Parallel Hybrid Method for SAT That Couples Genetic Algorithms and Local Search

Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano

***Abstract*—A parallel hybrid method for solving the satisfiability (SAT) problem that combines cellular genetic algorithms (GAs) and the random walk SAT (WSAT) strategy of greedy SAT (GSAT) is presented. The method, called cellular genetic WSAT (CGWSAT), uses a cellular GA to perform a global search from a random initial population of candidate solutions and a local selective generation of new strings. Global search is then specialized in local search by adopting the WSAT strategy. A main characteristic of the method is that it indirectly provides a parallel implementation of WSAT when the probability of crossover is set to zero. CGWSAT has been implemented on a Meiko CS-2 parallel machine using a two-dimensional cellular automaton as a parallel computation model. The algorithm has been tested on randomly generated problems and some classes of problems from the DIMACS and SATLIB test set.**

***Index Terms*—Cellular automata, genetic algorithms, parallel computation, satisfiability problem.**

I. Introduction

THE satisfiability (SAT) problem is commonly recognized as a fundamental problem in artificial intelligence applications, automated reasoning, mathematical logic, and related fields. Many practical problems including block-world planning problems [18], Boolean circuit synthesis problems [17], and circuit diagnosis [21] can be formulated as SAT problems. In automated reasoning, the SAT problem plays a key role because of its correspondence with deductive inferencing [4]. In computing theory, SAT is the core problem of the family of computationally intractable NP-complete problems [10]. Many of such problems have been identified as central in computing theory and engineering and may be transformed efficiently to SAT or to one of its variants, like MAX-SAT.

One of the most efficient, though incomplete, methods to solve SAT is random walk SAT (WSAT) [30], which is an extension of the popular local search method greedy SAT (GSAT) [29] that mixes a random walk strategy with greedy local search. This strategy overcomes the problem of getting trapped at local minima, typical of local search methods, by allowing uphill moves to solutions that could increase the number of unsatisfied clauses. WSAT is intrinsically a serial algorithm. The implementation on parallel systems is difficult to realize because the method works with only one string. A parallel implementation of the operations with considerable speedup is not an easy task.

Manuscript received December 7, 1999; revised October 10, 2000.

The authors are with the Institute of Systems and Informatics of the Italian National Research Council, c/o DEIS, Università della Calabria, 87036 Rende (CS), Italy (e-mail: folino@si.deis.unical.it; pizzuti@si.deis.unical.it; spezzano@si.deis.unical.it).

Publisher Item Identifier S 1089-778X(01)01537-5.

This paper presents a parallel hybrid method for solving SAT that combines cellular genetic algorithms (GAs) [37] and the random walk strategy of GSAT [29]. The method, called cellular genetic WSAT (CGWSAT), uses a cellular automata (CA) framework [36] that enables a fine-grained parallel implementation of GAs [11] through the *diffusion model* [26], [28]. The cellular GA executes a global search on a random initial population of candidate solutions. New strings are generated by crossing the current string with one of the neighbors with respect to a defined neighborhood relation. The search is then specialized locally by adopting the WSAT strategy as a mutation operator. The integration of a cellular GA and the WSAT approach has turned out to be very successful, as experimental results show, because it takes advantage of the features of both methods. In fact, because of the local interaction among the chromosomes, the CA model induces the formation of subpopulations of strings that evolve with a sufficient level of independence. Subpopulations diffuse information slowly, thus avoiding early entrapment in local minima; however, maintaining a steady diversity in the population over time is a very difficult task for a GA. Diversity can be lost quickly during recombination. High homogeneity among chromosomes diminishes the effectiveness of recombination and, by consequence, the possibility to generate new strings [11]. CGWSAT, by contrast, maintains a healthy diversity by allowing to substitute during recombination the current string with one of the offspring even if its fitness is worse than the parent and by adopting WSAT as a mutation operator.

A main characteristic of CGWSAT is that it indirectly provides a parallel implementation of WSAT, termed PWSAT. In fact, when the probability of crossover in CGWSAT is set to zero, every string works independently. This is equivalent to starting M WSAT procedures at the same time, where M is the size of the population.

A parallel version of the CGWSAT algorithm has been implemented on a Meiko CS-2 parallel machine. The algorithm has been tested on hard random 3-SAT [24], on some classes of problems of [16], and some benchmark problems of the SATLIB library [20] that encode block-world planning problems [19]. The experiments compare WSAT, PWSAT, and CGWSAT performances with respect to the number of iterations needed to obtain a solution and show the scalability of PWSAT and CGWSAT when a different number of processors is used. Furthermore, the speedups of PWSAT and CGWSAT are compared and, for one benchmark problem, the influence of the population size on the convergence of the method is also analyzed. The experimental results indicate the very good outcomes of CGWSAT for all the tested problems in terms of convergence and, for large problems, also in terms of speedup. Preliminary

1089–778X/01$10.00 © 2001 IEEE

results of the method appeared in [8] and [9]. In [8], however, the substitution of a string after the recombination process adopts a greedy strategy.

The paper is organized as follows. The SAT problem is formulated in Section II. Section III contains a detailed description of the WSAT algorithm. Section IV contains a brief description of GAs and a presentation of the CA model. Section V describes the new algorithm proposed. Section VI describes the parallel implementation of the method. Section VII discusses the experimental results. Section VIII, finally, compares CGWSAT with other existing approaches.

