are repaired, the construction of the solution continues to the next dead-end. Unlike this approach, our algorithm operates more like the local search method – it does not execute a local search after a dead-end is reached but it applies the same local steps during search. A similar approach is used in the algorithm presented in [18] as well.

One of the primary areas to which we intended to apply the proposed algorithm is a course timetabling problem at Purdue University (USA), described in details in [17]. It is a real-life large-scale problem that includes features of over-constrained as well as optimization problems. The goal is to timetable more than 800 lectures to a limited number of lecture rooms (about 50) and to satisfy as many as possible individual course requests of almost 30,000 students. Moreover, the algorithm should be able to react to additional changes, in particular, the

algorithm should be capable of repairing a modified timetable where some hard constraints are violated by the user changes. IFS satisfies all these requirements and as we will show later, it produces better solutions than existing approaches.

The paper is organized as follows. In the next section, we will describe the iterative search algorithm formally. Special subsections will be devoted to extensions of IFS, namely conflict-based statistics and maintaining dynamic arc consistency. After that, we will show how dynamic backtracking with maintaining arc consistency [10] can be rewritten as a special instance of IFS. A short summary of the implementation together with the experimental results for random CSPs and a real-life timetabling problem will conclude the paper.

2 Iterative Forward Search Algorithm

The iterative forward search algorithm, that we propose here, is based on ideas of local search methods [13]. However, in contrast to classical local search techniques, it operates over feasible, though not necessarily complete solutions. In such a solution, some variables can be left unassigned. Still all hard constraints on assigned variables must be satisfied. Similarly to backtracking based algorithms, this means that there are no violations of hard constraints.

Working with feasible incomplete solutions has several advantages compared to the complete infeasible assignments that usually occur in local search techniques. For example, when the solver is not able to find a complete solution, an incomplete (but feasible) one can be returned, e.g., a solution with the least number of unassigned variables found. Moreover, because of the iterative character of the search, the algorithm can easily start, stop, or continue from any feasible solution, either complete or incomplete.

The search processeds iteratively (see Fig. 1 for algorithm). During each step,

```
procedure SOLVE(initial)          // initial solution is the parameter
    iteration = 0;                // iteration counter
    current = initial;            // current solution
    best = initial;               // best solution
    while canContinue(current, iteration) do
        iteration = iteration + 1;
        variable = selectVariable(current);
        value = selectValue(current, variable);
        UNASSIGN(current, CONFLICTING_VARIABLES(current, variable, value));
        ASSIGN(current, variable, value);
        if better(current, best) then best = current
    end while
    return best
end procedure
```

Fig. 1. Pseudo-code of the search algorithm.

an unassigned or assigned variable is initially selected. Typically an unassigned variable is chosen like in backtracking-based search. An assigned variable may be selected when all variables are assigned but the solution is not good enough (for

example, when there are still many violations of soft constraints). Once a variable is selected, a value from its domain is chosen for assignment. Even if the best value is selected (whatever 'best' means), its assignment to the selected variable may cause some hard conflicts with already assigned variables. Such conflicting variables are removed from the solution and become unassigned. Finally, the selected value is assigned to the selected variable.

The algorithm attempts to move from one (partial) feasible solution to another via repetitive assignment of a selected value to a selected variable. During this search, the feasibility of all hard constraints in each iteration step is enforced by unassigning the conflicting variables. The search is terminated when the requested solution is found or when there is a timeout, expressed e.g., as a maximal number of iterations or available time being reached. The best solution found is then returned.

The above algorithm schema is parameterized by several functions, namely

- the termination condition (function *canContinue*),
- the solution comparator (function *better*),
- the variable selection (function *selectVariable*) and
- the value selection (function *selectValue*).

Termination Condition. The termination condition determines when the algorithm should finish. For example, the solver should terminate when the maximal number of iterations or some other given timeout value is reached. Moreover, it can stop the search process when the current solution is good enough, e.g., all variables are assigned and/or some other solution parameters are in the required ranges. For example, the solver can stop when all variables are assigned and less than 10% of the soft constraints are violated. Termination of the process by the user can also be a part of the termination condition.

Solution Comparator. The solution comparator compares two solutions: the current solution and the best solution found. This comparison can be based on several criteria. For example, it can lexicographically order solutions according to the number of unassigned variables (smaller number is better) and the number of violated soft constraints.

Variable Selection. As mentioned above, the presented algorithm requires a function that selects a variable to be (re)assigned during the current iteration step. This problem is equivalent to a variable selection criterion in constraint programming. There are several guidelines for selecting a variable [5]. In local search, the variable participating in the largest number of violations is usually selected first. In backtracking-based algorithms, the first-fail principle is often used, i.e., a variable whose instantiation is most complicated is selected first. This could be the variable involved in the largest set of constraints or the variable with the smallest domain, etc.

We can split the variable selection criterion into two cases. If some variables remain unassigned, the "worst" variable among them is selected, i.e., first-fail principle is applied.

The second case occurs when all variables are assigned. Because the algorithm does not need to stop when a complete feasible solution is found, the variable selection criterion for such case has to be considered as well. Here all variables are assigned but the solution is not good enough, e.g., in the sense of violated soft constraints. We choose a variable whose change of a value can introduce the best improvement of the solution. It may, for example, be a variable whose value violates the highest number of soft constraints.

Value Selection. After a variable is selected, we need to find a value to be assigned to the variable. This problem is usually called "value selection" in constraint programming [5]. Typically, the most useful advice is to select the best-fit value. So, we are looking for a value which is the most preferred for the variable and which causes the least trouble as well. This means that we need to find a value with the minimal potential for future conflicts with other variables. For example, a value which violates the smallest number of soft constraints can be selected among those values with the smallest number of hard conflicts.

2.1 Conflict-based Statistics

Conflict-based statistics were successfully applied in earlier works [6, 11]. In our approach, the conflict-based statistics works as an advice in the value selection criterion. It helps to avoid repetitive, unsuitable assignments of the same value to a variable by memorizing conflicts caused by this assignment in the past. In contrast to the *weighting-conflict* heuristics proposed in [11], conflict assignments are memorized together with the causal assignment which impacted them. Also, we propose the presented statistics to be unlimited, to prevent short-term as well as long-term cycles.

The main idea behind conflict-based statistics is to memorize conflicts and prohibit their potential repetition. When a value v_0 is assigned to a variable V_0, hard conflicts with previously assigned variables (e.g., $V_1 = v_1$, $V_2 = v_2$, ... $V_m = v_m$) can occur. These variables V_1,...V_m have to be unassigned before the value v_0 is assigned to the variable V_0. These unassignments, together with the reason for their unassignment (e.g., assignment $V_0 = v_0$), and a counter tracking how many times such an event occurred in the past, is stored in memory.

Later, if a variable is selected for an assignment again, the stored information about repetition of past hard conflicts can be taken into account, e.g., in the value selection heuristics. For example, if the variable V_0 is selected for an assignment again, we can weight the number of hard conflicts created in the past for each possible value of the variable. In the above example, the existing assignment $V_1 = v_1$ can prohibit the selection of the value v_0 for the variable V_0 if there is again a conflict with the assignment $V_1 = v_1$.

Conflict-based statistics is a data structure that memorizes hard conflicts which have occurred during the search together with their frequency (e.g., that assignment $V_0 = v_0$ caused c_1 times an unassignment of $V_1 = v_1$, c_2 times of $V_2 = v_2$ ... and c_m times of $V_m = v_m$). More precisely, it is an array

$$Stat[V_a = v_a, V_b \neq v_b] = c_{\mathrm{ab}},$$

saying that the assignment $V_a = v_a$ caused c_{ab} times unassignment of $V_b = v_b$ in the past. Note that in case of n-ary constraints (where $n > 2$), this does not imply that the assignments $V_a = v_a$ and $V_b = v_b$ cannot be used together. The proposed conflict-based statistics does not actually work with any constraint, it only memorizes the unassignments together with the assignment which caused them. Let us consider a variable V_a selected by `selectVariable` function, a value v_a selected by `selectValue`. Once an assignment $V_b = v_b$ is selected by CONFLICTING_VARIABLES to be unassigned, the array $Stat[V_a = v_a, V_b \neq v_b]$ is incremented by one.

The data structure is implemented as a hash table, storing information for conflict-based statistics. A counter is maintained for the tuple $A = a$ and $B \neq b$. This counter is increased when the value a was assigned to the variable A and b needed to be unassigned from B. The example of this structure

$$A = a \Rightarrow \begin{cases} 3 \times B \neq b \\ 4 \times B \neq c \\ 2 \times C \neq a \\ 120 \times D \neq a \end{cases}$$

expresses that variable B lost its assignment b three times and its assignment c four times, variable C lost its assignment a two times and D lost its assignment a 120 times, all because of later assignments of value a to variable A. This structure is being used in the value selection heuristics to evaluate existing conflicts with the assigned variables. For example, if there is a variable A selected and if the value a is in conflict with an assignment $B = b$, we know that a similar problem has already occurred $3\times$ in the past, and the conflict $A = a$ is weighted with this number.

For example, a *min-conflict* value selection criterion, which selects a value with the minimal number of conflicts with the existing assignments, can be easily adapted to a *weighted min-conflict* criterion. The value with the smallest sum of the number of conflicts multiplied by their frequencies is selected.

Stated in another way, the weighted min-conflict approach helps the value selection heuristics to select a value that might cause more conflicts than another value, but these conflicts occurred less frequently, and therefore they have a lower weighted sum. This can considerably help the search to get out of a local minimum.

We plan to study the following extensions of the conflict-based statistics:

- If a variable is selected for an assignment, the above presented structure can also tell how many potential conflicts a value can cause in the future. In the above example, we already know that four times a later assignment of $A = a$ caused that value c was unassigned from B. We can try to minimize such future conflicts by selecting a different value of the variable B while A is still unbound.
- The memorized conflicts can be aged according to how far they have occurred in the past. For example, a conflict which occurred 1000 iterations ago can have half the weight of a conflict which occurred during the last iteration or it can be forgotten at all.

Let us study the space complexity of the above data structure for storing conflict-based statistics. At a maximum, there could be a counter for each pair of possible assignments $V_a = v_a$ and $V_b = v_b$, where $V_a \neq V_b$ and there is a constraint between variables V_a and V_b which can prohibit concurrent assignments $V_a = v_a$ and $V_b = v_b$. However, note that each increment of a counter in the statistics means an unassignment of an assigned variable. Therefore each counter $Stat[V_a = v_a, V_b \neq v_b] = n$ in the statistics means that there was an assignment $V_b = v_b$ which was unassigned n times when v_a was assigned to V_a. Together with the fact, that there is only one assignment done in each iteration, the following invariant will be always true during the search: The total sum of all counters in the statistics plus the current number of assigned variables is equal to the number of processed iterations. Therefore, if the above described hash table (which is empty at the beginning and does not contain empty counters) is used, the total number of all counters in it will never exceed the number of iterations processed so far.

The presented approach can be successfully used in other search algorithms as well. For example, in the local search, we can memorize the assignment $V_x = v_x$, which was selected to be changed (re-assigned). A reason for such selection can be retrieved and memorized together with the selected assignment $V_x = v_x$ as well. Note that typically an assignment which is in a conflict with some other assignments is selected.

Furthermore, the presented conflict-based statistics can be used not only inside the presented algorithm. Its constructed 'implications' together with the information about frequency of their occurrences can be easily used by users or by some add-on deductive engine to identify inconsistencies[3] and/or hard parts of the input problem. The user can then modify the input requirements in order to eliminate the found problems and let the solver continue to search with such a modified input problem.

2.2 Maintaining Arc Consistency Using Explanations

Because the presented algorithm works with partial feasible solutions, it can be easily extended to dynamically maintain arc consistency during the search. This can be done by using well known dynamic arc consistency (MAC) algorithms (e.g., by AC|DC algorithm published in [16] or DnAC6 published in [4]) which are widely used in Dynamic CSPs [12].

Moreover, since the only constraints describing assignments (constraint *Variable=value*) can be added and removed during the search, approaches based on explanations [9, 10] can be used as well. In this section, we present how these explanations, which are traditionally used in systematic search algorithms, can be used in our iterative forward search approach in order to maintain arc consistency.

An explanation, $V_i \neq v_i \Leftarrow (V_1 = v_1 \& V_2 = v_2 ... \& V_j = v_j)$ describes that the value v_i cannot be assigned to the variable V_i since it is in a conflict with

[3] Actually, this feature allows to discover all inconsistent inputs during solving the Purdue University timetabling problem [17].

the existing assignments $V_1 = v_1$, $V_2 = v_2$, ... $V_j = v_j$. This means that there is no complete feasible assignment containing assignments $V_1 = v_1$, $V_2 = v_2$, ... $V_j = v_j$ together with the assignment $V_i = v_i$ (these equalities form a no-good set [9]).

During the arc consistency maintenance, when a value is deleted from a variable's domain, the reason (forming an explanation) can be computed and attached to the deleted value. Once a variable (say V_x with the assigned value v_x) is unassigned during the search, all deleted values which contain a pair $V_x = v_x$ in their explanations need to be recomputed. Such value can be either still inconsistent with the current (partial) solution (a different explanation is attached to it in this case) or it can be returned back to its variable's domain. Arc consistency is maintained after each iteration step, i.e., the selected assignment is propagated into the not yet assigned variables. When a value v_x is assigned to a variable V_x, an explanation $V_x \neq v'_x \Leftarrow V_x = v_x$ is attached to all values v'_x of the variable V_x, different from v_x.

In the case of forward checking, computing explanations is rather easy. A value v_x is deleted from the domain of the variable V_x only if there is a constraint which prohibits the assignment $V_x = v_x$ because of the existing assignments (e.g., $V_y = v_y$,..$V_z = v_z$). An explanation for the deletion of this value v_x is then $V_x \neq v_x \Leftarrow (V_y = v_y \& ... V_z = v_z)$, where $V_y = v_y \& ... V_z = v_z$ are assignments contained in the prohibiting constraint. In case of arc consistency, a value v_x is deleted from the domain of the variable V_x if there is a constraint which does not permit the assignment $V_x = v_x$ with other possible assignments of the other variables in the constraint. This means that there is no support value (or combination of values) for the value v_x of the variable V_x in the constraint. An explanation is then a union of explanations of all possible support values for the assignment $V_x = v_x$ of this constraint which were deleted. The reason is that if one of this support values is returned to its variable's domain, this value v_x may be returned as well (i.e., the reason for its deletion has vanished, a new reason needs to be computed).

As for the implementation, the above described algorithm schema (see Fig. 1) can remain as it is, we only need to enforce arc consistency of the initial solution and to extend UNASSIGN and ASSIGN methods. Procedure ASSIGN(solution, variable, value) should enforce arc consistency of the solution with the selected assignment *variable=value* and the procedure UNASSIGN(solution, variable, value) should 'undo' the assignment *variable=value*. It means that explanations of all values which were deleted and which contain assignment $variable = value$ in their explanations needs to be recomputed. This can be done via returning all these values into their variables' domains followed by arc consistency maintenance over their variables.

Using the presented explanations-based approach gives us more flexibility than dynamic arc consistency algorithms (e.g., AC|DC, DnAC, ...) where the value selection function can choose only among the values in the current domain of the variable, i.e., among the values that are not pruned by arc consistency. The values which were deleted via MAC can be selected as well (actually they are

only marked as no-good values in the implementation). If a deleted variable is selected, it can become feasible by repeatedly unassigning a selected value from its explanation until the value is returned to the selected variable's domain. This cannot be done as easily as in the case of dynamic arc consistency algorithms, since we do not know the cause of deletion of a deleted value. For instance, there are several possibilities how to treat a case when there is a variable with an empty domain (i.e., all its values were deleted via MAC). We discuss two of them below, see Sec. 4. Note that we might want to compute the largest feasible solution (in the number of assigned variables) in case of over-constrained problem.

3 IFS as Dynamic Backtracking with MAC

In this section, we describe how the presented iterative forward search framework can be used for modeling of dynamic backtracking (DB) search with the arc consistency maintenance (MAC). In some sense, the presented IFS algorithm with MAC can be seen as an extension of DB with MAC, e.g., described in [10], towards the local search based methods.

Dynamic backtracking with MAC can come out of the above presented IFS with MAC via the following modifications and/or restrictions:

- Variable selection function *selectVariable* always returns an unassigned variable. If there are one or more variables with empty domains, one of them is returned in the variable selection function.
- Value selection function *selectValue* always returns a value from the selected variable's domain (i.e., not-deleted value), if there is no such value, it returns **null**.
- When all the variables are assigned the solver terminates and returns the found solution (termination condition function *canContinue*). In case of **branch&bound** technique an existence of a complete solution should lower the bound so that a conflict arises, which leads to some unassignments.
- If the selected value is **null** (which means that the selected variable has an empty domain), a union of all assignments which prohibits all the values of the selected variable (a union of assignments of all values' explanations) is computed. The last made assignment of them is selected (each variable can memorize an iteration number, when it was assigned for the last time). This assignment has to be unassigned, all other assignments from the computed union are taken as an explanation for this unassignment. If the computed explanation is empty (e.g., $V_x \neq v_x \Leftarrow \varnothing$), the value can be permanently removed from its variable's domain because it can never be a part of a complete solution. If the computed union is empty, there is no complete solution and the algorithm returns **fail**.
- If a value v_x is assigned to a variable V_x, an explanation $V_x \neq v'_x \Leftarrow V_x = v_x$ is attached to all values from the domain of the variable V_x different from v_x. Note that in contrast to the IFS MAC algorithm described in the previous section, the already deleted values from the variable's V_x domain leave its original explanation (it is not changed to $V_x \neq v'_x \Leftarrow V_x = v_x$). This can be

done, because the last assigned variable from the variables which prohibits values from a variable with an empty domain is always unassigned.