II. SAT Problem

Formally, the SAT problem can be formulated as follows. Let $U = \{u_1, \ldots, u_n\}$ be a set of n Boolean variables. A truth assignment for U is a function $t{:}U \rightarrow \{\text{true}, \text{false}\}$. Corresponding to each variable u are two literals u and $\neg u$. A literal u (resp. $\neg u$) is true under t iff $t(u) = \text{true}$ (resp. $t(\neg u) = \text{false}$). A set C of literals is called a clause and represents the disjunction (or logical connective) of these literals. A set of clauses is called a formula. A formula f is interpreted as a formula of the propositional calculus in conjunctive normal form (CNF) so that a truth assignment t satisfies a clause C iff at least one literal $u \in C$ is true under t. t satisfies f iff it satisfies every clause in f. For example, let $U = \{u_1, u_2\}$, $C_1 = u_1 \vee \neg u_2$, $C_2 = u_1 \vee u_2$, and $f = C_1 \wedge C_2$, then $t(u_1) = t(u_2) = \text{true}$ satisfies f. The SAT problem consists of a set of n variables $\{u_1, \ldots, u_n\}$ and a set of m clauses $C_1, \ldots, C_m$. The goal of the SAT problem is to determine whether or not there exists an assignment t of truth values to variables that makes the formula $f = C_1 \wedge \cdots \wedge C_m$ in CNF satisfiable. Among the extensions to SAT, MAX-SAT [10] is the most known. In this case, a parameter K is given and the problem is to determine if there exists an assignment t of truth values to variables such that at least K clauses are satisfied. SAT can be considered as a special case of MAX-SAT when K equals the number m of clauses.

Because of its importance, there has been interest in developing efficient methods to solve SAT problems. A raw classification of these methods subdivides them into *complete* and *incomplete* ones. A method is complete if it is always able to determine whether a formula is satisfiable or unsatisfiable. A method is incomplete if it does not always find a solution to SAT even if it exists. In this case, the algorithm stops without having found a satisfiable assignment, but it is not known if such an assignment exists. The main drawback of complete methods is that they are computationally intensive when the input size increases. For example, the Davis–Putnam algorithm, one of the fastest developed, performs very poorly with more than 400 variables [38].

More recently, local search algorithms [29], [30] have received attention because they have been applied successfully to certain hard classes of large SAT problems and they have been shown to outperform the best-known complete methods. Although SAT is intractable in the worst case, many instances of the problem are solved easily in practice [5]. The class of random k-SAT problems has been defined by generating instances with respect to three parameters: the number n of variables, the number m of clauses, and the length k of each clause. Mitchel *et al.* [24] showed that with $k = 3$ these problems are very hard when the generated instances are equally likely to be either satisfiable or unsatisfiable, i.e., they are neither underconstrained nor overconstrained. The crossover point occurs when m/n is equal to about 4.23.

Local search is a very efficient technique devised to solve NP-hard combinatorial optimization problems. Given an initial point, a local minimum is found by searching for a local neighborhood that improves the value of the object function. The SAT problem can be formulated as an optimization problem in which the goal is to minimize the number of unsatisfied clauses. Thus, the optimum is obtained when the value of the evaluation function equals zero, which means that all clauses are satisfied. The main problem in applying local search methods to combinatorial problems is that the search space presents many local optima and, consequently, the algorithm can get trapped at local minima. Some heuristics have been implemented to overcome this problem. They are based on allowing moves to a new neighborhood point of the local search space even if the value of the evaluation function increases.

One of the most popular local search methods for solving SAT is GSAT [29]. This algorithm starts with a randomly generated truth assignment. It then changes (flips) the assignment of the variable that leads to the largest decrease in the total number of unsatisfied clauses. Such flips are repeated until either a satisfying assignment is found or a preset of maximum number of flips is reached. This process is repeated as needed up to a maximum of *Maxtries* times. An extension of GSAT, referred to as WSAT, has been realized with the purpose of escaping from local optima by making upward moves that could increase the number of unsatisfied clauses. A detailed description of WSAT procedure is given in the next section.

III. WSAT Procedure

WSAT [30] is an extension of GSAT [29] that combines a random walk strategy with greedy local search. It differs from GSAT in the selection of the variable to flip. It restricts the choice of a randomly flipped variable to the set of variables that appear in unsatisfied clauses. The basic GSAT algorithm starts with a randomly generated truth assignment. It then changes (flips) the assignment of the variable that leads to the largest decrease in the total number of unsatisfied clauses. Such flips are repeated until either a satisfying assignment is found or a preset of maximum number of flips (*Maxflips*) is reached. This process is repeated as needed up to a maximum of *Maxtries* times. Fig. 1 shows the basic GSAT algorithm implemented by Kautz and Selman [20].

GSAT is a greedy algorithm that flips variables so that as many clauses as possible are satisfied. Note that if the chosen variable p is such that $\text{Diff}[p] > 0$, then the total number of unsatisfied clauses decreases. This is called a *downward* move. If $\text{Diff}[p] = 0$, then the total number of satisfied clauses remains constant; this is called a *sideways* move. Finally, if the flipped variable has $\text{Diff}[p] < 0$, then an *upward* move is performed in which the number of satisfied clauses decreases. Each iteration of the inner loop is referred to as a *flip* and each iteration of the outer loop as a *try*.

```
procedure GSAT
Input: a set of clauses CL, and integers Maxflips and Maxtries.
Output: a satisfying truth assignment of CL, if any is found.
begin
  for i := 1 to Maxtries do
    T := a randomly generated truth assignment;
    for j := 1 to Maxflips do
      if T satisfies CL then return T;
      for each variable p:
        let Make[p] = the number of clauses currently unsatisfied by T that would
        become satisfied if the truth value of p were reversed (flipped).
        let Break[p] = the number of clauses currently satisfied by T that would
        become unsatisfied if the truth value of p were flipped.
        let Diff[p] = Make[p] - Break[p];
      end for
      let MaxDiffList = list of variables with the greatest Diff;
      p := a random member of MaxDiffList;
      T := T with the truth assignment
        of p flipped;
    end for
  end for
return "no satisfying assignment found";
end.
```

Fig. 1. Basic GSAT algorithm.

The random walk strategy for SAT consists in flipping variables that appear in unsatisfied clauses. Several options are possible. In particular: 1) if no downward move is possible then, with probability f, flip a variable that appears in an unsatisfied clause instead of picking one from the MaxDiffList and 2) regardless of the number of unsatisfied clauses, with probability f, make a random walk move instead of a greedy move. Kautz and Selman [20] have experimented that the best performances are obtained by adopting the second option with probability f between 0.5–0.6. The random walk strategy is one of the fastest (though incomplete) implemented procedures for SAT: it can solve hard problems with thousands of variables in a few seconds.

The realization of a parallel implementation of WSAT is not simple. WSAT works with only one string of bits. The most expensive operation is constituted by the evaluation of the SAT of the clauses. This evaluation, if n is the number of variables, is done n times at each step because in order to find the variable that gives the greatest decrease of the number of unsatisfied clauses, every variable is flipped in turn and all the clauses are tested with that assignment of variables. However, in order to compute $\mathrm{Diff}[p]$, only the clauses where p appears are to be tested. In [33], it has been shown that for random k-SAT problems, each variable occurs, on average, in k^*m/n clauses. Thus, for each variable, only k^*m/n clauses must be processed. Furthermore, the serial implementation of WSAT uses an optimized data structure to store the clauses and a very efficient procedure to evaluate the satisfiablity. Thus, a parallel implementation giving a considerable speedup is not an easy task.

IV. Cellular Genetic Algorithms

GAs are naturally suited to be implemented on parallel architecture. Surveys on parallel GAs can be found in [3] and [35]. Two main approaches to parallel implementations of GAs have been proposed: the *island* model [22] and the *diffusion* model [26]. The island model divides the population into smaller subpopulations. A standard GA works on each partition and is responsible for initializing, evaluating, and evolving its own subpopulation. The standard GA is augmented with a migration operator that exchanges individuals among the subpopulations periodically. How many individuals migrates and how often migration should occur is an important debated problem [3].

In the diffusion model, each individual is associated with a spatial location on a low-dimensional grid. The population is considered as a system of active individuals that interact only with their direct neighbors. Different neighborhoods can be defined for the cells. The most common neighborhoods in the two-dimensional (2-D) case are the four-neighbor (*von Neumann neighborhood*) consisting of the north, south, east, and west neighbors and eight-neighbor (*Moore neighborhood*) consisting of the same neighbors augmented with the diagonal neighbors. Fig. 2 shows these two neighborhoods. Fitness evaluation is performed simultaneously for all the individuals and selection, reproduction, and mating take place locally within the neighborhood. Information diffuses slowly across the grid, forming clusters of solutions around different optima.

CA [36], [37] can be used as a framework to enable a fine-grained parallel implementation of GAs through the diffusion model. A CA is composed of a set of cells in a regular spatial lattice, either one-dimensional or multidimensional. Each cell can have a finite number of states. The states of all the cells are updated synchronously according to a local rule, called a transition function. The state of a cell at a given time depends only on its own state at the previous time step and the states of its *nearby* neighbors at that previous step. Thus, the state of the entire automaton advances in discrete time steps. The global behavior of the system is determined by the evolution of the states of all the cells as a result of multiple interactions.

A *cellular GA* [37] can be designed by associating to each cell of a CA two substates: one contains a chromosome and the other its fitness. At the beginning, a random population of strings is generated and the fitness is evaluated. Then, at each generation, the transition function associated with a cell selects the chromosome with the best fitness in the neighborhood. Crossover is applied to the current string and the selected string. After eval-

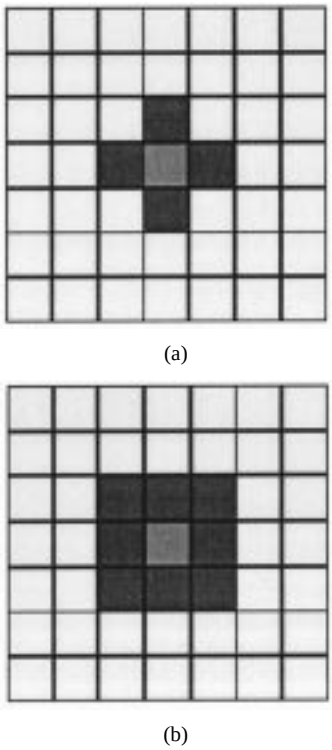


Fig. 2. (a) von Neumann and (b) Moore neighborhoods.

uating the offspring, if one of them has a better fitness than the current string, it becomes the current string. Next, the mutation operator with a given probability is applied to this string. This model, thus, solves the problem of the global selection of individuals, since reproduction and recombination are executed locally, and partially avoids the problem of population homogeneity, since the fittest individuals diffuse slowly across the grid. In essence, however, a cellular GA is a greedy technique and can present the same problem of premature convergence of standard GAs.

In contrast to cellular GAs, the random walk strategy allows for uphill moves to worse states and avoids entrapment in local minima. The adoption of this kind of strategy in the mutation operator guarantees the exploration of the search space toward different and, with a given probability, most promising regions. The Section V introduces the CGWSAT method, which integrates the cellular GA and WSAT.