Like in the above presented IFS MAC algorithm, arc consistency maintenance and its undo is called automatically after each assignment and unassignment, respectively.

4 Experiments

The above described algorithm together with its presented extensions has been implemented in Java. It contains a general implementation of the iterative search algorithm. The general solver operates over abstract variables and values with a selection of available extensions, basic general heuristics, solution comparators, and termination functions. It may be customized to fit a particular problem (e.g., as it has been extended for Purdue University timetabling, see Sec. 4.2) by implementing variable and value definitions, adding hard and soft constraints, and extending the parametric functions of the algorithm. The results presented here were computed on 1GHz Pentium IIIm PC running Windows 2000, with 512 MB RAM and J2SDK 1.4.2.

Because we attempt to solve large scale problems, maintaining arc consistency is based on AC3 algorithm (e.g., see [19]). In the Purdue University timetabling problem we have almost 830 variables (there is a variable for each course) with the total number of more than 200,000 values (there is a value for each location of a course in the timetable, including a selection of time(s), room and instructor). Furthermore, nearly every two variables are related by some constraint, e.g., typically there is at least one room they can both use. Due to the memory reasons, this prohibits any consistency method which is based on memorizing values for each pair of values or for each pair of value and variable.

In the following experiments we compare several mutations of the above presented algorithm and its improvements. For all these variants, an unassigned variable is selected randomly and the value selection is based on min-conflict strategy. This means that a value is randomly selected among the values whose assignment will cause the minimal number of conflicts with the existing assignments. The search is terminated when a complete solution is found or when the given time limit is reached. As for the solution comparator, a solution with the highest number of assigned variables is always selected. The compared algorithms are:

- **IFS MCRW** ... min-conflict selection of values with 2% random walk[4]
- **IFS TABU** ... tabu list of the length 20 is used to avoid cycling[5]
- **IFS ConfStat** ... min-conflict value selection where conflicts are weighted according to the conflict-based statistics (as described in Sec. 2.1)

[4] With the given probability, a value is selected randomly from all values of the selected variable's domain.

[5] Repeated selection of the same pair $(variable, value)$ is prohibited for the given number of following iterations.

– **IFS MAC** ... arc-consistency maintenance; if there is a variable with an empty domain, a variable which caused a removal of one or more of values is selected and unassigned.[6]
– **IFS MAC+** ... arc-consistency maintenance; the algorithm continues extending the solution even when there is a value with an empty domain. If the selected variable has an empty domain (pruned by MAC) then one of values deleted by MAC is selected (via min-conflic value selection).
– **DBT MAC** ... dynamic backtracking algorithm with arc consistency maintenance (as described in Sec. 3)
– **DBT FC** ... dynamic backtracking algorithm with forward checking

4.1 Random CSP

In this section, we present results achieved on the Random Binary CSP with uniform distribution [3]. A random CSP is defined by a four-tuple (n, d, p_1, p_2), where n denotes the number of variables and d denotes the domain size of each variable, p_1 and p_2 are two probabilities. They are used to generate randomly the binary constraints among the variables. p_1 represents the probability that a constraint exists between two different variables (tightness) and p_2 represents the probability that a pair of values in the domains of two variables connected by a constraint are incompatible (density).

Figure 2 presents the number of assigned variables in percentage to all vari-

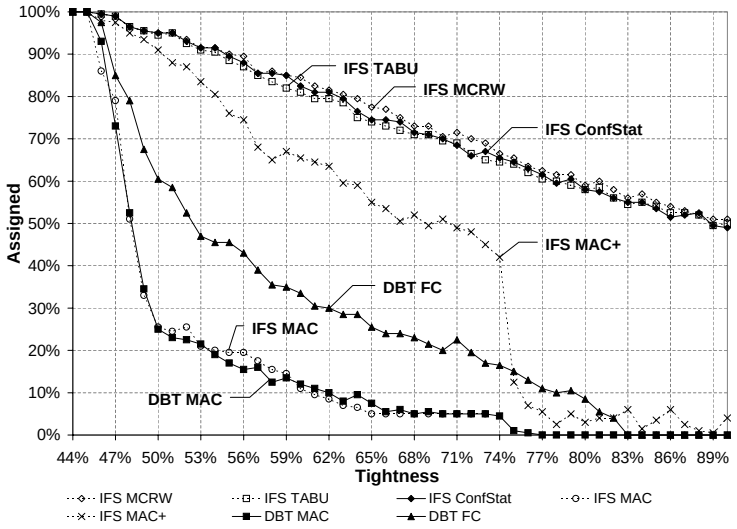


Fig. 2. $CSP(20, 15, 43\%, p_2)$, number of assigned variables (in percentage to all variables), the best achieved solution within 60 seconds, average from 10 runs.

[6] This is done so that a value v_x of such a variable with an empty domain V_x is selected randomly and a randomly selected assignment from the explanation $V_x \neq v_x$ is unassigned.

ables wrt. the probability p_2 representing tightness of the generated problem $CSP(20, 15, 43\%, p_2)$. The average values of the best achieved solutions from 10 runs on different problem instances within the 60 second time limit are presented.

Each of the compared algorithms was able to find a complete solution within the time limit for all the given problems with a tightness under 46%. Achieved results from min-conflict random walk, tabu-list and the presented conflict-based statistics seem to be very similar for this problem. Also, it is not surprising that a usage of consistency maintenance techniques lowers the maximal number of assigned variables, e.g., both dynamic backtracking with MAC and IFS with MAC extend an incomplete solution only when it is arc consistent with all unassigned variables. As we can see from the Figure 2, we can get much better results when we allow the search to continue even if there is a variable with an empty domain.

For the results presented in Figure 3, we turned the random CSP into an op-

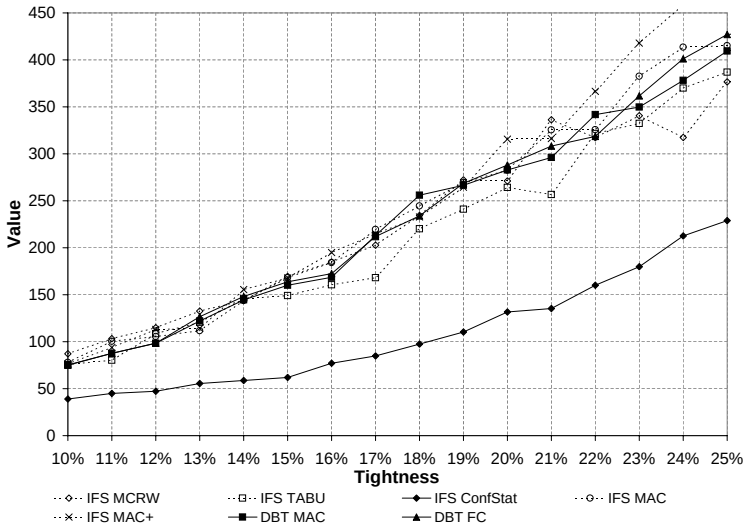


Fig. 3. $minCSP(40, 30, 43\%, p_2)$, the sum of all assigned values of the best solution within 60 seconds wrt. problem tightness, average from 10 runs.

timization problem where we are searching for a complete feasible solution with the smallest total sum of the assigned values. Recall that each variable has d generated values from $0, 1, ...d-1$. For the comparison we used $CSP(40, 30, 43\%, p_2)$ problems with the tightness p_2 taken so that every measured algorithm was able to find a complete solution for each of 10 different generated problems within the given 60 second time limit. The min-conflict value selection criterion was adapted to take the smallest value from the values which cause the minimum number of conflicts. For DBT, the smallest value from a domain of a selected variable was taken as well. As for conflict-based statistics, each value is weighted by itself added to the number of weighted conflicts. For example, value 7 with 4 conflicts has a weight $7 + 4 = 11$. If value 3 has 9 conflicts, so its weight is 12, then the value 7 will be preferred over 3. For all the measured algorithms,

the solver continues even if a complete solution is found until the time limit is reached. A complete solution with the smallest sum of the assigned values is then returned. For this problem, the presented conflict-based statistics was able to give much better results than other compared algorithms. The algorithm is obviously trying to stick much more with the smallest values than the others, but it is able to find a complete solution since the conflict counters are rising during the search. Such behaviour can be very handy for many optimization problems, especially when optimization criteria (expressed either by some optimization function or by soft constraints) go against the hard constraints.

Conclusion. For the tested random constraint satisfaction problems, IFS MAC does not seem to be suitable: it produces very similar results to DBT MAC, but unlike DBT MAC, it can not guarantee finding a complete solution, if one exists, or to prove that the problem has no solution.

Since IFS MAC+ can continue extending a partial solution even when there is a variable with an empty domain, it can produce much better results than IFS MAC in case that the given problem is over-constrained (i.e., there is no complete solution). For some tasks, it can be an interesting compromise between backtrack-based search and local search.

For pure, not-optimization constraint satisfaction problems (e.g., results from Fig. 2), the presented IFS with conflict-based statistics returns very similar results to other tested traditional local search principles preventing cycles (i.e., tabu-list and random walk). However the presented conflict-based statistics can be very useful when optimization criteria are considered.

4.2 Real-Life Application: Purdue University Timetabling

In this section, we present some results achieved on the large lecture timetabling problem from Purdue University. The following tests were performed on the complete Fall 2004 data set[7] which consists of 826 classes (forming 1782 meetings, 4050 half-hours) that must fit into 50 lecture rooms with capacities up to 474 students. In this problem, 89,633 course demands of 29,808 students must be considered.

The timetable maps classes (students, instructors) to meeting locations and times. A primary objective is to minimize the number of potential student course conflicts which occur during this process. Other major constraints on the problem solution are instructor availability and a limited number of rooms with sufficient capacity, specific equipment, and a suitable location. Some of these constraints must be satisfied; others are introduced within an optimization process in order to avoid an over-constrained problem.

Figure 4 presents some results achieved with the presented framework. Average values together with their RMS (root-mean-square) variances of the best achieved solutions from 10 different runs found within 30 minute time limit are presented. *Time* refers to the amount of time required by the solver to find the presented solution.

[7] Similar results were achieved also on a complete data set from Fall 2001, used in our previous work [17].

Test cases	IFS ConfStat	IFS TABU	IFS MCRW
Assigned variables [%]	100.00 ± 0.00	97.67 ± 0.15	98.29 ± 0.16
Time [min]	24.11 ± 4.42	24.17 ± 3.62	24.52 ± 3.83
Student conflicts [%]	1.97 ± 0.06	1.97 ± 0.07	2.05 ± 0.19
Preferred time [%]	85.64 ± 1.57	89.86 ± 0.69	89.63 ± 1.06
Preferred room [%]	50.39 ± 5.34	66.48 ± 3.42	64.84 ± 3.86

Fig. 4. Purdue University Timetable, characteristics for the best achieved solutions within 30 minutes, average values and RMS variances from 10 runs.

The value selection criterion was extended to take into account three optimization criteria. *Student conflicts* give the percentage of unsatisfied requirements for the courses chosen by the students. One student conflict means that there are two courses required by a student that cannot be both attended, e.g., because they overlap in time. *Preferred time* and *preferred room* estimate the satisfaction of time and room preferences respectively. These preferences are given by the instructors individually for each class. In our heuristics, room preferences are considered much less important than time preferences and student conflicts.

IFS with conflict-based statistics was able to find a complete solution of a good quality in each run; the first complete solution was found after $6 - 10$ minutes. Moreover, it was able to significantly improve the first complete solution with approximately 2.3% of violated student requirements and 80% of time preferences satisfied. On the other hand, neither tabu search nor min-conflict random walk were able to find any complete solution within the given 30 minute time limit; at least 17 variables for TABU and 12 variables for MCRW remained unassigned after each run.

Dynamic backtracking with either MAC or FC (not included in Figure 4) was able to assign in average approximately 93% of variables, with almost 3% violated student requirements and only approximately 40% of time preferences satisfied. IFS MAC was able to assign only about 65% of variables. IFS MAC+ assigned about 94% variables, with approximately 2.2% student conflicts and around 75% of time preferences. Consistency was maintained over all hard constraints.

We plan to use MAC+ only over the "additional" constraints, e.g. a precedence constraint between two or more courses or not-overlap constraint between a lecture and its seminars. However, the used data set contains only 203 of such constrains, so there is no significant difference between a solution with and without MAC+ (IFS ConfStat versus IFS ConfStat with MAC+ on additional constraints). Currently we work on solving other Purdue University timetabling problems where the number of these constraints significantly increases.

Conclusion. The general consensus, that local search is more suited for optimization problems than backtrack-based search, is valid also for our large lecture timetabling problem. Unlike the other tested algorithms, the presented conflict-based statistics is capable to produce high quality and stable results. Furthermore, if there is any time available after the first complete solution is found, the solver is also able to gradually improve this solution. We believe that the arc

consistency maintenance can help us solve some complicated situations which can arise in our timetabling problem.

5 Conclusions and Future Work

We have presented a promising iterative forward search framework which is, as we believe, together with the presented improvements, capable of solving various constraint optimization problems. We have presented some results on random CSP and on Purdue University timetabling problem. Our solver is able to construct a demand-driven timetable, which satisfies approximately 98% course requests of students together with about 85% of time preferences.

Our future research will include extensions of the proposed general algorithm together with improvements to the implemented solver. We would like to do an extensive study of the proposed framework and its possible application to other, non timetabling-based problems. As for Purdue University timetabling, we are currently extending the CLP solver [17] with some of the features included here to present a fair comparison. However, the CLP solver was not yet able to find a complete solution in the accomplished preliminary experiments.

Acknowledgements

This work is partially supported by the Czech Science Foundation under the contract No. 201/04/1102 and by Purdue University.

References

[1] C. Bessière and J. C. Régin. Arc consistency for general constraint networks: Preliminary results. In *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 398–404, Nagoya, Japan, 1997.

[2] C. Bessière and J. C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings IJCAI'01*, pages 309–315, Seattle WA, 2001.

[3] Christian Bessière. Random uniform csp generators, 1996. `http://www.lirmm.fr/~bessiere/generator.html`.

[4] R. Debruyne. Arc-consistency in dynamic csps is no more prohibitive. In *Proceedings of 8th Conference on Tools with Artificial Intelligence (TAI'96)*, pages 299–306, 1996.

[5] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[6] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002.

[7] Philippe Galinier and Jin-Kao Hao. Tabu search for maximal constraint satisfaction problems. In *Proceedings 3rd International Conference on Principles and Practice of Constraint Programming*, pages 196–208. Springer-Verlag LNCS 1330, 1997.

[8] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, pages 23–46, 1993.

[9] Narendra Jussien. The versatility of using explanations within constraint programming. In *Habilitation thesis of Universit de Nantes*, France, 2003.

[10] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming*, pages 249–261, 2000.

[11] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, 2002.

[12] Narendra Jussien and Gérard Verfaillie. Dynamic constraint solving. In *A tutorial including commented bibliography presented at CP 2003*, Kinsale.

[13] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2000.

[14] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[15] Tomáš Muller and Roman Barták. Interactive timetabling: Concepts, techniques, and practical results. In *PATAT 2002 — Proceedings of the 4th international conference on the Practice And Theory of Automated Timetabling*, pages 58–72, 2002.

[16] B. Neveu and P. Berlandier. Maintaining arc consistency through constraint retraction. In *Proceedings of the IEEE International Conference on Tools for Artiocial Intelligence (TAI)*, pages 426–431, New Orleans, LA, 1994.

[17] Hana Rudová and Keith Murray. University course timetabling with soft constraints. In *Practice And Theory of Automated Timetabling, Selected Revised Papers*, pages 310–328. Springer-Verlag LNCS 2740, 2003.

[18] Andrea Schaerf. Combining local search and look-ahead for scheduling and constraint satisfaction problems. In *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1254–1259, Nagoya, Japan, 1997.

[19] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

Stochastic solver for constraint satisfaction problems with learning of high-level characteristics of the problem topography

Yehuda Naveh

IBM Research Labs in Haifa, Haifa University, Haifa 31905, Israel
naveh@il.ibm.com

Abstract. A new generic method for solving constraint satisfaction problems is presented. The method is based on stochastic search which is non-local in the sense that at any point in the search, the next state sampled may be arbitrarily far from the current state. The solver relies heavily on knowledge of the high-level characteristics of the topography defining the problem. It obtains these characteristics in two complementary ways. First, it continuously and automatically learns the topography in the course of the search. Second, it accepts domain-knowledge from the user if such knowledge is available. We describe experiments performed with this solver on a set of hard arithmetical CSPs adopted from practical hardware verification problems, as well as on the well-known low-autocorrelation binary sequence problem. The experiments confirmed the strength of the method.

1 Introduction

1.1 Background

Constraint satisfaction problems appear in many knowledge domains from staffing and scheduling, to radio-frequency partitioning, to functional verification of hardware designs. A constraint satisfaction problem (CSP) is a triplet (V, D, C) consisting of a set of variables V, a corresponding set of domains D, and a set of constraints C on the variables in V. A solution to a CSP is a complete assignment to V from domains D such that all constraints in C are satisfied. A CSP solver is an algorithm that accepts a CSP as input and returns one of three outputs: (1) a solution to the CSP, (2) '`unsat`' if there is no such solution, or (3) '`time-out`'. Existing CSP solvers fall into two basic categories: systematic search solvers and stochastic search solvers.

1.2 Systematic Search

Systematic solvers typically move between partial assignments to V, at each step either adding an assignment to a not-yet-assigned variable (normal step), or removing assignments from already-assigned variables (backtrack). The strength

of such solvers stems from their ability to reduce ('prune') large fractions of the variable domains by enforcing some level of consistency on the problem after each assignment to a variable.

In many cases, systematic algorithms are very effective. However, this framework is not suitable for some broad classes of CSPs. One such class is the case where the size of the domain of some of the variables remains very large even after enforcing consistency. The time-complexity in this case is of the order of the size of the domain divided by the number of satisfying assignments. In some cases this ratio is exponentially large. Another class of CSPs that is hard for systematic search involves cases where consistency cannot be enforced efficiently.

1.3 Stochastic Search and Topographic View of the CSP

The second class of solvers are complete-assignment solvers, *e.g.*, min-conflicts [17], GSAT [24], Walksat [22], or Tabu Search [11, 6]. Here, the solver does not move between partial assignments to V, but rather between complete assignments. Since searching all complete assignments may be a prohibitively large task, the solver must decide which complete assignments to check. In order to do so, two metrics are invariably defined: first a 'distance' is defined between any two complete assignments. This distance satisfies the ordinary requirements on distance relations (symmetry, positive definiteness, and triangle relation). Second, a 'cost' is defined for every complete assignment. The cost satisfies one strong requirement and one weak requirement. The strong requirement states that the cost of a complete assignment is non-negative, and it is zero if and only if the assignment satisfies the CSP. The weak requirement is for some fitness-distance correlation [4] to exist between the cost and the distance.

The two metrics define a topography whose hyper-plane represents the set of all complete assignments to V, and whose elevation is provided by the cost of each complete assignment. Thus, the CSP is mapped into an optimization problem, with the difference being that we are only interested in reaching a global minimum of value 0 (or proving that such a minimum does not exist).

As with any optimization problem, the problem reduces to the ability to escape local minima in the course of the search. In particular, CSPs that can be mapped into topographies with few or no local minima are easily solvable within such an approach. Most practical problems, however, are mapped into topographies that contain a substantial number of local minima (see [8] for a detailed analysis of the topographies of random SAT problems). Various broad strategies have been created in order to escape the local minima. In some cases (*e.g.,* GSAT), the solver checks a local neighborhood of the current complete assignment, and chooses a new complete assignment at random out of all checked assignments that have the lowest cost. In others (*e.g.*, simulated annealing [14], random-walk [23]), the solver makes a random decision about whether to move to a neighboring complete assignment even if it has a higher cost than the current one. All those strategies share two characteristics. First, part of the decision about which complete assignments to explore is made stochastically. Second, in order for these strategies to work, the move from one complete assignment to

the next is done locally, *i.e.*, involving a distance that is much smaller than the typical size of the full problem (although may be larger than the minimal 'single assignment' distance, as in variable-neighborhood search [12]). In fact, these two characteristics are so prevalent that 'stochastic methods' and 'local-search methods' have become the common names for search methods going through complete assignments.

Stochastic local-search methods are suitable for some classes of problem topographies. For example, simulated annealing works best in the case where sequences of asymmetric local minima surrounded by barriers of gradually decreasing heights eventually lead to global minima [14]. In general, however, many practical topographies are hard for any of these methods. Most critically, local-search methods are usually inefficient on problems for which the typical barriers surrounding local minima are wide and high [10].

In this paper, we present a CSP solver that relies on a different strategy for escaping local minima. Through automatic learning, the solver attempts to gain high-level information about the topography of the problem. It applies this information when moving from one complete assignment to another. These moves are non-local in nature, and hence may result in efficient solutions even for problems characterized by wide or high barriers. Learning was previously applied to stochastic local search only in the 'cost' direction – see, *e.g.*, [10, 7]. In these works, which deal with SAT problems, clauses that are found to be hard to satisfy have their costs increased during the course of the solution. However, learning of the problem characteristics in the planar directions is significantly more complex, and is discussed here for the first time.

In order to appreciate how learning the characteristics of the topography may be beneficial in terms of solvability, consider the following hypothetical example. Suppose that the problem topography is composed of many regions. Each of these regions is similar in its high-level structure to all other regions - and is characterized by many local minima separated by roughly the distance L_1 from each other. In addition, the regions are separated from each other by roughly $L_2 \gg L_1$ (so the spectral composition of the topography is characterized by two values in any of a number of directions). Finally, suppose that the number of global minima (*i.e.*, solutions) is small compared to the number of regions. During the solution process, any stochastic search algorithm is likely to reach the quasi-global minimum of one of the regions (the point with the lowest cost in that region). However, once stuck in such a minimum, only a large hop of the order of L_2 may be of any benefit. Therefore, by learning the length-scales characterizing the topography, the algorithm may bias its moves towards ones which are multiples of L_2, and by doing so critically enhance its ability to escape macroscopically large but infeasible regions.

We conjecture that many CSPs of practical origin can be represented by topographies which feature well defined high-level characteristics as in the above example. The results of section 3 support this conjecture.

2 Simulated Variable Range Hopping

2.1 Terminology

In what follows, we consider a CSP (V, D, C), where each variable in V is binary-encoded into a bit string, the domains D are represented as unary constraints on V, and each constraint (either from C, or implied by D) is defined by a cost function. With this view of the CSP, we use the following terminology:
bit size: Total number of bits representing all variables.
state: A particular assignment from $\{0, 1\}$ to all bits in the problem. A state is a complete assignment of values to all variables in V, not necessarily out of D.
current state: The state the solver is currently at, used for reference with other states.
cost: The total cost of a given state. This is the sum of costs of all constraints on that state. The cost is zero if and only if the state is a solution of the CSP.
current cost: the cost of the current state.
flip: A change of the value of a single bit.
attempt: A set of bits, usually in the context of flipping all bits in the set, and comparing the cost of the resulting state with the current cost.
step size: Number of bits in an attempt.
successful attempt: an attempt which leads to a cost lower than the current cost.
attempt list: A set of attempts.
difficulty: An integer characterizing how difficult it is to escape the current state.
hop: A move from one state to another.
distance: The Hamming distance between two given states.
topography characteristic: Any of a given set of characteristics used by the solver to characterize the topography. Examples are 'length scales' and 'preferred directions'.
attempt type: A string characterizing the origin of a given attempt. Attempt types are "stochastic", "user-defined", and an additional 'learned' type for each topography characteristic (*e.g.*, "learned-step-size", "learned-direction").

2.2 Outline of the Algorithm

In broad terms, the heart of the solver works as follows: At each round, an attempt list is chosen according to the procedure outlined in the next paragraph. All successful attempts in the list are saved (without distorting the current state), and their characteristics are learned. Then the state hops to a better state according to one of the successful attempts with the lowest cost, and the next attempt list is chosen and similarly processed. If at any stage a state with zero cost is found, the solver terminates with success. On the other hand, if no successful attempt was found in the list, the difficulty is raised, which means that the next round would try a larger number of attempts from the same state.

The wisdom of the solver is encapsulated in the way it chooses attempt lists. This choice relies heavily on the learned characteristics of the topography, on available domain-knowledge from the user, and on randomness. When choosing an attempt, the solver first decides which attempt type it wants. This high-level choice is itself based on previous learning. Still, even when past experience has shown that all successful attempts were of the 'learned' or 'user-defined' types, the solver never completely abandons purely random attempts.

As in previous works [9, 20, 16, 13], our modeling scheme is object-oriented, and definitions of domain-knowledge, as well as cost calculations and initialization policies, are encapsulated within independent user-defined structures.

The learning mechanism we implemented is simple—the values of all characteristics of successful attempts are saved, and when the solver needs a learned value of some characteristic it chooses randomly from the vicinity of all saved values. The random choice is weighted according to the frequency of occurrence of the value. So, if all past successful attempts had the following step sizes (1, 1, 1, 1, 5, 5, 15, 15), then, when the solver creates an attempt of type "learned-step-size", the new attempt has a probability 0.5 to have a step-size close to 1, and probability 0.25 each to have a step-size close to 5 or close to 15 (in our experiments, the 'close to' parameter was set to 1, so the chosen step-size may be any of the set of values $\{1, 2, 4, 5, 6, 14, 15, 16\}$). Note that while simple, this mechanism provides for a positive feedback loop and therefore strengthens successful characteristic values. The mechanism also requires saving parameters of only successful hops, which means that space-complexity, as well as time-complexity in maintaining the learned lists, do not pose a problem. All results reported below were based on learning of two topography parameters: characteristic length scales and preferred directions (through correlated bits).

The step sizes of attempts are not limited. In fact, in the example topography described in Section 1.3, we expect attempts of step sizes close to L_1 and L_2 to be relatively successful. Therefore, the solver is automatically biased towards further choosing a relatively large number of those step sizes. Hence the non-locality of the solver. Note also that a hop from one state to another is performed only if the new state is of lower cost than the original. Hence, the solver is strictly zero-temperature[1].

Our approach to the problem is reminiscent in many ways of the well known physical problem of an excited localized electron in a disordered material at zero-temperature [19]. In both cases, many meta-stable states are formed between high barriers, the positions of those states do not form a regular array, and their costs (energies) are distributed more or less uniformly on some energy scale. In both problems, the zero-cost (ground) states are scarce compared to the number of meta-stable states, and in general there may be a strong mixture of high and low-cost states in the same spatial vicinity. There are also similarities between the dynamics of the physical electron and the dynamics of the current state in

[1] In some cases (depending on previous statistics, and according to predefined heuristics), we restart the algorithm, while keeping all previously-learned values. Such restarts may be viewed as finite-temperature moves, but in practice they are rare.

our case. First, in both cases, the state evolves only through cost-reducing hops. Second, it can be shown that in both cases, a rapid, linear reduction of the cost is followed by a much slower, sub-linear reduction. Finally, in both problems, hops of various length scales are taken in no predictable order. The dynamics of localized electrons in disordered materials is known as 'variable-range hopping'; therefore, we name this algorithm 'simulated variable range hopping' (SVRH). (One important difference between the two problems is that the physical electron cannot learn the topography or otherwise gain knowledge about it. This is compensated by a very rapid attempt rate compared to the simulated problem[2]).

2.3 Input to the Algorithm

The input to SVRH is twofold: it comprises of the CSP and, possibly, user domain-knowledge.

CSP The CSP (V, D, C) is input to the algorithm in the following way:
(1) Variables V are in the form of bit strings of varying lengths, and (2) constraints C and domains D are both in the form of constraints on V. Each such constraint sees one or more of the input variables. Each constraint is associated with a function Cost() that returns 0 if the state of the variables satisfies the constraint, and a positive integer otherwise. In addition, through a function Initialize(), each constraint has the ability to initialize the variables it sees into values consistent with the constraint.

As an example, consider an integer-equality constraint acting on two variables A and B, where A is encoded to a bit string of length 7, and B to a bit string of length 4. In this case, the function Cost() may return the Hamming distance between the four least significant bits of A and B, plus the number of '1' bits in the three most significant bits of A. The function Initialize() may set both A and B to have the same arbitrary integer value.

User Domain Knowledge User domain knowledge is presented to the solver by implementation of a family of functions (indexed by i) CreateUserAttemptList(i, difficulty d). Each function returns a list of attempts believed by the user to be beneficial in their particular domain. The size of the list (*i.e.*, the number of attempts returned) grows with the difficulty parameter d.

One example for use of user-defined lists is to enforce invariants of the problem. For example, in the queens problem, each variable may represent a single row in the checkerboard (so that each bit in the variable represents a single square, and a '1' bit represents an occupied square). A user function may then return a list of combinations of bits within a variable that, if flipped, each combination will result in exactly a single '1' bit in the variable.

[2] Still, even with an attempt rate of 10^{12} $\sec^{-1}$, an electron may spend months or years until it reaches the ground state in a typical disordered material.

```
algorithm SVRH(V,C,F)
  global V,C,F
  RandomizeVariables()
  for all c in C do
    c.InitializeVariables()
  repeat
    attempt := ChooseBestAttempt()
    Hop(attempt)
  until Cost() = 0
  return V
```

Fig. 1. Algorithm SVRH: accepts as input variables V, constraints C, and a family F of CreateUserAttemptList() functions, and returns variables at a zero-cost state (time-outs are not considered here). Note that constraints may override the default random initialization, as well as each other's initialization. Hence, an heuristic initialization order which initializes the tightest constraints last is sometimes implemented.

```
function Cost()
  totalCost := 0
  for all c in C do
    totalCost := totalCost + c.Cost()
  return totalCost
function Hop(attempt)
  for all bit in attempt do
    σ(bit) := 1 - σ(bit)
function AttemptCost(attempt)
  Hop(attempt)
  cost := Cost()
  Hop(attempt)
  return cost
```

Fig. 2. Basic functions: Cost, Hop, and AttemptCost. AttemptCost() returns the cost of the state reached from the current state by *attempt*. $\sigma(bit)$ is the value of bit *bit*.

2.4 Algorithm

Figures 1–8 show pseudo-code for SVRH. Together with the terminology of Section 2.1, they provide a formal description of the algorithm which was outlined and discussed in Section 2.2. In Figure 1, variables are first initialized, and the state then hops between best attempts, until reaching a zero-cost state. Figure 2 shows some basic functions used extensively by the algorithm. Figures 3–5 show how the solver chooses attempts, including its decision on whether to choose completely stochastic, user-defined, or learned attempts. Figure 6 shows how non-user-defined attempts are created. Finally, in Figures 7 – 8 attempts are tried, and if successful their characteristics are learned.

```
function ChooseBestAttempt()
  difficulty := 0
  bestAttempts := ∅
  while bestAttempts = ∅ do
    bestAttempts := FindBestAttempts(difficulty)
    difficulty := difficulty + 1
  return (a random attempt from bestAttempts)
```

Fig. 3. Function ChooseBestAttempt.

```
function FindBestAttempts(difficulty)
  best := ∅
  for all i in F do
    userAttempts := CreateUserAttemptList(i, difficulty)
    for all attempt in userAttempts do
      best := TryAndLearn(attempt, best, "user-defined")
      for all t in attempt-types except "user-defined" do
        otherAttempts := GetAttemptsPerUserDefinedAttempt(t)
        best := TryAndLearn(otherAttempts, best, t)
  return best
```

Fig. 4. Function FindBestAttempts. User-defined attempts are obtained here from CreateUserAttemptList() functions according to the level of difficulty. Stochastic and learned attempts are then obtained from GetAttemptsPerUserDefinedAttempt() in relative numbers reflecting previous successes of these types of attempts. All attempts are tried and, if successful, learned. We assume here that $F \neq \emptyset$. Otherwise the algorithm takes a slightly different form.

3 Experimentation and Analysis

3.1 General

The two experiments described below made use of learning of two topography characteristics: length scales (manifested in the step-sizes of successful attempts) and preferred directions (manifested in correlation between the bits forming successful attempts).

3.2 Floating Point Unit Verification

Problem, Motivation, and Model This stage of experiments involved a problem taken from the realm of functional hardware verification [3]. Here, we experimented with many CSPs designed for the verification of floating point units of high-end microprocessors [2].

All CSPs contained two variables, A and B, each consisting of 32, 64, or 128 bits. Constraints were of three types: mask (forcing some of the bits to be 0 or 1), range (forcing the variable to be within some range), and number-of-ones (forcing the variable to have exactly some number of bits with value 1, but not specifying which bits they are). Modeling these types of constraints

```
function GetAttemptsPerUserDefinedAttempt(t)
  successT := Previous success rate for attempts of type t
  successUser := Previous success rate for "user-defined" attempts
  if successT ≥ successUser then
    numOfAttempts := min(successT / successUser, K)
    attempts := ∅
    for i := 0 to numOfAttempts do
      attempts := attempts ∪ CreateAttempt(t)
  else
    prob := max(successT / successUser, 1/K)
    attempts := ∅
    With probability prob
      attempts := CreateAttempt(t)
  return attempts
```

Fig. 5. Function GetAttemptsPerUserDefinedAttempt. If the previous success rate of attempts of type t was better then the user-defined attempts, the function returns a corresponding number of these attempts. Otherwise, it returns a single attempt of this type with a corresponding probability. K is a heuristic cutoff, taken to be 20 in our experiments.

```
function CreateAttempt(t)
  if t = "stochastic" then
    stepSize := a random integer smaller than bit-size
    attempt := a random attempt with stepSize bits
  else
    learnVal := a learned value for characteristic t
    attempt := a random attempt with value close to learnVal for t
  return attempt
```

Fig. 6. Function CreateAttempt. See section 2.2 for details on how learned values are chosen, and how the corresponding attempts are generated.

for SVRH (*i.e.*, implementing their cost functions and initialization policies) is straightforward, and is not described here because of space considerations. Any of the three constraints may apply either to A or B, or to the result of some operation[3] between them, *e.g.*, to $A \times B$. For example, a full mask on the result of $A \times B$ reduces to the factorization problem $A \times B = C$, where C is the constant value of the mask. In addition, if we require (by adding an additional constraint) that the number of '1' bits in A is exactly some k, the problem becomes even harder.