V. CGWSAT Method

The SAT problem can be mapped directly onto a population of chromosomes of length n by interpreting every string of bits as an assignment of truth values to the set of n variables. The ith bit represents the truth value of the ith *Boolean* variable: *true* if the bit value is 1, *false* otherwise. The fitness function evaluates a string with respect to the number of unsatisfied clauses. A string whose fitness value is zero is a solution to the SAT problem because it means that there are no unsatisfied clauses. The CGWSAT method maps the population of strings into a 2-D square lattice. Every cell (i, j) of the CA contains a chromosome s and can only interact with the set of cells that neighbors itself. This defines a neighborhood relation. Every string

```
Let pcross, pmut be the crossover and mutation probability
for each cell i in CA do in parallel
  generate a random individual t_i,
  evaluate the fitness of t_i;
end parallel for
 while (not solution) do
 for each cell i in CA do in parallel
  generate a random probability value cross;
  if (cross > pcross) then {
   select the cell j, in the neighborhood of i,
    such that t_j has the best fitness;
   (u,v)=crossover(t_i,t_j );
    evaluate_fitness(u);
    evaluate_fitness(v);
   t_i= best_fitness(u,v); }
   generate a random probability mut;
   if (mut > pmut) then
    t_i = WSAT_Strategy(t_i);
    evaluate_fitness(t_i) ;
 end parallel for
end while
```

Fig. 3. Pseudocode of the CGWSAT algorithm.

s in the population is, thus, mated with the element w among the k neighbors, where k depends on the neighborhood relation chosen, with the best fitness. Each cell contains a transition function that defines the CGWSAT algorithm.

The pseudocode of the CGWSAT algorithm performed by every cell is shown in Fig. 3. The cellular GA adopts a 2-D toroidal grid using the Moore neighborhood. At first, a random population of chromosomes is generated and their fitness is evaluated. At each generation, the transition function associated with a cell selects the element among the k Moore neighbors having the best fitness. The genetic operators of *crossover* and *mutation* are then applied with *pcross* and *pmut* probability.

Crossover is a two-point crossover and is realized by selecting two positions h and l at random between one and n and two new strings u and v are generated by swapping all the bits of the two parents s and w contained in a neighborhood of h and l of length d (see Fig. 4). d is a parameter of the method. Thus, u (resp. v) receives $4d$ (resp. $4d-n$) bits from a parent and $4d-n$ (resp. $4d$) bits from the other parent. The current string s is then substituted by either u or v, the one having the best fitness.

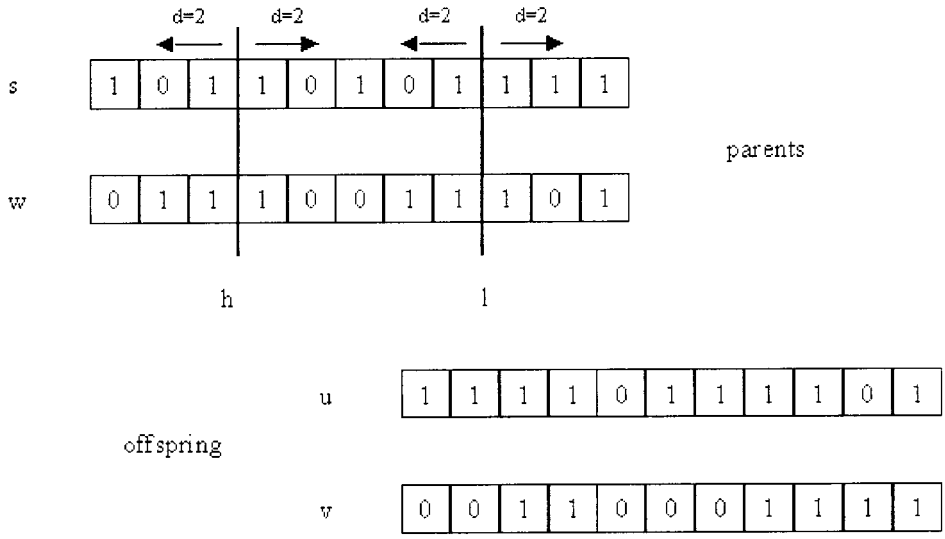


Fig. 4. Two-point crossover operator.

Notice that this kind of substitution differs from the one adopted by standard cellular GAs (i.e., the best of the offspring replaces the parent only if the latter has a worse fitness). Here, the best offspring always replaces the current string. The main motivation for this choice comes from the necessity to avoid that at the next generation the number of new strings be low. This kind of crossover, coupled with the selection of the mate based on the neighborhood relation, has a twofold advantage: first, the substitution of a string, even if the fitness could get worse, helps in maintaining diversity. Furthermore, it allows for the formation of subpopulations of strings having common characteristics inside the niches and relatively isolated among them. Every niche, thus, evolves with a sufficient grade of independence by exploring different portions of the search space. Subpopulations diffuse information among them very slowly. This strategy avoids the tendency for the population to homogenize, which would mean getting trapped in local minima, while the interaction among the chromosomes helps each other to improve their fitness.

Bit mutation is performed according to the mixed random walk strategy of [30]. With probability f, the variable with the best decrease in the number of unsatisfied clauses is flipped and, with probability $(1 - f)$, a variable that appears in some unsatisfied clause is picked and its assignment value changed. Both mutation and crossover operators have the role of bringing the population toward the best individuals and, at the same time, introducing new strings to maintain a good diversity in the extant individuals. In fact, they introduce uphill moves that could increase the number of unsatisfied clauses, but which, at the same time, employ a mechanism to escape from local minima by generating strings that belong to completely different portions of the search space. The combination of the new local selection mechanism along with specialized crossover and mutation operators allow the CGWSAT algorithm to reach a solution in a low number of generations.