This problem is hard for CSP solvers based on systematic search because there is no possibility of efficient pruning. For example, consider the factorization problem. While we have only two variables, there is no fast way to prune either of their domains in order get rid of domain values inconsistent with the constraint $A \times B = C$.

[3] By operations in this section we mean IEEE floating-point operations [2].

```
function TryAndLearn(attempts, bestAttempts, t)
  newBestAttempts := bestAttempts
  currentCost := Cost()
  for all attempt in attempts do
    bestAttempt := choose any attempt from bestAttempts
    bestCost := AttemptCost(bestAttempt)
    cost = AttemptCost(attempt)
    if cost < currentCost then
      successfulTries[t] := successfulTries[t] + 1
      LearnCharacteristics(attempt)
      if cost < bestCost then
        newBestAttempts := ∅
      if cost ≤ bestCost then
        newBestAttempts := newBestAttempts ∪ attempt
    else
      unsuccessfulTries[t] := unsuccessfulTries[t] + 1
  return newBestAttempts
```

Fig. 7. Function TryAndLearn. If an attempt in the list leads to lower cost than the current cost, learn its characteristics. If, in addition, its cost is lower than or equal to the best cost found so far then keep this attempt (if truly lower then discard of all previous best attempts). In any case keep account of how many of the attempts of the input type t where successful and how many were unsuccessful.

```
function LearnCharacteristics(attempt)
  for all tc in topography characteristics do
    value := value of tc for attempt
    add value to the set of learned values corresponding to tc
```

Fig. 8. Function LearnCharacteristics. This function implements the learning mechanism described in section 2.2. For example, for tc = 'length-scales', the corresponding learned value is 'step-size', so the number of bits in *attempt* is added to the set of learned step-size values.

Furthermore, these problems also fall under the class of problems which are hard for local-search methods, as described in Section 1.3. To understand why this is so, consider again the factorization problem. Now, suppose $A \times B$ is relatively close to C (but not equal to it). At this stage changing either A or B by any small value is likely to make the cost (*e.g.*, $|A \times B - C|$) large (because each of A and B are typicaly very large numbers). Hence, in the topographic view, local minima are narrow and contained within high barriers.

Results Our SVRH solver confronted these types of problems with surprising ease. Most problems were solved in a fraction of a second, and the rest were solved within a few minutes. The user-defined input functions we used returned all combinations of bits lying in some vicinity from each other on the same operand, where the vicinity grows with the difficulty parameter. The rationale

behind these functions is that some specific combinations may eliminate the propagation of carry bits, which are responsible for a long range effect on the result of any small disturbance of an operand. However, neither these functions nor the initialization process of Figure 1 were sufficient to solve the problem. In all hard cases, without the random large-step-size hops, and the repetition of similar hops through learning, the problem was not solved.

We summarize this experiment in Tables 1 and 2. In Table 1, we compare the results of the SVRH solver with the results of using zChaff [18], a general purpose, state-of-the-art SAT solver. The translation of the CSP to a SAT problem was done in two stages: transformation of the mathematical expressions into Boolean logic expressions on the bits of the operands, the result, and an auxiliary carry-bit vector, followed by a transformation to CNF-SAT. To our best judgement, this is the most efficient translation scheme possible for this problem. It is clear from Table 1 that under this scheme, SVRH outperforms zChaff by orders of magnitudes. We attribute this large difference in performance to the fact that zChaff is systematic, and is, therefore, inherently unsuitable for this problem. In Table 2, we compare the results of the SVRH solver to zChaff and to two special-purpose solvers specifically designed and maintained to solve CSPs of the form in question[4]. We see that SVRH, despite the fact that it is a general-purpose solver, finds more solutions than either of the SP solvers, as well as than zChaff. This advantage of SVRH over the other solvers is attributed to its unique combination of 'brute force' non-local stochastic search together with learning of advantageous steps in the particularly complex and highly-non-monotonic topography of this problem. (note that in Table 2, the average time per solution is not a good parameter for comparison because the tasks solved by SVRH and not solved by SP1 are considerably harder than the tasks solved by both).

Table 1. Comparison between SVRH and zChaff on a benchmark of 133 satisfiable floating-point unit verification tasks containing multiply operation. All tasks were solved by both engines. All times are in seconds. Problems in this benchmark are not suitable for the special purpose solvers SP1 and SP2.

	SVRH	zChaff
Average solution time	0.97	200.5
Solution time, largest ratio case	0.3	2861
Solution time, smallest ratio case	5.7	25

3.3 Low Autocorrelation Binary Sequences

General In order to further evaluate our solver, it was important to examine its capabilities against well-known benchmarks. Three aspects are relevant in this

[4] The special-purpose solvers are described in [2]. SP1 is the best available solver of its type.

Table 2. Comparison between SVRH, two special-purpose (SP) solvers, and zChaff on a benchmark of 150 satisfiable and unsatisfiable floating-point-unit verification tasks containing various operations. Time-out was 600 seconds. All times are in seconds. Note that since SVRH is incomplete, it succeeded only on satisfiable instances.

	SVRH	SP1	zChaff	SP2
Number of successes	93	88	54	29
Average solution time per success	9	3	65	0.3

evaluation. First, since the SVRH solver is generic, it is important to verify that new problems can be conveniently modeled for its input, so that once a problem is specified, the total bring-up time of the solver on that problem does not exceed a few hours. Second, and again related to the genericity of the solver, it should be verified that new problems do not require changes to the algorithm, and that all solution strategies and heuristics suitable for a given problem may be input to the solver through the constraints' cost and initialization functions, and the user domain-knowledge input functions. Third, we should confirm that the solution abilities of the solver (both in finding solutions, and in run-time performance) are at least comparable to abilities of dedicated solution algorithms.

The evaluation process was performed using benchmark problems from CSPLib [1]. We checked the generic aspects of the solver (bring-up time and possibility to apply heuristics) on many of the problems in [1]. In virtually all cases, bring up time was short, and we were able to incorporate almost all necessary solution strategies through the input functions to the solver.

Problem and Motivation As for evaluation of solution abilities, we chose to concentrate on the 'low autocorrelation binary sequences' (LABS) problem (prob005 in [1]). Here, a sequence of N bits, each with value $\sigma_i \in \{+1, -1\}$, is considered, and a minimum over all configuration of bits is sought for the autocorrelation parameter $E = \sum_{k=1}^{n-1} C_k^2$, where $C_k = \sum_{i=0}^{N-k-1} \sigma_i \sigma_{i+k}$.

The main reasons we chose this problem for our experiments was that on one hand, this problem was studied extensively (see [15, 21] and references therein), and on the other hand, it poses a significant challenge for stochastic methods [25]. This is because the topography of the problem consists of many local minima, with global minima extremely scarce and confined to single-bit neighborhoods ("golf-course topography").

To the best of our knowledge, the best results reported to date for the LABS problem were obtained [21] using Constrained Local Search (CLS), which is a hybrid prune-based/stochastic algorithm designed for this problem[5]. Below, we will compare our results with the results of Ref. [21].

[5] Note added: results for N larger than considered here were recently reported in [5].

Model We have modeled the LABS problem as follows. For a given N, a single bit string variable v of length N is defined. In addition, $n-1$ constraints are constructed, each defined by a cost function returning C_k^2, where the values σ_i in the definition of C_k are equal to $2v_i - 1$, v_i being the value of the i^{th} bit in v. v is initialized randomly. We have not implemented any special input functions for this problem, but rather used the same functions as in section 3.2.

LABS is not, strictly speaking, a CSP, but rather an optimization problem. Nevertheless, SVRH treats LABS as a CSP, with the only addition that we monitor the evolving cost of the state, at any time keeping the lowest-cost state found so far. No other modifications are done to SVRH when applying it to optimization problems.

Results We performed experiments on instances of LABS with $45 \leq N \leq 48$ (these are the four hardest instances for which results are reported in Ref. [21]). Results of these experiments are summarized in Table 3: for each N, the table shows the number of restarts, the total number of attempts tried, the number of successful attempts according to the type of the attempt, and the run time[6]. Detailed examination of the learned data shows a large preference to small step sizes, but also distinguished preference to some large step size values (e.g., for $N = 45$, a relatively large success rate for steps of size 35-40, and a very low success rate for steps of size 10-15). In addition, a clear preference is seen to some unique combination of bits, indicating the existence of preferred directions. Similar behavior was noticed also in the floating point verification problem.

The table also shows numbers of backtracks and run times in the CLS experiment [21]. With the exception of $N = 48$, SVRH shows results which are comparable to or better than those obtained by CLS (when comparing SVRH attempts with numbers of CLS backtracks, we should remember that for each backtrack step there is at least one forward step). These results are remarkable for three main reasons: first, SVRH is purely stochastic, while CLS has a strong pruning component. As mentioned earlier, LABS is expected to be a particularly hard problem for stochastic search. Second, SVRH is a highly generic algorithm, while CLS was optimized for LABS, and third, no special domain knowledge was provided to SVRH for this problem.

4 Discussion

We have presented SVRH, a stochastic CSP solver that relies heavily on stochastic moves and on learning of the high-level structure of the topography of the CSP. The solver may also accept user-domain knowledge. The algorithm was outlined and then detailed. This algorithm is based on, and mimics, a zero-temperature physical process governing the dynamics of localized electrons in disordered materials.

[6] Runs were performed on a mid-end IBM PowerPC workstation.

Table 3. SVRH results on the LABS problem. The actual sequences found are 82121121231234321111111 for $N = 45$, 11111111221212132134328 for $N = 46$, and 24212421313131411121114 for $N = 47$. (Numbers in the sequence indicate maximal lengths of contiguous series of +1's and -1's, so $+++-\,-+-+--$ is represented by 421112) For $N = 45$ and $N = 46$ those sequences were already found in Refs. [21] and [15], respectively. However, the sequence for $N = 47$ is new. SVRH failed to find a solution for $N = 48$ within the time limit of 5 hours.

			Attempts	Successful attempts				CLS results [21]	
N	E	restarts	$\times 10^6$	random	user-defined	learned	hours	backtracks$\times 10^6$	hours
45	118	3677	136	4931	37158	23248	1.32	368	14.70
46	131	13991	179	18959	135935	86825	1.74	35	1.44
47	135	18144	242	24757	184964	118036	2.35	165	7.30
48	–	–	–	–	–	–	–	78	3.36

In some sense, the solver may be viewed as a brute-force stochastic solver (the fact that typically only 10^{-5}–10^{-3} of attempts lead to hops supports this view). Still, in all problems we experimented on, the solver sampled only a minute fraction of the state space (a fraction much smaller than the reciprocal of the number of solutions). The only reason the solver could eventually reach a solution is the policy it uses regarding when to hop, and the advantageous learning of the problem (as concluded from the observed repeated choice of previously successful step-sizes and directions).

The SVRH algorithm is of particular value for classes of CSPs that are hard for systematic algorithms. Still, there are such cases that are hard also for SVRH. For example, CSPs with constraints for which any natural choice of a cost function would return a constant (or nearly constant) cost for all states violating the constraint (inequality constraint is one example). In the future we intend to add to the solver the ability to do implications (and thus pruning) after stochastic hops the solver takes. Other future functionality which may enhance the solver is the ability to use static learning databases for classes of similar CSPs, and more sophisticated dynamic learning schemes. Finally, a more convenient modeling scheme may be designed to deal with integer domains.

5 Acknowledgments

I am grateful to Gil Shurek for many valuable discussions and to Raviv Nagel for educating me on the floating-point verification problem, and for providing me with benchmarks for this problem.

References

1. Csplib, www.csplib.org.

2. M. Aharoni, S. Asaf, L. Fournier, A. Koyfman, and R. Nagel. A test generation framework for datapath floating-point verification. In *HLDVT-03*, 2003.
3. E. Bin, R. Emek, G. Shurek, and A. Ziv. Using constraint satisfaction formulations and solution techniques for random test program generation. *IBM Systems Journal*, 41(3):386–402, 2002.
4. K. D. Boese, A. B. Kahng, and S. Muddu. A new adaptive multi-start technique for combinatorial global optimizations. *Operations Res. Lett.*, 16(3):101–113, 1994.
5. F. Brglez, X. Y. Li, M. F. Stallmann, and B. Militzer. Reliable cost predictions for finding optimal solutions to labs problem: Evolutionary and alternative algorithms. In *Workshop on Frontiers in Evolutionary Algorithms (FEA'2003)*, 2003.
6. P. Codognet and D. Diaz. Yet another local search method for constraint solving. In *Proceedings of SAGA'01*, 2001.
7. Jeremy Frank. Learning short-term weights for GSAT. In *IJCAI-97*, pages 384–391, 1997.
8. Jeremy Frank, Peter Cheeseman, and John Stutz. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research*, 7:249–281, 1997.
9. P. Galinier and J.-K. Hao. A general approach for constraint solving by local search. In *CP-AI-OR'00*, 2000.
10. Ian P. Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for SAT. In *AAAI-93*, pages 28–33, 1993.
11. F. Glover and M Laguna. *Tabu Search*. Kluwer, 1997.
12. P. Hansen and N. Mladenovic. introduction to variable neighbourhood search. In *Metaheuristics: Advances and Trends in Local Search Procedures for Optimization*, pages 433–458, 1999.
13. Pascal Van Hentenryck and Laurent Michel. Control abstractions for local search. In *CP 2003*, 2003.
14. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
15. Stephan Mertens. The bernasconi model. *http://odysseus.nat.uni-magdeburg.de/m̃ertens/bernasconi/*.
16. Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. In *OOPSLA 2002*, 2002.
17. S. Minton, M.D. Johnston, A.B. Phillips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *AAAI-90*, pages 17–24, 1990.
18. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *DAC-01: 39th Design Automation Conference*, 2001.
19. N.F. Mott and E.A. Davis. *Electronic Processes in Non-Crystalline Materials*. Oxford: Clarendon, 1979.
20. A. Nareyek. Using global constraints for local search. In *Constraint Programming and Large Scale Discrete Optimization, DIMACS Vol. 57*, pages 9–28, 2001.
21. Steven Prestwich. A hybrid search architecture applied to hard random 3-sat and low-autocorrelation binary sequences. In *CP 2000*, 2000.
22. Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26*, 1996.
23. Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *AAAI-94*, pages 337–343, 1994.
24. Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *AAAI-92*, pages 440–446, 1992.
25. Toby Walsh. low autocorrelation binary sequences. *www.csplib.org prob005/results*.

On the impact of small-world constraint graphs on local search

Andrea Roli

Dipartimento di Scienze
Università degli Studi "G. D'Annunzio"
Pescara – Italia
a.roli@unich.it

Abstract. The impact of problem structure on search is a relevant issue in artificial intelligence and related areas. Among the possible approaches to analyze problem structure, the one referring to constraint graph enables to relate graph parameters and characteristics with search algorithm behavior. In this work, we investigate the behavior of local search applied to SAT instances associated to graphs with small-world topology. Small-world graphs, such as friendship networks, have low characteristic path length and high clustering. In this work, we first present a procedure to generate SAT instances characterized by a constraint graph with a small-world topology. Then we show experimental results concerning the behavior of local search algorithms applied to this benchmark.

1 Introduction

The impact of problem structure on search is a relevant issue in artificial intelligence and related areas. In order to design and tune effective and efficient algorithms for Constraint Satisfaction Problems (CSPs), the relations between structural problem features and algorithm performance have to be investigated. These relations have been studied from different perspectives. Studies on the impact of problem structure on heuristic search can be found, for example, in [2, 31, 28, 15]. Important results and observations on structure and problem hardness are reported in [10, 8, 9]. The effects of problem encoding on instance structure are discussed in [14, 3]. Finally, the search algorithms behavior w.r.t. graph properties of some constraint satisfaction problems and combinatorial optimization problems has been discussed in [27, 26].

The definition of structure emerging from the literature on CSPs is usually based on the informal notion of a property enjoyed by non-random problems. Thus, *structured* is used to indicate that the instance is derived from a real-world problem or it is an instance generated with some similarity with a real-world problem. Commonly, we say structured for a problem that shows, at a given level of abstraction, regularities such as well defined subproblems, patterns or correlations among problem variables. Quantitative measures of structure can also be introduced, such as entropy (see for example [11]), small-world proximity [26] and compression ratio [23].

In this work, we focus on one among the possible ways of characterizing the structure of a problem: We analyze the structure of links among its components, i.e., the

network that connects the components. Some problems suggest a natural structural description, since they have a representation that can be directly used for structure analysis. A classical example are problems defined on graphs, such as the Graph Coloring Problem and the k-Cardinality Tree Problem. In general, for CSPs a constraint graph can be defined, in which nodes correspond to variables and edges connect two variables if there exists a constraint involving them[1]. Hence, the structure of any CSP can be characterized by a graph.

Relevant features of a graph that can affect search behavior are, for example, the *average node degree* and its distribution, the *path length* and the *clustering*. The impact of node degree distribution on search has been studied in [27, 18, 22, 19]. In this work, we investigate the relations between the *small-world* property and search algorithm performance. Small-world graphs [29, 30] are characterized by the simultaneous presence of two properties: the average number of hops connecting any pair of nodes is low and the clustering is high. Social networks defined on the basis of friendship relationships are a typical example of graphs with a small-world topology.

The impact of small-world topology on search problems (e.g., Graph Coloring Problem) has been discussed in [26], where it is shown that many Constraint Satisfaction Problems and Combinatorial Optimization Problems have a small-world topology and the search cost can be characterized by a heavy-tail distribution.

In this work, we report experimental results concerning the behavior of local search applied to instances of the Satisfiability Problem (SAT) with a constraint graph characterized by a small-world topology. The aim of these experiments is to answer the question whether small-world SAT instances are harder to solve than the others and if this behavior is common to different local search algorithms.

The contribution of this work is twofold. First, we define a procedure to construct SAT instances with a lattice structure, along with a method to generate small-world SAT instances. Then, we test three local search algorithms on the generated benchmark. Results show that the behavior strongly differentiates across the algorithms. In some cases, results show that many harder instances are located in the small-world area. This empirical analysis shows that, even if local search can be affected by instance structure, an important role is played by the actual search space exploration strategy.

This paper is structured as follows. Sec.2 introduces the basic concepts of small-world graphs and graphs associated to SAT instances. In Sec.3, we describe the properties of the instances composing the testbed and the procedure used to generate them. Sec.4 presents experimental results obtained by applying three different local search algorithms, namely WalkSAT, GSAT and Iterated local search. We conclude by briefly discussing the results obtained and outlining future work.

2 Preliminaries

In this section, we succinctly introduce small-world graphs and the graph associated to SAT instances.

Given a graph $G = (V, E)$, where V is the set of nodes and E the set of edges, the *characteristic path length* $L(G)$ of G is formally defined as the median of the means

[1] We restrict our considerations to binary constraints.

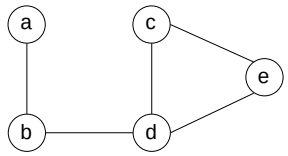


Fig. 1. Constraint graph associated to the SAT instance $(a \vee \neg b) \wedge (b \vee d) \wedge (c \vee \neg d \vee \neg e)$

of the shortest paths connecting each node $v \in V$ to all other nodes. The *clustering* coefficient is defined on the basis of the notion of neighborhood. The neighborhood Γ_v of a node $v \in V$ is the subgraph consisting of the nodes adjacent to v (not including v itself). The clustering of a neighborhood is defined as:

$$\gamma_v = \frac{|E(\Gamma_v)|}{\binom{k_v}{2}},$$

where $|E(\Gamma_v)|$ is the number of edges in Γ_v and k_v is the number of neighbors of v. Therefore, γ_v is the ratio between the number of edges of the neighborhood and the maximum number of edges it can have. The clustering coefficient γ of a graph G is defined as the average of the clustering values γ_v for all $v \in V$. For example, we compute L and γ for the graph depicted in Fig.1. The characteristic path length is the median of the average path lengths of the nodes a, b, c, d and e, i.e., $L = \text{median}\{\frac{9}{4}, \frac{6}{4}, \frac{7}{4}, \frac{5}{4}, \frac{7}{4}\} = \frac{7}{4} = 1.75$. The clustering γ is the average of the neighborhood clustering values, i.e., $\gamma = \frac{1}{5}(0/\binom{1}{2} + 0/\binom{2}{2} + 1/\binom{2}{2} + 1/\binom{3}{2} + 1/\binom{2}{2}) \approx 0.467$.

Typically, random graphs are characterized by low characteristic path length and low clustering, whilst regular graphs (such as lattices) have high values for L and γ. Conversely, small-world graphs are characterized by low L and high γ.

In this work, we apply the notion of small-world to a graph associated to SAT instances. SAT belongs to the class of NP-complete problems [5] and can be stated as follows: given a set of clauses, each of which is the logical disjunction of $k > 2$ *literals* (a literal is a variable negated or not), we ask whether an assignment to the variables exists that satisfies all the clauses.

The graph we associate to a SAT instance is an undirected graph $\mathcal{G} = (\mathcal{V}, A)$, where each node $v_i \in \mathcal{V}$ corresponds to a variable and edge $(v_i, v_j) \in A$ $(i \neq j)$ if and only if variables v_i and v_j appear in a same clause[2]. For instance, in Fig.1 the graph corresponding to the formula $F_1 = (a \vee \neg b) \wedge (b \vee d) \wedge (c \vee \neg d \vee \neg e)$ is depicted.

Observe that the same graph corresponds to more than one formula, since nodes are connected by only one arc even if the corresponding variables belong to more than one clause. For brevity, in the following we will refer to this simple graph we defined to as *SATgraph*. Having a set of clauses associated to the same graph, makes this representation quite rough. Nevertheless, in the following, it will be shown that some properties of the *SATgraph* can strongly affect the behavior of local search applied to SAT instances.

[2] This graph has apparent similarities with the constraint graph defined for constraint satisfaction problems.

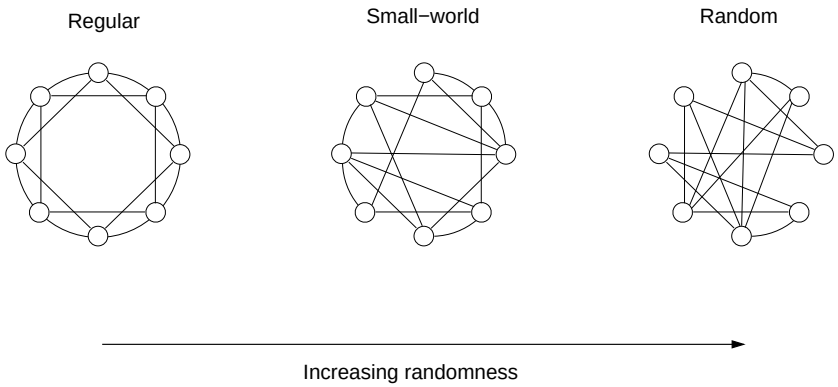


Fig. 2. Morphing between a lattice graph and a random graph. Small-world graphs can be obtained by randomly rewiring just a few links between nodes.

3 Small-world SAT instances

In order to explore the behavior of search algorithms on small-world SAT instances, we generated a benchmark by morphing between instances constructed on lattice graphs and random instances. The core idea of the morphing procedure is derived from [6], wherein a method that enables to generate instances gradually morphing from a source to a destination instance is presented. This procedure is also quite similar to the one used in [30] to generate small-world graphs by interpolating between lattice and random graphs (see Fig.2). Starting from a lattice graph, links are randomly removed and rewired. Small-world graphs can be obtained by randomly rewiring just a few links between nodes.

SAT instances with a small-world graph topology can be obtained by morphing between a SAT instance associated to a lattice graph and a random SAT instance. Therefore, we have first to define a procedure to construct SAT instances associated to a lattice graph. Instead of starting from a SAT formula, expressed as a conjunction of clauses, we start from a graph with the desired topology and we use it as a skeleton for generating a SAT formula.

The starting graph is a *lattice graph*. Lattice graphs have a very regular topology and every node is connected to a fixed (usually quite small) number of neighbors. Examples of lattice graphs are ring lattices (also called cycles) and hypercubes. For instance, a lattice graph in which every node has four neighbors is depicted in Fig.3. Observe that the graph can be seen as a circular structure with adjunctive links connecting neighbors at distance 2.

Once obtained the graph with the given topology, we have to assign variables to nodes and to generate the clauses of the formula. The first step can be completed very easily by assigning variables in order: variable x_i is assigned to node i, for $i = 1, \ldots, n$. The generation of clauses, i.e., of a formula that can be mapped into the given lattice graph, is a bit more complex. First of all, we remind that the graph associated to a

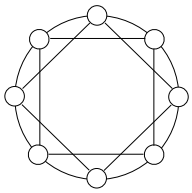


Fig. 3. Example of lattice graph. Each node has 4 neighboring nodes.

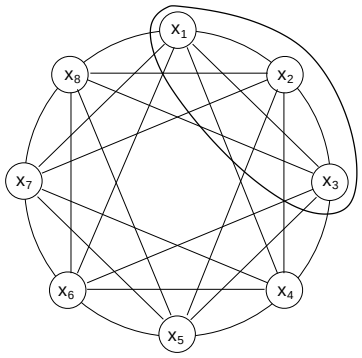


Fig. 4. Construction of the first clause involving variable x_1.

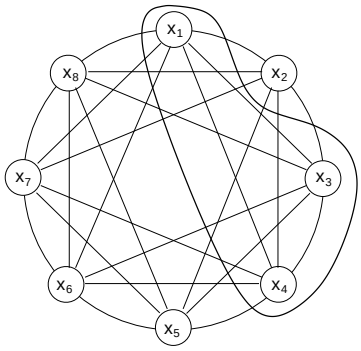


Fig. 5. Construction of the second clause involving variable x_1.

SAT instance, as previously defined, corresponds to a set of SAT instances. Therefore, it is important to define a given structure for the formula. Our choice is to follow the usual experimental settings for random generated SAT instances: 3-SAT formulas with controlled ratio m/n, where n is the number of variables and m the number of clauses.

In the following, we describe the algorithm to generate 3-SAT instances with given ratio m/n on a lattice graph. The generalization of the algorithm to k-SAT instances is straightforward. The high level algorithm is described in Alg.1. The algorithm is structured in two phases. In the first phase, a minimal set of clauses is generated to obtain a formula that can be represented by the given lattice graph. In the second phase, the additional required number of clauses is generated by adding clauses randomly chosen from the first set and by randomly changing the sign of literals.

In the first phase, clauses of three literals are constructed, by taking in turn each variable as a *pivot* and adding two subsequent variables (see Fig.4 and Fig.5). In order to avoid repetitions of clauses, for every variable x_i only subsequent variables $x_j, j > i$ (modulo n) are considered. Indeed, given the symmetry of the graph, the clauses involving the symmetric part of neighbors will be generated by using those neighbors as pivot (see Fig.6 and Fig.7).

A complete example of a lattice-3-SAT instance with $n = 6$, $m = 12$ and nodes with 4 neighbors is the following:

Algorithm 1 Generation of a 3-SAT instance on a lattice graph

INPUT: n, m, γ {γ is the number of neighbors}
OUTPUT: 3-SAT formula $\Phi = \{C_1, \ldots, C_m\}$ with n variables and m clauses associated to a lattice graph with n nodes with γ neighbors each.

Build a lattice graph $G(n,\gamma)$ (on a circle) with n nodes with γ neighbors each;
Assign variables (clockwise) to nodes;
$\Phi \leftarrow \emptyset$
for $i = 1$ to $n-1$ **do**
  The neighbors of x_i are $\mathcal{N}^+ = \{x_{i+1}, \ldots, x_{i+\gamma/2}\}$ (mod n) and $\mathcal{N}^- = \{x_{i-1}, \ldots, x_{i-\gamma/2}\}$ (mod n);
  for each pair x_j, x_{j+1} in $\mathcal{N}^+$ **do**
    Construct the clause $C = x_i \vee x_j \vee x_{j+1}$
    Negate each variable in C with probability 0.5;
    $\Phi \leftarrow \Phi \cup C$
  end for
end for
{Now the number of clauses is $|\Phi| = n(\gamma/2 - 1)$ }
while $|\Phi| < m$ **do**
  repeat
    Pick randomly a clause C' in Φ;
    Negate each variable in C' with probability 0.5;
  until a new clause C' is generated
  $\Phi \leftarrow \Phi \cup C'$
end while

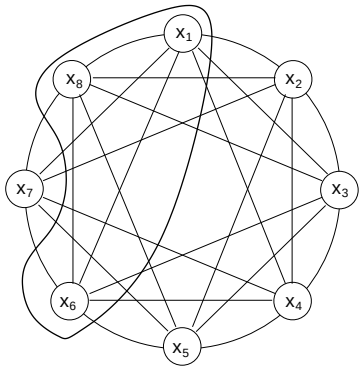


Fig. 6. Construction of the third clause involving variable x_1. The pivot is variable x_6.

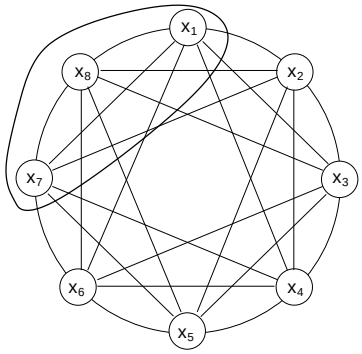


Fig. 7. Construction of the fourth clause involving variable x_1. The pivot is variable x_7.

$\Phi = \{(\neg x_1 \vee x_2 \vee x_3), (x_2 \vee \neg x_3 \vee x_4), (\neg x_3 \vee x_4 \vee x_5), (\neg x_4 \vee \neg x_5 \vee x_6), (x_5 \vee x_6 \vee x_1), (x_6 \vee x_1 \vee x_2), (\neg x_5 \vee x_6 \vee \neg x_1), (x_3 \vee x_4 \vee \neg x_5), (\neg x_5 \vee x_6 \vee x_1), (x_2 \vee x_3 \vee \neg x_4), (x_6 \vee \neg x_1 \vee x_2), (\neg x_2 \vee x_3 \vee x_4)\}$.

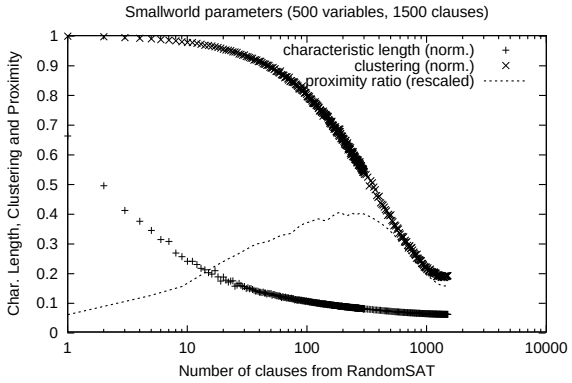


Fig. 8. Characteristic path length L, clustering γ and proximity ratio μ for instances generated by morphing from a lattice SAT instance to a random SAT instance of 500 variables and 1500 clauses.

The instances composing the benchmark are generated by morphing between a lattice SAT instance and a random one. Each instance is obtained by taking from the lattice SAT instance all the clauses except for a prefixed number which are randomly chosen from a random SAT instance with the same number of variables and clauses. This procedure is indeed very similar to the morphing procedure described in [6], but in this case we control the exact number of clauses taken from the destination instance. In this way it is possible to smoothly interpolate from lattice to random and observe the arising of small-world properties in SAT instances.

In order to have a quantitative measure of the small-world characteristic, we introduce the *proximity ratio* μ [26], defined as the ratio between clustering and characteristic path length, normalized with the same ratio corresponding to a random graph, i.e., $\mu = (\gamma/L)/(\gamma_{rand}/L_{rand})$. In Fig.8, the clustering and the characteristic path length of SAT instances gradually interpolating from lattice to random are plotted (in semi-log scale). We observe that L drops very rapidly with the introduction of clauses from the random instance. Conversely, γ maintains a relatively high value for a higher amount of perturbation. The instances with low length and high clustering are characterized by the small-world property. This is also indicated by the proximity ratio curve, which approximately assumes its maximum in correspondence of that region.

We generated four sets of instances (respectively with 100, 200, 500 and 800 variables), each obtained by morphing between a lattice 3-SAT and a random 3-SAT with same number of variables and clauses. All the generated instances are satisfiable (unsatisfiable instances have been filtered by means of a complete solver). The ratio between the number of clauses and the number of variables is 3, lower than the so-called critical ratio (which is close to 4.3 for 3-SAT instances [1, 17, 12]). This is due to the structure of lattice SAT instances which turned out to be almost all unsatisfiable at the critical ratio.

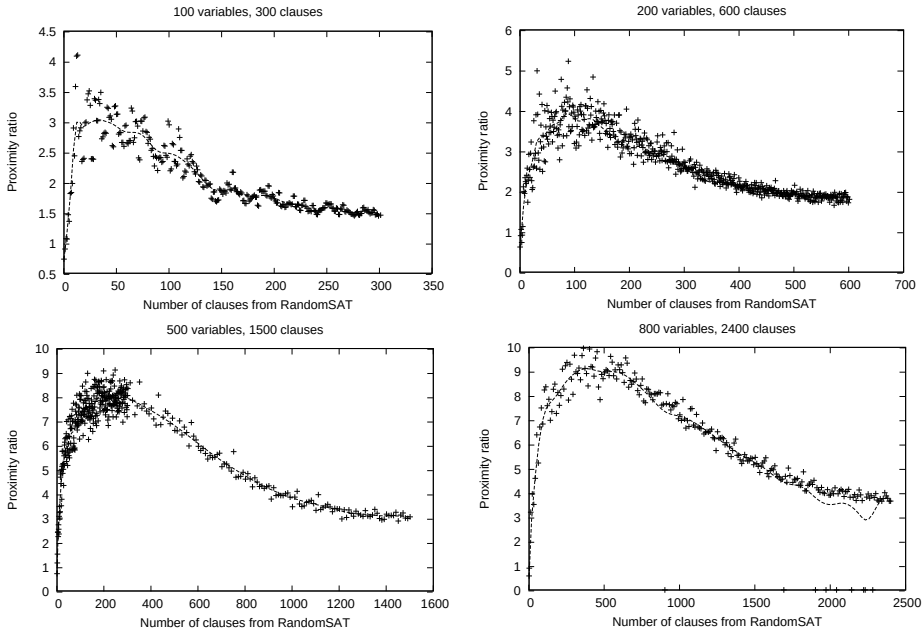


Fig. 9. Proximity ratio of the instances composing the benchmark.