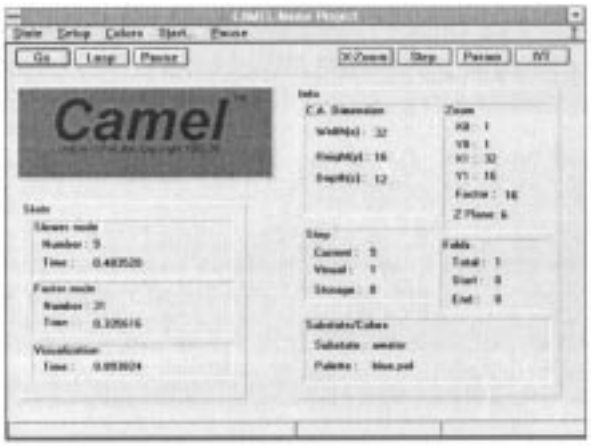
An important feature of CGWSAT is that it indirectly provides a parallel implementation of WSAT, named as PWSAT. In fact, when the probability of crossover in CGWSAT is set to zero, only the mutation operator works. This means that every string of the automaton is evolved independently without interacting with the neighboring cells by the WSAT procedure. Thus, each cell of the automaton implements a WSAT procedure at the same time, i.e., if M is the size of the population, M WSAT procedures start at the same time with M different initial strings. The algorithm terminates when the value of the fitness of at least a cell is equal to zero. This approach, of course, gives more chances to WSAT to be successful in obtaining the solution with a smaller number of flips. The recombination process, however, plays an important role in the reduction of the number of steps needed to reach the solution, as experimental results show.

VI. Parallel Implementation

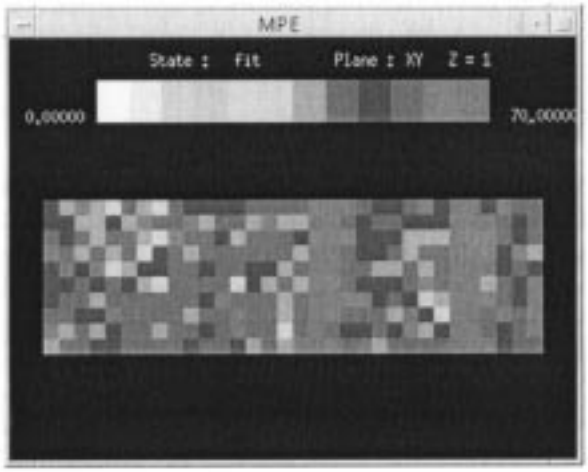
The CGWSAT method is naturally suitable for running on distributed-memory multiple-instruction multiple-data(MIMD) machines. It uses a grid of elements (population) of fixed size, where each cell performs a transition function on a string. An efficient parallel implementation of CGWSAT has been realized using the parallel computation environment provided by the CAMEL system [2]. CAMEL is an interactive parallel environment based on cellar automata model that allows to develop and execute cellular programs coded by CARPET language [34] on a parallel machine. CAMEL integrates computation, visualization, and control into one environment for the interactive steering of cellular applications. CAMEL consists of:

1) the CA *Engine*, comprising the CA Engine harness and the CARPET-defined CA model;
2) the graphical user interface (GUI) for editing, compiling, configuring, executing, and steering the computation;
3) the tool for the interactive visualization of the results.

The concurrent program that implements the software architecture of the CAMEL system comprises a set of *macrocell* processes, a *controller* process, and a GUI process. It has been implemented using C plus the standard MPI library and can be executed on different parallel machines such as the Meiko CS-2, IBM Sp2, CRAY T3E, and a LINUX cluster of PCs.



(a)



(b)

Fig. 5. Snapshot of a CGWSAT's run. (a) Parameter setting and (b) automaton visualization.

Each macrocell process runs on a single processing element of the parallel machine and holds a strip of cells of the CA. The macrocell processes implement the CA *Engine* as a single program multiple data parallel program. According to this approach, each macrocell process applies the transition function of the model to a subset of the model under the assumption that it holds all the data it requires locally. The CA Engine performs the evolution of the algorithm specified in the CARPET program. The synchronization of the automaton and the execution of the commands provided by the user through the GUI interface are carried out by the controller process. The macrocell processes are mapped automatically onto the nodes of the parallel machine according to a ring topology.

The CGWSAT algorithm is coded in CARPET. CARPET is a high-level language based on C with additional constructs to describe the rules of the state-transition function of a single cell of a CA. In CARPET, nothing must be specified about the parallel execution. In fact, after the CARPET compiled code is linked to the CAMEL environment it is mapped automatically on each computing node and executed in parallel to update the state of each cell.

By the GUI of CAMEL, shown in Fig. 5, the user can view the evolution of the algorithm by visualizing the output according to the defined visualization step and changing during execution the parameters that define the length of the neighborhood for the two-point crossover and the probability with which to perform crossover and mutation.