In Fig.9, the proximity ratio of the instances composing the benchmark is plotted. The value μ ranges approximately from 0.5 to 10. We can observe the typical behavior of instances interpolating between regular and random instances.

In the next section we present experimental results on the behavior of local search algorithms on the benchmark defi ned.

4 Experimental Results

Some constraint satisfaction problems and combinatorial optimization problems with small-world topology have been found to require a higher computational search cost with respect to "non small-world" ones [26]. Since those results only concerned complete algorithms, we ask whether this behavior can also be observed in the case of approximate algorithms, namely local search.

We performed a series of experiments aimed at checking whether small-world SAT instances are harder to solve than both regular (lattice) and random ones. In the following, we will use the notion of *hardness* referred to the pair $\langle$*instance,algorithm*$\rangle$. We estimate the hardness by means of the search cost, namely the number of iterations required for the algorithm to fi nd a satisfying assignment. Since the algorithms we deal with are stochastic, we run each of them 1000 times on the same instance and we took the median value. We emphasize that we use the concept of hardness referring to a given algorithm $\mathcal{A}$ and we say that an instance I_1 is harder than I_2 if the search cost (as

defined above) for solving I_1 via $\mathcal{A}$ is higher than that of I_2. Even if this definition of hardness is grounded to the algorithm used, in general it is possible to observe that a class of instances is harder than another class for a set of algorithms. This case reveals that there is a characteristic of the class that makes the instances difficult for all the considered algorithms.

We applied three different local search procedures, that are based on different heuristic strategies. The algorithms we considered are WalkSAT [24], GSAT [25] and Iterated local search (ILS, [16, 20]). GSAT was the first effective local search algorithm proposed for SAT. It applies a greedy strategy, by flipping the variable that increases the number of satisfied clauses the most. GSAT suffers from frequent stagnation, therefore its performance is often not satisfactory for large instances. WalkSAT is based on the principle of repair: It randomly chooses one unsatisfied clause and flips one variable within it. There are some different heuristics for the choice of the variable to flip [13]. In our implementation, we applied a GSAT-like heuristic, i.e., the variable that produces the largest increment in the number of satisfied clauses is flipped (no random walk is performed). WalkSAT has usually a far better performance than GSAT. Nevertheless, both algorithms lack a global strategy that could guide them during the exploration of the search space. Algorithms equipped with such a strategy are commonly called metaheuristics [4]. In order to extend the diversity of the techniques compared, we applied also an ILS designed to attack SAT and MAXSAT problems [20, 21]. In essence, this metaheuristic is a tabu search-based WalkSAT guided by a strategy that tunes both the tabu tenure and the intensification/diversification balance by using the search history.

Results are shown in Figs. 10, 11 and 12. In the plots, we reported for each algorithm the median iterations over 1000 runs on every instance. The algorithms run until a feasible solution was found[3].

Results are very interesting and show the complexity of empirical analysis of local search behavior. First of all, we note that the behavior across the three algorithms is very different. In the first two plots of Fig.10, we observe that some of the hardest instances for WalkSAT are located in the small-world area. Nevertheless, for the instances of size 500 and 800, the search cost regularly increases –linearly in log scale– while morphing from lattice to random. This peculiar behavior requires a deeper investigation and from these preliminary results we can only conjecture that size scaling amplifies a characteristic of the instances such that the more random the instance structure, the harder the instance for WalkSAT. GSAT and ILS show a mild tendency of requiring higher search cost in the vicinity of the small-world area, as shown in Fig.11 and Fig.12, respectively. The hardest instances for GSAT and ILS are the ones located in the first part of the plots, i.e., the instances with strong lattice/small-world topologies[4]. In some plots, we also observe that the instances corresponding to the maximal proximity ratio

[3] In the case of GSAT, due to the extremely high execution time, we stopped the algorithm at a maximum number of non-improving moves and we reported the success ratio, i.e., the number of successful runs out of 1000.

[4] The 800-2400 instances are indeed not solved by GSAT in the range corresponding to small-world.

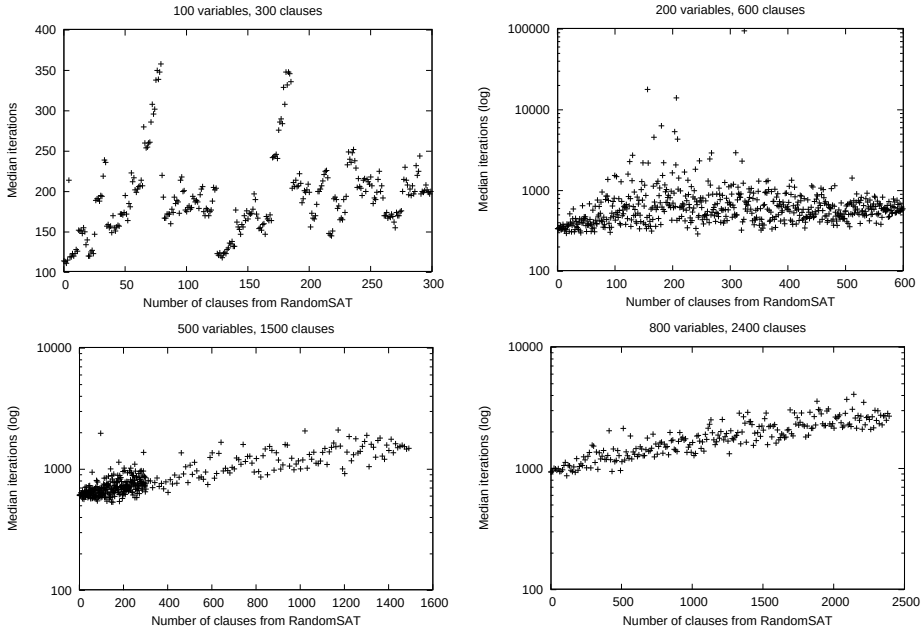


Fig. 10. Search cost of WalkSAT across the instances, from lattice to random structure. Points represent median iterations over 1000 runs. Log-scale on the y-axis has been used when necessary.

are the hardest on average[5]. Nevertheless, this behavior is not regular nor clear and the statistical correlation between search cost and proximity ratio is quite low.

The peculiar behavior observed is a clear signal that different factors other than small-world topology affect algorithm behavior. Among the main factors, we consider the search landscape characteristics induced by the SAT instance and the actual search process performed by the algorithm on the landscape. In fact, the landscape characteristics and the strategy used to explore it are the main elements that affect local search behavior. The relations between instance structure and search landscape are an extremely important research issue, that is subject of ongoing work.

We conclude this section by showing results concerning the application of a complete algorithm. The results clearly show that, in the case of tackling the benchmark with a systematic technique, small-world SAT instances are the hardest. As a complete solving procedure we chose BerkMin [7], one of the most effi cient complete SAT solvers available nowadays. The search cost has been evaluated as the number of variable assignments performed by the algorithm before solving the instance. In Fig.13, the search cost is plotted against the proximity ratio. We clearly observe that the higher the proximity ratio, the higher the search cost.

[5] This observation is also confirmed by evaluating a moving window average.

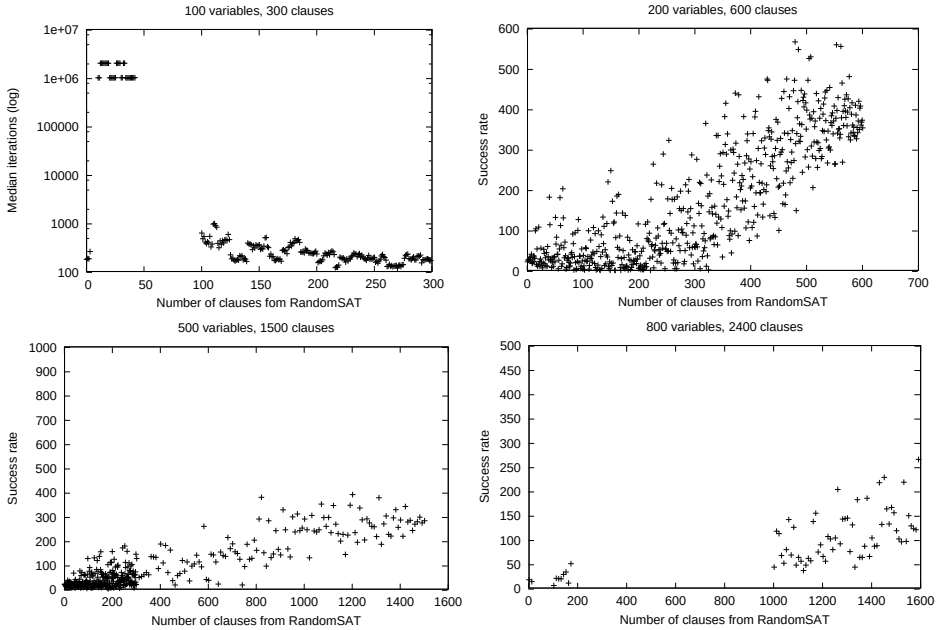


Fig. 11. Search cost of GSAT across the instances, from lattice to random structure. In the uppermost left plot, points represent median iterations over 1000 runs. The remaining plots report an estimation of the search cost in terms of success ratio, i.e., the number of instances solved – given a termination condition defined as the maximum number of non-improving moves.

5 Conclusion and Future work

In this work we have presented a procedure to generate SAT instances associated to constraint graph with small-world topology. Small-world SAT instances are constructed by introducing clauses from random instances into lattice graph based SAT ones.

We tackled the benchmark instances with three different local search-based algorithms and observed their behavior across the whole spectrum, from regular lattice to random topologies. Our aim was to check whether there is a positive correlation between search cost and small-world topology of SAT instances, as observed when small-world constraint satisfaction problems are tackled with systematic solvers. Results showed primarily that the behavior of local search algorithms is fairly different. In some cases, we also observed that most of the hardest instances are concentrated in the lattice/small-world area. Nevertheless, this result is not as clear as in the case of complete solvers and further investigations are required. If the conjecture on the positive correlation between instance hardness and proximity ratio is true, then the phenomenon may be explained considering the locality of decisions taken by the heuristics, as supposed in [26]: a locally good decision taken w.r.t. the clustering properties might be wrong with respect to the whole graph.

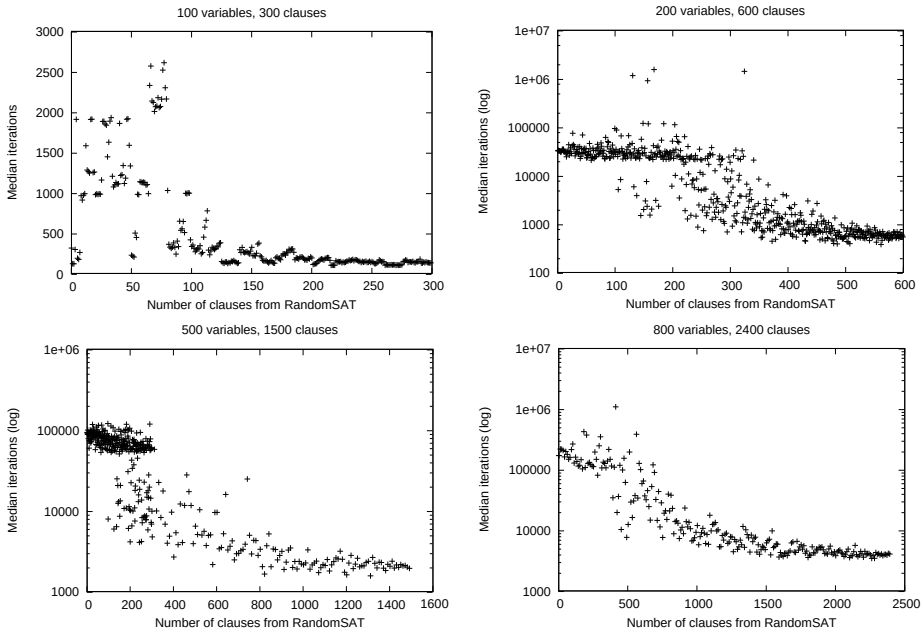


Fig. 12. Search cost of ILS across the instances, from lattice to random structure. Points represent median iterations over 1000 runs. Log-scale on the y-axis has been used when necessary.

The study of relations between structure and local search behavior is still a partially unexplored area. First of all, in this work we have just considered one of the possible ways of characterizing structure. Other definitions for graphs are possible (such as weighted graphs), to capture different problem features and to extend our results to problems other than SAT. Moreover, concerning local search algorithms, we believe that the core issue to explain the algorithm behavior is the investigation of the relations between problem structure and search landscape, and, in turn, search landscape and the actual strategy used to explore it.

Acknowledgments

I would like to thank Michela Milano for discussions and very interesting suggestions on this research.

References

1. D. Achlioptas and C. Moore. The asymptotic order of the random k -SAT threshold. In *Proceedings of FOCS 2002*, pages 779–788, 2002.
2. J.C. Beck and M.S. Fox. Dynamic problem structure analysis as a basis for constrained -directed scheduling heuristics. *Artificial Intelligence*, 117:31–81, 2000.

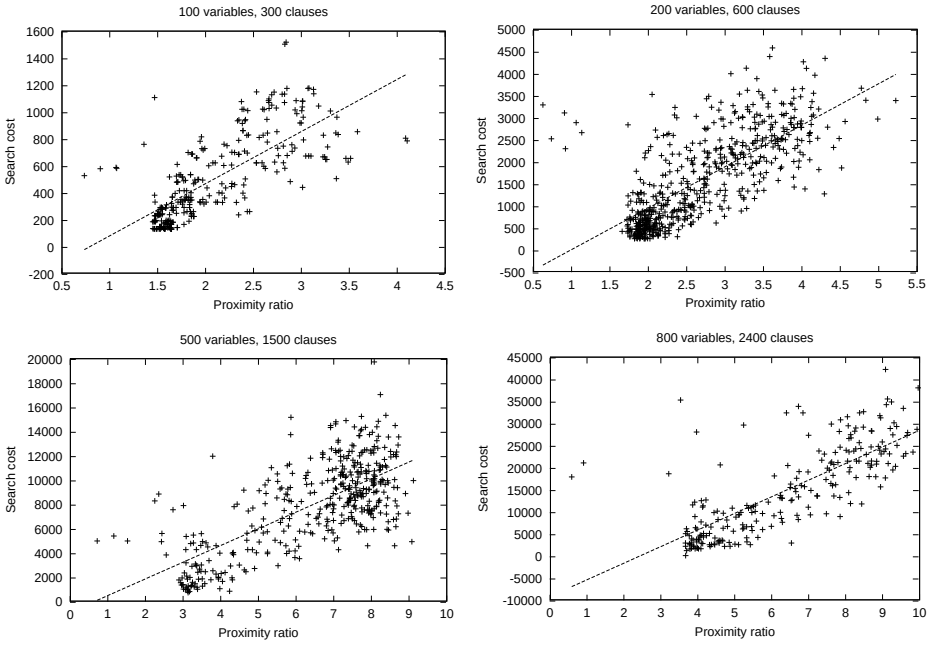


Fig. 13. Correlation between search cost and proximity ratio.

3. R. Béjar, A. Cabiscol, and C.P. Gomes C. Fernàndez, F. Manyà. Capturing structure with satisfiability. In T. Walsh, editor, *Principles and Practice of Constraint Programming – CP 2001, 7th International Conference*, Lecture Notes in Computer Science, pages 137–152. Springer, 2001.
4. C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, September 2003.
5. M. R. Garey and D. S. Johnson. *Computers and intractability; a guide to the theory of NP-completeness*. W.H. Freeman, 1979.
6. I. P. Gent, H. H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proceedings of AAAI99*, pages 654–660, 1999.
7. E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT Solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.
8. C.P. Gomes. Structure, duality, and randomization – common themes in AI and OR. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, 2000.
9. C.P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.
10. C.P. Gomes and D. Shmoys. Completing quasigroups or latin squares: A structured graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and Extensions*, 2002.
11. T. Hogg. Which search problems are random? In *Proc. of AAAI98*, pages 438–443, 1998.
12. T. Hogg, B. A. Huberman, and C. P. Williams. Phase transitions and the search problems. *Artificial Intelligence*, 81(1–2), 1996. Special issue on Phase Transitions and Search Problems.