TABLE I
HARD RANDOM 3-SAT PROBLEMS, DIMACS, AND SATLIB TEST SUITE

Problem	Variables	Clauses	Iterations WSAT	Iterations PWSAT	Iterations CGWSAT
3-SAT	256	1100	1579	259	167
3-SAT	512	2201	3071	675	376
3-SAT	1024	4403	21717	4685	1890
3-SAT	2048	8806	25759	7951	3170
ii32b3	348	5734	2670	363	214
ii32c3	279	3272	3400	420	190
ii32d3	824	19478	32234	1377	1645
ii32e3	330	5020	1831	336	171
Jnh201	100	800	629	59	38
Jnh204	100	800	9614	206	108
jnh207	100	800	14070	226	161
jnh210	100	800	1342	89	62
g125.18	2250	70163	30693652	12695	6339
g250.15	3750	233965	15634	11348	5305
f1000	1000	4250	799518	81045	26344
bw_large.a	459	4675	27999	1068	658
bw_large.b	1087	13772	900049	11046	5523
bw_large.c	3016	50457	20386574	142860	74663
anomaly	48	261	1462	27	15
medium	116	953	4090	104	46
huge	6325	131973	51044	1207	833

The database of formulas is replicated entirely on each processor and each replica is accessible only by the cells allocated on the same processor. This solution allows evaluating in parallel the fitness of those cells belonging to different macrocell processes. In order to obtain an efficient implementation, the same data structure of WSAT for representing formulas has been used.

Experiments with different population size, number of processors, and parameters can be easily performed through the GUI that allows the user to quickly reconfigure the CAMEL system.

VII. EXPERIMENTAL RESULTS

The parallel implementation has been executed on a Meiko CS-2 parallel machine. The CS-2 is a distributed memory

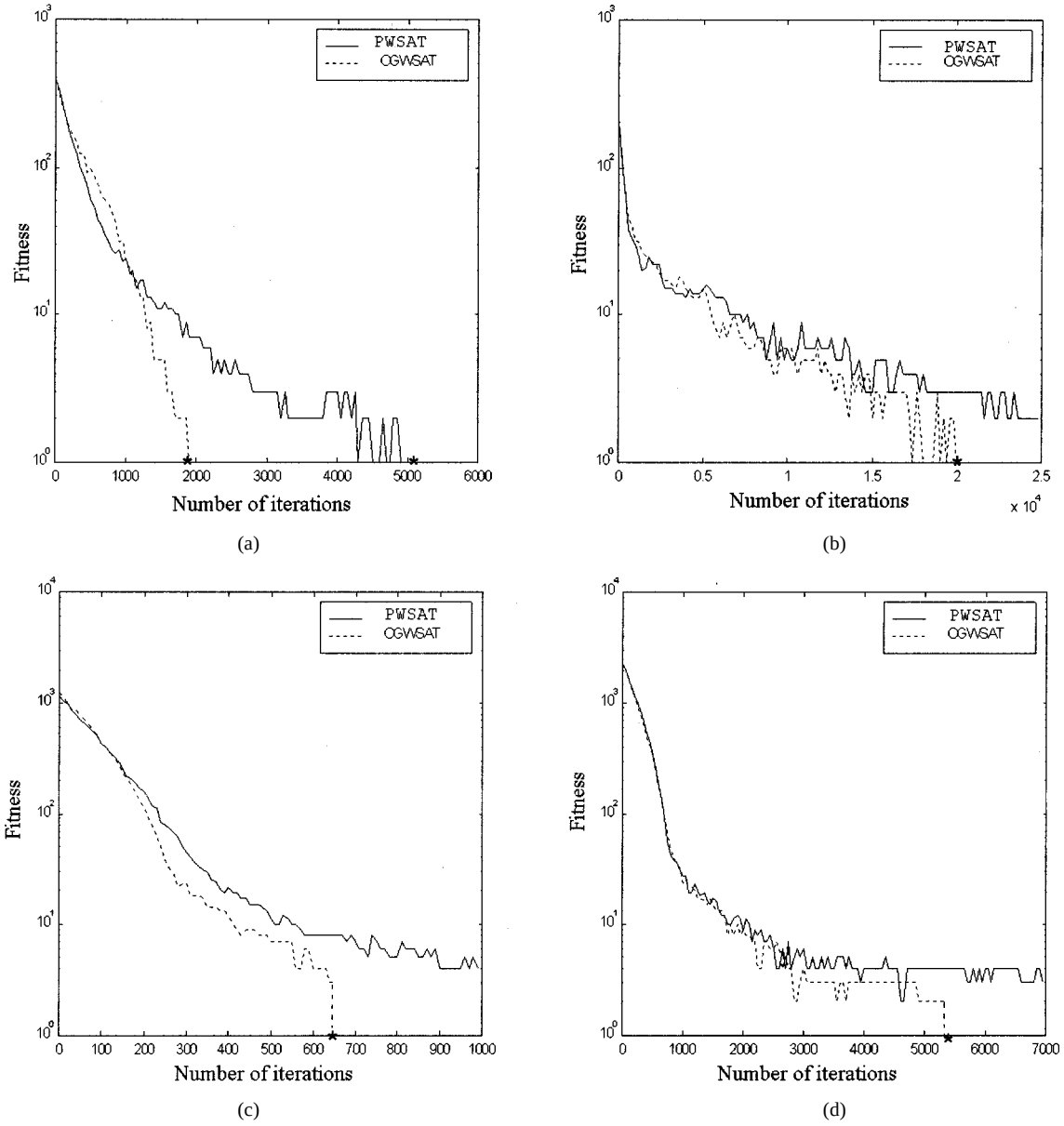


Fig. 6. Fitness values of a generic run for 3-SAT with (a) 1024 variables, (b) f1000, (c) huge, and (d) bw_large.b.

MIMD parallel computer. It consists of Sparc-based processing nodes running the Solaris operating system on each node. It resembles a cluster of workstations connected by a fast network. Each computing node is composed of one or more Sparc processors, a communication coprocessor, and the Elan processor that connects each node to a fat tree network built from Meiko 8 $\times$8 crosspoint switches. Our machine is a 12-processor CS-2 based on 200-MHz HyperSparc processors running Solaris 2.5.1. CGWSAT has been tested on hard randomly generated 3-SAT problems. In particular, tests with 256 to 2048 variables have been considered and on some problems of the DIMACS test suite [16] and the SATLIB library [20]. We used a population size of 320, a probability between 0.01 and 0.5 for crossover, depending on the difficulty of the problem, and between 0.9 and 1.0 for mutation and a length approximately equal to 10% the number of variables for two-point crossover.

The results are shown in Table I and represent data averaged over 30 independent runs. This table compares WSAT, PWSAT, and CGWSAT performances with respect to the number of iterations needed to obtain a solution. In particular, with regard to WSAT, we present both the number of flips when the sequential algorithm runs and the number of flips when the algorithm is executed in parallel on the CA. The parallel execution of WSAT, as already said, is obtained when the probability of crossover in CGWSAT is set to zero. The sequential version of WSAT is the one developed by Kautz and Selman [20] and available on the network (version 35). It was executed on a SUN workstation with two 200-MHz Sparc processors by using the option *-walk 0.5* in order to choose the random walk strategy. The experimental results indicate the very good outcomes of CGWSAT with respect to both WSAT and PWSAT for all the tested problems. As Table I shows, there is a dramatic decrease in the number of flips of PWSAT with respect to WSAT and a

TABLE II
EXECUTION TIME BY USING DIFFERENT NUMBERS OF PROCESSORS

			CGWSAT				PWSAT			
			number of processors				number of processors			
Prob.	**Var.**	**Cl.**	**1**	**2**	**4**	**8**	**1**	**2**	**4**	**8**
3-SAT	256	1100	77.54	39.82	20.86	11.58	36.98	18.76	9.27	4.67
3-SAT	512	2201	209.39	111.24	60.56	32.51	216.82	108.31	54.93	32.35
3-SAT	1024	4403	1217.03	607.25	328.41	173.34	3248.09	1509.55	770.79	424.62
3-SAT	2048	8806	3999.60	2069.72	1031.88	574.39	10478.13	5365.09	2598.46	1425.44
ii32b3	348	5734	809.37	418.64	216.70	117.38	167.84	84.32	43.37	22.86
ii32c3	279	3272	397.48	203.41	103.81	56.38	128.37	65.59	33.84	19.17
ii32d3	824	19478	1998.65	1000.23	507.17	337.32	1221.92	565.10	287.25	163.61
ii32e3	330	5020	538.71	274.31	140.66	76.18	133.95	66.25	34.79	19.61
Jnh201	100	800	20.85	10.88	5.38	2.95	7.49	4.04	2.05	1.06
Jnh204	100	800	55.12	28.19	14.11	7.95	29.99	15.53	7.84	3.96
jnh207	100	800	92.61	47.25	24.42	12.98	17.63	8.77	4.27	2.18
jnh210	100	800	6.74	3.39	1.69	0.89	10.18	5.40	2.42	1.33
g125.18	2250	70163	4792.58	2412.11	1265.84	665.64	8691.11	4345.92	2212.10	1129.50
g250.15	3750	233965	10543	5250	2672	1391	31315	15342	7926	4030
f1000	1000	4250	8035	4027	2098	1140	17348	8645	4351	2187
bw_large.a	459	4675	115.15	58.89	31.86	17.20	156.35	77.34	36.12	22.07
bw_large.b	1087	13772	1815.23	910.86	491.44	251.34	4072.57	2011.27	1010.73	527.19
bw_large.c	3016	50457	67269	31778	15950	9026	114778	54216	27540	14844
anomaly	48	261	0.35	0.182	0.112	0.07	0.34	0.188	0.101	0.052
medium	116	953	1.69	0.96	0.55	0.30	2.72	1.38	0.71	0.43
huge	6325	131973	2002.33	991.29	487.39	261.10	2834.14	1380.52	660.15	352.88

remarkable decrease in the number of flips of CGWSAT with respect to PWSAT. This means that recombination and the diffusion of information inside the neighborhood is significant and brings CGWSAT to a better convergence. Such a behavior is much more evident in Fig. 6, where the values of the fitness of both CGWSAT and PWSAT are shown during the computation of a generic run among the 30 performed for a 3-SAT problem with (a) 1024 variables, (b) f1000, (c) huge, and (d) bw_large.b.

In order to show the scalability of the method, Table II presents the execution time in seconds of CGWSAT and PWSAT when 1, 2, 4, or 8 processors are used. Notice that PWSAT is faster than CGWSAT only for small problems. As soon as the size of the problem increases, CGWSAT outperforms PWSAT in spite of the overhead due to the crossover operator. Fig. 7 shows the speedup of CGWSAT for the same problems for which the convergence has been shown. The speedup for all the problems is nearly linear. Finally, Fig. 8 shows the influence of the population size on the convergence of the method for the *bw_large.b* problem. This experiment demonstrates that as the population size increases the diversity in the population increases as well and the number of iterations is reduced significantly. In particular, if we double the population from 160 to 320, we obtain a reduction of about 4000 iterations and from 320 to 640, we obtain a reduction of about 1000 iterations. This means that the algorithm guarantees a good scaling behavior.