13. H. H. Hoos and T. Stützle. Towards a characterisation of the behaviour of stochastic local search algorithms for sat. *Artificial Intelligence*, 112:213–232, 1999.
14. H.H. Hoos. SAT-encodings, search space structure, and local search performance. In *Proceedings of IJCAI-99*, pages 988–993, 1999.
15. K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems. In P. Van Henteryck, editor, *Proceedings of CP02 - Eighth International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*. Springer, 2002.
16. H. R. Lourenço, O. Martin, and T. Stützle. Iterated local search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 321–353. Kluwer Academic Publishers, Norwell, MA, 2002.
17. D. G. Mitchell, B. Selman, and H. J. Levesque. Hard and easy distributions of sat problems. In *Proceedings, Tenth National Conference on Artificial Intelligence*, pages 459–465. AAAI Press/MIT Press, July 1992.
18. A. Roli. Criticality and parallelism in GSAT. *Electronic Notes in Discrete Mathematics*, 9, 2001.
19. A. Roli. Criticality and parallelism in structured SAT instances. In P. Van Henteryck, editor, *Proceedings of CP02 - Eighth International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 714–719. Springer, 2002.
20. A. Roli. Design of a new metaheuristic for MAXSAT problems (extended abstract). In P. Van Henteryck, editor, *Proceedings of CP02 - Eighth International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, page 767. Springer, 2002.
21. A. Roli. Metaheuristics and structure in satisfiability problems. Technical Report DEIS-LIA-03-005, University of Bologna (Italy), May 2003. PhD Thesis - LIA Series no. 66.
22. A. Roli and C. Blum. Critical Parallelization of Local Search for MAX–SAT. In F. Esposito, editor, *AI*IA2001: Advances in Artificial Intelligence*, volume 2175 of *Lecture Notes in Artificial Intelligence*, pages 147–158. Springer, 2001.
23. A. Roli and F. Zambonelli. Emergence of macro spatial structures in dissipative cellular automata. In *Proc. of ACRI2002: Fifth International Conference on Cellular Automata for Research and Industry*, volume 2493 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2002.
24. B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for local search. In *Proc. 12th National Conference on Artificial Intelligence, AAAI'94, Seattle/WA, USA*, pages 337–343, 1994.
25. B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. American Association for Artificial Intelligence, AAAI Press.
26. T. Walsh. Search in a small world. In *proceedings of IJCAI99*, pages 1172–1177, 1999.
27. T. Walsh. Search on high degree graphs. In *Proceedings of IJCAI-2001*, 2001.
28. J.P. Watson, L. Barbulescu, A.E. Howe, and L.D. Whitley. Algorithm Performance and Problem Structure for Flow-Shop Scheduling. In *Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, 1999.
29. D.J. Watts. *Small Worlds: The Dynamics of Networks between Order and Randomness*. Princeton University Press, 1999.
30. D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

31. C. Williams and T. Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70:72–117, 1994.

Exploiting Relaxation in Local Search

Steven Prestwich

Cork Constraint Computation Centre
Department of Computer Science
University College, Cork, Ireland
email: s.prestwich@cs.ucc.ie

Abstract. Branch-and-bound uses relaxation to prune search trees but sometimes scales poorly to large problems. Conversely, local search often scales well but may be unable to find optimal solutions, perhaps because it does not exploit relaxation. Both phenomena occur in the construction of low-autocorrelation binary sequences, a problem arising in communication engineering. This paper proposes a hybrid approach to optimization: using relaxation to prune local search spaces. An implementation gives competitive results, showing the feasibility of the approach.

1 Introduction

Branch-and-bound is a well-established approach to solving combinatorial optimization problems (see [2] for example). It uses problem relaxations to compute a cost bound in order to prune a search tree, which greatly speeds up backtrack search. It is guaranteed to find optimal solutions and to prove them optimal. However, it sometimes scales poorly to large problems. In such cases local search is a useful alternative. Though it is not guaranteed to find an optimal solution, its superior scalability often makes it indispensable.

A natural way to apply local search to optimization problems is to explore the space of solutions while attempting to minimize the given cost function. For example a well-known local search algorithm for the Traveling Salesman Problem explores the space of tours (Hamiltonian cycles) by exchanging pairs of variable assignments [13]. Exchanges that improve the cost function are always accepted while other exchanges are only accepted under certain conditions. Most local search algorithms for optimization problems similarly try to transform solutions into better solutions, though they vary considerably in detail. Unfortunately this natural approach is not always successful, for example the optimization problem considered in this paper is considered to be very hard for local search. Previous researchers have applied to this problem either randomized search to find large sequences of high quality, or branch-and-bound search to find smaller optimal sequences.

Thus finding optimal solutions to large problems is sometimes difficult. One way forward is to find an improved local search algorithm but this may not be possible. This paper describes an alternative approach: changing the local

search space to allow pruning by relaxation, thus combining the scalability of local search with the cost reasoning of branch-and-bound. The paper covers some of the same ground as [21] but contains new results (in particular for skew-symmetric sequences), modified heuristics and updated references. The remainder of this section introduces an optimization problem and summarizes previous approaches to solving it, while subsequent sections describe the new approach, evaluate it and draw conclusions.

2 The LABS problem

Consider a binary sequence $S = (s_1, \ldots, s_N)$ where each $s_i \in \{1, -1\}$. The *off-peak autocorrelations* of S are defined as

$$C_k(S) = \sum_{i=1}^{N-k} s_i s_{i+k} \quad (k = 1 \ldots N-1)$$

and the *energy* of S as

$$E(S) = \sum_{k=1}^{N-1} C_k^2(S)$$

The *low-autocorrelation binary sequence* (LABS) problem is to assign values to the s_i such that $E(S)$ is minimized. A common measure of sequence quality is the *merit factor* $F(S) = N^2/2E(S)$. Theoretical considerations [8] give an upper bound on $F(S)$ of approximately 12.32 as $N \longrightarrow \infty$, and empirical curve fitting on known optimal sequences [14] yields an estimate of $F \approx 9.3$ for large N. This problem has many practical applications in communications engineering, and is of theoretical interest to physicists because it models 1-dimensional systems of Ising-spins. For our purposes it is mainly of interest as a difficult optimization problem with a cheaply-computed relaxation, and several published papers containing computational results.

2.1 Previous approaches

Much computer time has been devoted to finding sequences with high merit factor. Since the 1970s researchers, many from the Physics community, have applied various methods to the problem. Analytical methods have been used to construct optimal sequences for certain values of N (see [15] for example), but for the general case search is necessary. Two possibilities are *systematic* and *randomized* search.

Systematic search usually involves the enumeration of possibilities by backtracking. Golay [8] used exhaustive enumeration to find optimal sequences for $N \leq 32$. Mertens [14] enumerated optimal sequences for $N \leq 48$ using systematic search augmented with two techniques to reduce the size of the search tree: *branch-and-bound* and *symmetry breaking* (at the time of writing this algorithm has enumerated optimal sequences up to $N = 58$[1]). Branch-and-bound

[1] http://itp.nat.uni-magdeburg.de/~mertens/bernasconi/open.dat

improves performance significantly: a systematic search for sequences of length 44 took approximately 2 days, as opposed to an extrapolated 68 days for exhaustive enumeration (on a Sun UltraSparc I 170 workstation). Symmetry breaking, sometimes called *symmetry exclusion*, exploits the fact that sequences occur in equivalence classes of size 8. However, even with these enhancements, systematic search is unlikely to scale up to large sequences, and it is conjectured that for $N > 100$ progress will be made through mathematical insight rather than computer power [14].

When systematic search becomes impractical, it is common to resort to randomized methods such as simulated annealing, evolutionary algorithms, neural networks, ant colonies or hill climbing, which are often able to solve much larger instances. Unfortunately they perform quite poorly on some problems and finding optimal LABS solutions seems to be an example, despite decades of research effort on simulated annealing [8, 4], evolutionary search [6, 16, 19, 31] and other heuristic algorithms [3]. The cause is considered to be the search space, whose cost function E has a very irregular structure with isolated minima [4]. Bernasconi [4] predicted that local search will be unable to find sequences with merit factor greater than 5, and Mertens [14] suggested that local search algorithms should be evaluated by the percentage of known optimal sequences they find. Until recently [5] no reported randomized algorithm were able to find optimal sequences (we discuss recent results below).

Thus neither systematic nor randomized search seems ideal for finding large optimal sequences, a situation mirrored in some other combinatorial problems. Though both systematic and randomized search are highly successful, neither is seen as adequate for all problems [7] so new hybrid algorithms are of interest.

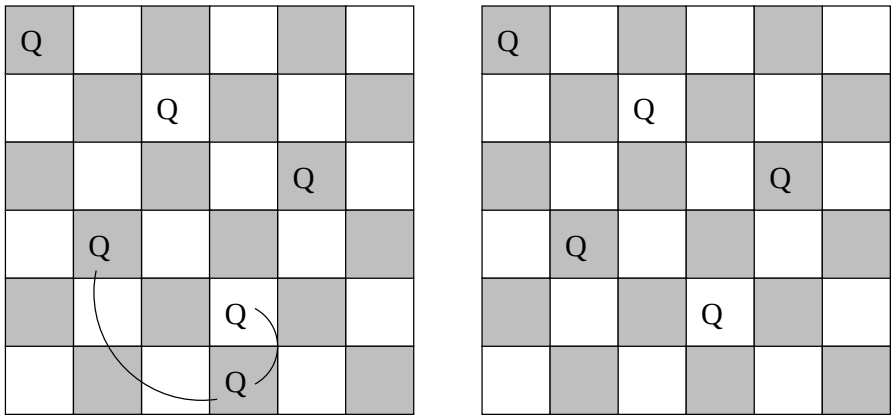
3 A hybrid approach

Before describing the new approach we digress to consider the well-known N-queens problem. The aim is to place N queens on an $N \times N$ chess board in such a way that no queen attacks another (a queen attacks another if they lie on the same row, column or diagonal). N-queens has no cost function to optimize: it is a constraint satisfaction problem (CSP). A popular model for this problem uses N variables each taking a value from the integer domain $\{1, \ldots, N\}$, each variable corresponding to a row and each value to a column. Systematic backtrack search can solve instances only up to approximately 100 queens in a reasonable time, but local search is much faster.

3.1 Local search for constraint satisfaction

The usual local search approach for a non-optimization CSP such as N-queens is to transform it into an optimization problem: the search space is the set of total assignments (all N queens are placed somewhere on the board) and the cost function is the number of constraint violations (attacks). Figure 1(i) shows a total assignment containing two constraint violations: the last queen can attack

two others and vice-versa (attack is symmetrical). We may try to remove these violations, at the risk of introducing new ones, by repositioning a queen involved in a conflict, that is by reassigning a variable to another value. The Min-Conflicts Hill-Climbing algorithm works in this way and solves large instances in a constant number of local moves and linear time [17]. A similar approach is taken for many other problems, for example the GSAT algorithm for Boolean satisfiability [30]. (For optimization problems in which legal solutions are hard to find, the two cost functions may be combined as a weighted sum. Local search then tries to reduce solution cost and infeasibility at the same time.)



(i) inconsistent total assignment (ii) consistent partial assignment

Fig. 1. Two search spaces for N-queens

However, a non-computer scientist might intuitively design a different form of local search for N-queens: starting from an empty board, place queens randomly in non-attacked positions; when a queen cannot be placed, randomly remove one or more placed queens; repeat until all queens are placed. The states of this algorithm correspond to consistent partial assignments in our model, as shown in Figure 1(ii). We may add and remove queens randomly, or bias the choice using heuristics. This algorithm explores a different space but is still local search. The number of unassigned variables (unplaced queens) is the cost function to be minimized. No queens can be added to the state in Figure 1(ii), which is therefore a local minimum under this function.

This approach to N-queens was found to be inferior to Min-Conflicts, but when further techniques were added (including constraint propagation) it was able to solve large instances in even fewer search steps [24]. A few other local search algorithms explore partial assignments: the open-shop scheduling algorithm of [11]; the timetabling algorithm of [29], which allows some constraint violations; the IMPASSE class of graph colouring algorithms [12, 18], in which a

consistent partial assignment is called a *coloration neighbourhood*; the colouring algorithm of [22] which prunes an IMPASSE-style search by constraint propagation; and the SAT algorithms of [10, 21, 27].

3.2 An alternative local search space

The above ideas suggest treating LABS (and other optimization problems) as a series of CSPs with decreasing energy. Each CSP requires the construction of a sequence under a constraint $E(A) \leq U$, where A is a partial sequence, $E(A)$ is the energy of A, and U is an upper bound. As above, each CSP may be solved by applying local search to minimize the number of unassigned variables. If this can be reduced to zero then A is a total assignment (a complete sequence) whose energy is no greater than U. By iteratively decreasing U we may eventually find an optimal sequence.

This may appear to be a convoluted way of applying local search to an optimization problem, but it is worth exploring for several reasons. Firstly, there is the simple hope that changing the search space and cost function will make some problems easier to solve. Secondly, the computation of a lower bound on $E(A)$ provides an opportunity for exploiting relaxation, yielding an interesting hybrid of local search and branch-and-bound. Thirdly, a similar approach has yielded good results on graph colouring [22], Boolean satisfiability [21, 23, 27], maximum cliques [24], Golomb rulers [24], block design [26] and a sports scheduling problem [26]. For these problems we combined local search with constraint propagation via the same form of non-systematic backtracking. The aim of the method described in this paper is to extend the approach from Constraint Programming to Operations Research, by combining local search with relaxation instead of propagation.

3.3 The algorithm

The algorithm will be referred to as LSR (Local Search with Relaxation) and is shown in Figure 2. The *optimize* function finds a solution under a cost constraint given by an upper bound U (initially set to ∞), decrements the upper cost bound by Δ to ensure that future solutions have lower cost, and continues. For general optimization problems we would use $\Delta = 1$, but for LABS it is known [16] that sequences have energy differing by multiples of 4 so we can set $\Delta = 4$ (when searching among the *skew-symmetric* sequences below we can set $\Delta = 8$).

The *solve* function finds a solution A: a set of N variable assignments, one per binary variable, defining a sequence whose energy is no greater than U. Not shown is a simple heuristic: the search for each solution starts by trying to reuse the previous solution. Termination occurs when the lower cost bound L is reached; this may never occur, so a time limit must be imposed.

The *solve* function finds a solution as follows. The set of assignments A is initially empty and the set of unassigned variables V initially contains all N variables. At each iteration an unassigned variable v is selected from V and the set $D \subseteq \{-1, 1\}$ of consistent assignments is (lazily) computed. A *consistent*

```
function optimize({s_1, ..., s_N}, L)
    U = ∞
    while U ≥ L
        A = solve({s_1, ..., s_N}, U)
        U = E(A) − Δ
    return A

function solve({s_1, ..., s_N}, U)
    A = {},   V = {s_1, ..., s_N}
    while V ≠ {}
        s = select-variable(V)
        D = {d ∈ {−1, 1} | E(A ∪ {(s=d)}) ≤ U}
        if D ≠ {}
            d = select-value(D)
            A = A ∪ {(s=d)},   V = V \ {s}
        else
            (s′=d) = select-assignment(A)
            A = A \ {(s′=d)},   V = V ∪ {s′}
    return A
```

Fig. 2. The LSR algorithm for LABS

assignment is one that does not violate the cost constraint, that is the energy of the partial sequence must be no greater than U.

If D is not empty then one of its values is selected, the new assignment added to A, and the newly assigned variable removed from V. If D is empty then A cannot be extended to a solution, so at least one assignment must be undone. For LABS one unassignment seems to be sufficient (though in principle more than one may be necessary) and it is selected heuristically (*select-assignment*) from A, deleted from A, and the newly unassigned variable added to V. The search proceeds until V is empty in which case A is a solution; the search is non-systematic so this may never occur.

Variable selection is performed by a heuristic (*select-variable*). With probability p it selects a random member of V, otherwise it selects the most recently unassigned variable. This heuristic produces a slowly-changing set of assigned variables. The value of p is set to $1/(10N \times 1.06^N)$; this function was chosen empirically and we do not claim that it is optimal, nor even that it should be of this form. The assignment selection heuristic (*select-assignment*) is random. The value selection heuristic (*select-value*) depends upon the selected variable: if this was the most recently unassigned variable then the heuristic prefers the unused value, otherwise it prefers the value last assigned to that variable. If the preferred value cannot be used without violating the cost constraint then the other value is used. The aim is to produce a minimal change in the search state, either in the values of the assigned variables or in the set of assigned variables.

In order to minimize the minimum energy

$$E_{min}(A) = \min\left(\sum_{k=1}^{N-1} C_k(A)^2\right)$$

of a partial sequence A, the relaxation

$$E^*_{min}(A) = \sum_{k=1}^{N-1} \min(C_k(A)^2)$$

can be used as a lower bound $E^*_{min}(A) \leq E_{min}(A)$. Because $E^*_{min}(A)$ is still expensive to calculate, Mertens' branch-and-bound algorithm uses a cheap lower bound $E_b(A) \leq E^*_{min}(A)$ based on an arbitrary completion of the current partial sequence. We take a similar definition for $E_b(A)$ that is not based on an arbitrary sequence but still provides a lower bound for $E^*_{min}(A)$. Define a product $s_i s_j$ to be *computable* if s_i and s_j are both assigned in the current partial sequence A. Then let

$$E_b(A) = \sum_{k=1}^{N-1} L_k(A)$$

where $L_k(A) = \max(b_k, |T_k(A)| - F_k(A))^2$, $b_k = (N-k) \bmod 2$, $T_k(A)$ is the sum of its computable products, and $F_k(A)$ is the number of its uncomputable products. The reasoning is that the sum $T_k(A)$ of the computable products may be offset by the sum of the uncomputable products, which is no greater than $F_k(A)$. If $|T_k(A)| - F_k(A) > 0$ then $(|T_k(A)| - F_k(A))^2$ is a lower bound on $C_k(A)$. If $|T_k(A)| - F_k(A) \leq 0$ then we cannot use the computable products to compute a lower bound. But we also know that $C_k(A) \geq b_k$ because if $N-k$ is odd then $C_k(A) \neq 0$, so the lower bound can be refined slightly by using b_k.

The value of E_b can be computed incrementally. On [un]assigning a variable s_i the algorithm does the following for each occurrence $s_i s_w$ in each C_k where s_w is assigned: subtract $L_k(A)$ from E_b, decrement [increment] $F_k(A)$, add [subtract] $s_i s_w$ to $T_k(A)$, recalculate $L_k(A)$, and add it to $E_b(A)$. The occurrences of each variable s_i are stored in a table, together with the numbers w and k.

3.4 A note on symmetry breaking

Mertens' algorithm benefits from symmetry breaking: a sequence S can be reversed, have all their values inverted, or have alternate values inverted, without changing $E(S)$. Thus sequences occur in equivalence classes of size 8 (apart from a few sequences such as palindromes) and this fact can be used to reduce the search space significantly. It seems natural to exploit this technique in our algorithm also, but we believe that it will be counter-productive. In experiments on several combinatorial problems, symmetry breaking was found to have a harmful effect on local search performance [25]. Symmetry breaking can cause the search to lured into regions of the search space containing only excluded solutions. Dynamic symmetry breaking techniques can counteract this effect [1] and

have been applied to LABS with backtrack search, but with randomized search the size of the search space is less important than the density and distribution of solutions. We therefore do not exclude symmetries.

4 Experimental results

In this section the LSR algorithm is evaluated on LABS instances. The algorithm is implemented in C and all experiments are performed on a 733 MHz Pentium III.

4.1 Optimal sequences

First we evaluate its ability to find known optimal sequences. We calculated the median CPU time over 100 runs taken by the algorithm to find optimal sequences for $N = 20, 21, \ldots, 40$. By linear regression on the logarithm of the CPU time we find that the time taken to find an optimal solution is $O(1.51^N)$, shown in Figure 3 as a straight line — an improvement over our previous result [21]. Exhaustive enumeration takes $O(2^N)$, while branch-and-bound takes $O(1.85^N)$ to find optimal solutions and to prove them optimal [14]. A recent evolutionary algorithm [5] takes $O(1.40^N)$. All these results were calculated using different methods and cannot be directly compared, but if proving optimality is unnecessary then LSR is clearly an attractive option.

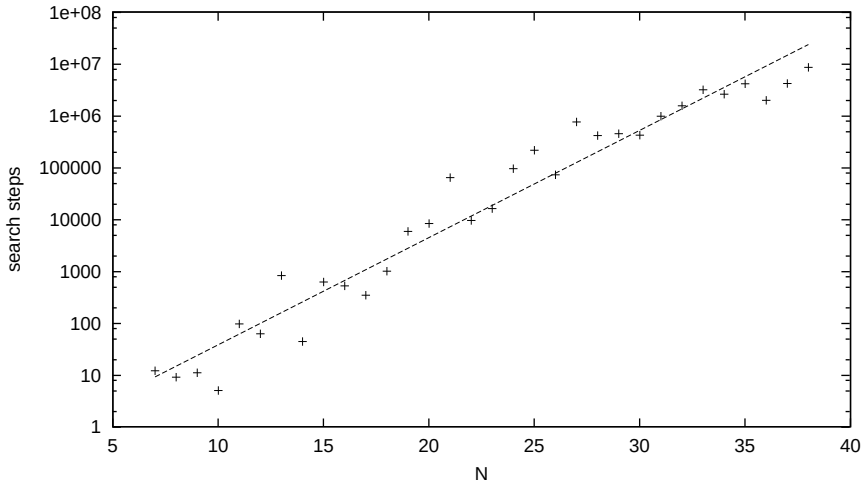


Fig. 3. Performance on optimal sequences

4.2 A long sequence

Next we use LSR to find sequences of length $N = 61$, which is currently just beyond the range of results found by branch-and-bound. de Groot et al. [6] found sequences up to $F = 6.5$ using evolutionary strategies. Bernasconi [4] found $F \approx 6.8$ using simulated annealing. A previous LSR implementation [21] found a sequence with $F = 7.56$. Brglez et al. [5] recently found sequences with $F = 8.23$ after an average of 27.7 hours (on a 266 MHz workstation) and argue that this is likely to be optimal. The best sequence found by LSR after a few hours of computation is

$$33211112111235183121221111311311$$

also with $F = 8.23$ ($E = 226$). Following convention, the sequence is shown in *run-length notation*, each number indicating the number of consecutive elements with the same value. For example a sequence

$$(1, 1, -1, 1, -1, -1, -1, -1, 1)$$

would be written 21141. (For runs of length greater than 9 upper-case letters are used with A=10, B=11 etc.)

4.3 Skew-symmetric sequences

Finally we apply LSR to larger sequences. Because most randomized search algorithms fail to find optimal sequences they are usually applied to LABS via a *sieve* to restrict the search to a promising subspace. The most common sieve is *skew-symmetry* [8]: only sequences of odd length, with $N = 2n - 1$ for some n, satisfying

$$s_{n+i} = (-1)^i s_{n-i} \quad (i = 1 \ldots n - 1)$$

are considered, roughly halving the number of independent variables in the problem and thus greatly reducing the search space. Such sequences often have good merit factors because $C_k = 0$ for all odd k. Note that the restriction to skew-symmetric sequences should not be confused with symmetry breaking. We do not exclude symmetrically equivalent sequences, but sequences that do not possess a symmetry property. Whereas symmetry breaking allows to recovery of all solutions to a problem, there is no guarantee that the best sequence of given length N is skew-symmetric (a counter-example is $N = 61$).

Optimal skew-symmetric sequences have been enumerated by branch-and-bound for $N \leq 71$ [6]. Skew-symmetry is added to LSR by searching on variables $\{s_1, \ldots, s_n\}$ and adjusting the energy computations to add or subtract increments from the corresponding variables in $\{s_{n+1}, \ldots, s_N\}$. The probability p used for variable selection is set as above, but with n in place of N because we now have n independent variables instead of N: $p = 1/(10n \times 1.06^n)$. LSR is able to find optimal skew-symmetric sequences, for example finding the optimal skew-symmetric sequence for $N = 61$ (with $E = 230$ and $F = 8.09$) in a few

minutes. For sequences longer than 71 several randomized algorithms have been applied. [3] evaluate five local search algorithms. [16], [6] and [28] apply evolutionary strategies, which use no genetic recombination. [31] and [19] apply other evolutionary algorithms with recombination operators. [9] use a heuristic based on the observation that the interleaved sequences typically each have high merit factors.

N	Beenker	Golay	Wang	de Groot	Reinholz	Mühlenbein	Militzer	LSR
73	7.49	**7.66**						**7.66**
75	**8.25**	8.20						**8.25**
77	8.10	**8.28**						**8.28**
79	7.34	**7.67**						**7.67**
81	7.32	**8.20**		8.04	**8.20**	**8.20**	**8.20**	**8.20**
83	7.81	**9.14**						**9.14**
85	7.03	**8.17**						**8.17**
87	7.46	**8.39**						**8.39**
89	7.56	**8.18**						**8.18**
91	7.13	**8.68**						**8.68**
93	7.23	**8.61**						**8.61**
95	7.15	**9.42**						**9.42**
97	7.35	**8.78**						**8.78**
99	7.28	8.38						**8.49**
101	6.06	8.36	6.91	8.36	8.36	8.36	**8.82**	8.36
103	5.90	**9.56**	7.77		**9.56**	**9.56**	**9.56**	**9.56**
105	6.07	**8.89**	7.61			8.25	8.78	**8.89**
107	6.53	**8.46**				**8.46**	**8.46**	8.36
109	6.15	**8.97**			**8.97**	**8.97**	**8.97**	7.84
111	6.02	**8.97**				**8.97**	**8.97**	7.95
113	6.33	**8.49**			**8.49**	**8.49**	**8.49**	8.31
115	6.40					8.60	**8.88**	7.79
117	6.42					8.12	**8.71**	**8.71**
119	6.01					7.67	**8.02**	7.54

Fig. 4. Merit factors for large skew-symmetric sequences

Figure 4 shows results for these algorithms and LSR on large sequences. (Note that [3], and some papers citing them, give a skew-symmetric sequence of length 75 and merit factor 9.25; this is a misprint and should read 8.25 — see http://www.cbl.ncsu.edu/OpenExperiments/LABS/ for further details.) LSR was allowed one run of 10^9 steps per problem, each run taking several hours, with a small number of additional runs when results were poor. Best results for each N are shown in **bold**. Based on the available results LSR ranks above the algorithms of Wang, Beenker et al. and de Groot et al., roughly level with that of Mühlenbein, and below those of Reinholz, Golay & Harris, and Militzer et al. It found an improved sequence 5255212212A311224 for $N = 99$ with $E = 577$ and $F = 8.49$ and is not dominated by any other algorithm, but it is clearly not the

best. These results demonstrate that using relaxation within local search can be very competitive. In future work we hope to improve the results by evolutionary techniques or heuristics such as those of Brglez et al.

5 Conclusion

We have shown that a branch-and-bound algorithm can be transformed into a local search algorithm by using a randomized form of backtracking. The resulting algorithm is quite different from the more usual local search approach to optimization, in which the search space is the set of solutions to the problem, and the objective function is given by the problem. In our approach the optimization problem is reduced to a series of constraint satisfaction problems to be solved iteratively, with decreasing upper bounds on the given objective function. In each iteration the search space is the set of partial solutions whose cost is bounded by a relaxation, and the objective function to be minimized is the number of unassigned variables. This provides a clean way of exploiting relaxation during local search: essentially we perform local search in a branch-and-bound search space. The approach is generic and we expect it to find application to other optimization problems.

The new approach was evaluated on the notoriously difficult LABS optimization problem with positive results. Firstly, it is the first reported randomized algorithm to find optimal sequences (though another recent local search algorithm has also achieved this). Secondly, it does so much more quickly than branch-and-bound, as one would hope with local search. Thirdly, it performs competitively against several other randomized algorithms on large skew-symmetric sequences, finding an improved result. The algorithm is rather simple, leaving scope for future improvement by techniques such as population-based search.

It should be noted that the success of the approach depends partly on the cheapness of the relaxation computation. Expensive computation at each search step is only worthwhile if the penalty for making a wrong choice is heavy. Computation that is worth performing for backtrack search is less likely to be worthwhile for local search, and preliminary experiments with the Simplex algorithm on mixed integer programs were less successful. However, it is unsurprising that different algorithms require different trade-offs. In future work we intend to experiment with other optimization problems with known cheap relaxations.

There seems to be little work in the literature that is closely related to our approach. In [20] branch-and-bound is used to efficiently explore local search neighbourhoods. In [32] branch-and-bound is used as an anytime algorithm, referred to as *truncated depth-first branch-and-bound*, and shown to outperform local search on instances of the Asymmetric Traveling Salesman Problem (this problem could be an interesting application for LSR). Lagrangian relaxation has been used to guide local search by several researchers. However, we know of no other work that prunes local search space by using relaxation.

References

1. R. Backofen and S. Will. Excluding Symmetries in Constraint-Based Search. *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 1713, Springer-Verlag, 1999, pp. 73–87.
2. E. Balas, P. Toth. Branch and Bound Methods. *The Traveling Salesman Problem: a Guided Tour of Combinatorial Optimization*, Wiley, 1985.
3. G. Beenker, T. Claasen, P. Hermens. Binary Sequences With a Maximally Flat Amplitude Spectrum. *Philips J. of Research* vol. 40, 1985, pp. 289–304.
4. J. Bernasconi. Low Autocorrelation Binary Sequences: Statistical Mechanics and Configuration Space Analysis. *J. Physique* vol. 48, 1987, pp. 559–567.
5. F. Brglez, X. Y. Li, M. F. Stallman, B. Militzer. Reliable Cost Prediction for Finding Optimal Solutions to LABS Problem: Evolutionary and Alternative Algorithms. *Fifth International Workshop on Frontiers in Evolutionary Algorithms*, Cary, NC, USA, 2003.
6. C. de Groot, D. Würtz, K. H. Hoffmann. Low Autocorrelation Binary Sequences: Exact Enumeration and Optimization by Evolutionary Strategies. *Optimization* vol. 23, 1992, pp. 369–384.
7. E. C. Freuder, R. Dechter, M. L. Ginsberg, B. Selman, E. Tsang. Systematic Versus Stochastic Constraint Satisfaction. *Fourteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1995, pp. 2027–2032.
8. M. Golay. The Merit Factor of Long Low Autocorrelation Binary Sequences. *IEEE Trans. Inf. Theory* vol. 28 no. 3, 1982, pp. 543–549.
9. M. Golay, D. Harris. A New Search for Skew-Symmetric Binary Sequences with Optimal Merit Factors. *IEEE Trans. Inf. Theory* vol. 36, 1990, pp. 1163–1166.
10. E. A. Hirsch and A. Kojevnikov. Solving Boolean Satisfiability Using Local Search Guided by Unit Clause Elimination. *Seventh International Conference on Principles and Practice of Constraint Programming*, Cyprus, 2001.
11. N. Jussien, O. Lhomme. Local Search With Constraint Propagation and Conflict-Based Heuristics. *Artificial Intelligence* vol. 139 no. 1, 2002, pp. 21–45.
12. G. Lewandowski, A. Condon. Experiments With Parallel Graph Coloring Heuristics and Applications of Graph Coloring. *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science* vol. 26, D. S. Johnson, M. A. Trick (eds.), American Mathematical Society, 1996, pp. 309–334,
13. S. Lin, B. W. Kernighan. An Effective Heuristic for the Traveling Salesman Problem. *Operations Research* vol. 21, 1973, pp. 498–516.
14. S. Mertens. Exhaustive Search for Low-Autocorrelation Binary Sequences. *J. Phys. A: Math. Gen.* vol. 29, 1996, pp. L473–L481.
15. S. Mertens, C. Bessenrodt. On the Ground States of the Bernasconi Model. *J. Phys. A: Math. Gen.* vol. 31, 1998, pp. 3731–3749.
16. B. Militzer, M. Zamparelli, D. Beule. Evolutionary Search for Low Autocorrelated Binary Sequences. *IEEE Trans. Evol. Comp.* vol. 2, 1998, pp. 34–39.
17. S. Minton, M. D. Johnston, A. B. Philips, P. Laird. Minimizing Conflicts: A Heuristic Repair Method For Constraint Satisfaction and Scheduling Problems. *Constraint-Based Reasoning*, E. C. Freuder, A. K. Mackworth (eds.), MIT Press, 1994.
18. C. Morgenstern. Distributed Coloration Neighborhood Search. *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series in*

Discrete Mathematics and Theoretical Computer Science 26 D. S. Johnson, M. A. Trick (eds.), American Mathematical Society, 1996, pp. 335–357.
19. H. Mühlenbein H. Asynchronous Parallel Search by the Parallel Genetic Algorithm. *Third IEEE Symposium on Parallel and Distributed Processing*, IEEE Computer Society Press, 1991, pp. 526–33.
20. G. Pesant, M. Gendreau. A View of Local Search in Constraint Programming. *Second International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 1118, Springer-Verlag, 1996, pp. 353–366.
21. S. D. Prestwich. A Hybrid Search Architecture Applied to Hard Random 3-SAT and Low-Autocorrelation Binary Sequences. *Sixth International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 1894, Springer-Verlag, 2000, pp. 337–352.
22. S. D. Prestwich. Coloration Neighbourhood Search With Forward Checking. *Annals of Mathematics and Artificial Intelligence* vol. 34 no. 4, 2002, pp. 327–340.
23. S. D. Prestwich. SAT Problems With Chains of Dependent Variables. *Discrete Applied Mathematics* vol. 3037, 2002, pp. 1–22.
24. S. D. Prestwich. Combining the Scalability of Local Search with the Pruning Techniques of Systematic Search. *Annals of Operations Research* vol. 115, 2002, pp. 51–72.
25. S. D. Prestwich. Negative Effects of Modeling Techniques on Search Performance. *Annals of Operations Research* vol. 18, 2003, pp. 137–150.
26. S. D. Prestwich. Incomplete Dynamic Backtracking for Linear Pseudo-Boolean Problems. *Annals of Operations Research* vol. 130, 2004, pp. 57–73.
27. S. D. Prestwich, S. Bressan. A SAT Approach to Query Optimization in Mediator Systems. *Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, University of Cincinatti, 2002, pp. 252–259. To appear in *Annals of Mathematics and Artificial Intelligence*.
28. A. Reinholz. Ein Paralleler Genetischer Algorithmus zur Optimierung der Binären Autokorrelations-Funktion. Diplom thesis, Universität Bonn, 1993.
29. A. Schaerf. Combining Local Search and Look-Ahead for Scheduling and Constraint Satisfaction Problems. *Fifteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1997, pp. 1254–1259.
30. B. Selman, H. Levesque, D. Mitchell. A New Method for Solving Hard Satisfiability Problems. *Tenth National Conference on Artificial Intelligence*, MIT Press, 1992, pp. 440–446.
31. Q. Wang. Optimization by Simulating Molecular Evolution. *Biol. Cybern.* vol. 57, 1987, pp. 95–101.
32. W. Zhang. Depth-First Branch-and-Bound versus Local Search: a Case Study. *Seventeenth National Conference on Artificial Intelligence*, Austin, Texas, 2002, pp. 930–935.