VIII. RELATED WORK

Many different approaches have been proposed and compared with WSAT [1], [6], [7], [13], [14], [23], [25], [27], [31]–[33]. They are based on different techniques, such as simulated annealing, Lagrangian-based methods, neural networks, GAs, and

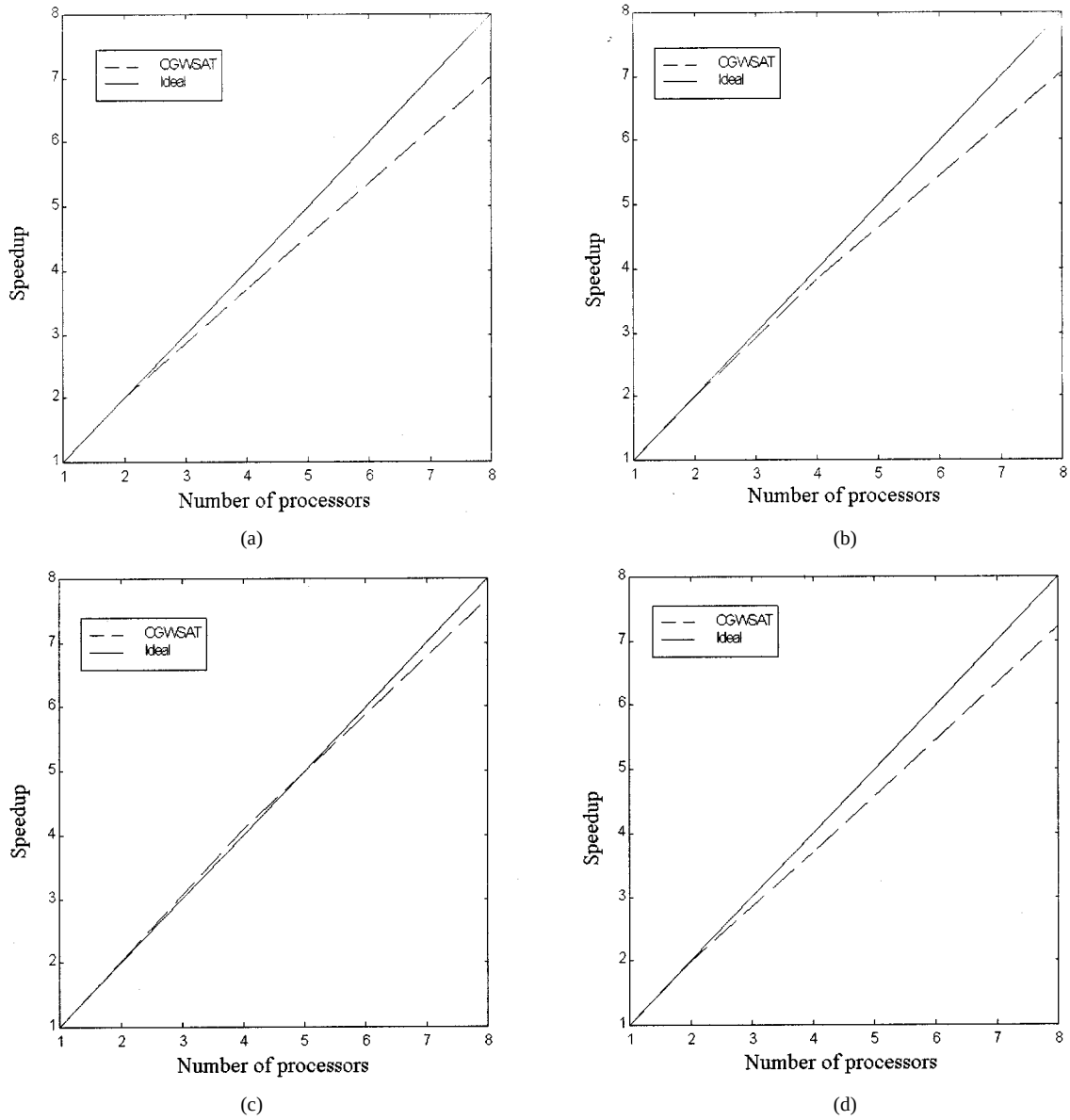


Fig. 7. Speedup for 3-SAT with (a) 1024 variables, (b) f1000, (c) huge, and (d) bw_large.b.

tabu search, and mix more than one heuristic. One of the first proposals for using GAs to solve SAT is contained in [6]. De Jong and Spears present a general strategy to solve NP-complete problems by means of GAs. They use a population of 100 individuals and show that GAs are efficacious search procedures though their performances are not better than other existing methods. To guarantee diversity in the population, the GA is restarted when the homogeneity exceeds a given threshold. Their experiments, however, consider only small-sized problems, with the number of variables varying between 10 and 90. In [32], Spears suggests *neural networks* to solve NP-complete problems and in [33], a *simulated annealing* algorithm called SASAT for the SAT problem. SASAT is similar to GSAT, but flips of variables are determined using an annealing schedule. The author compares SASAT and GSAT and claims that his method scales up better than GSAT and solves at least as many hard SAT problems as GSAT with a lower number of flips. Table III compares the experimental results of some of the test problems reported in [33] with CGWSAT. The superiority of our method with respect to SASAT is clear: the number of flips of SASAT is of the order of millions, while for CGWSAT it is no more than thousands.

Hao and Dorne [14] introduce a population-based evolutionary search method called MASK for finding models of SAT problems. The method evolves a population of partially instantiated binary strings. Such strings are called masks and are composed of values 1, 0, and *, which represents an undefined value. At first, only k of the n variables are fixed. A population of 2^k ordered masks is generated and evaluated. The better half of the population is chosen and each mask is split in two by fixing a free position to 0 and 1. These steps are repeated until there are no free positions. When all variables have been fixed, it can happen that none of the masks is a solution. In this case, the algorithm must be restarted with a different initial population. The authors show that their method outperforms the GA of [6]. A comparison with our

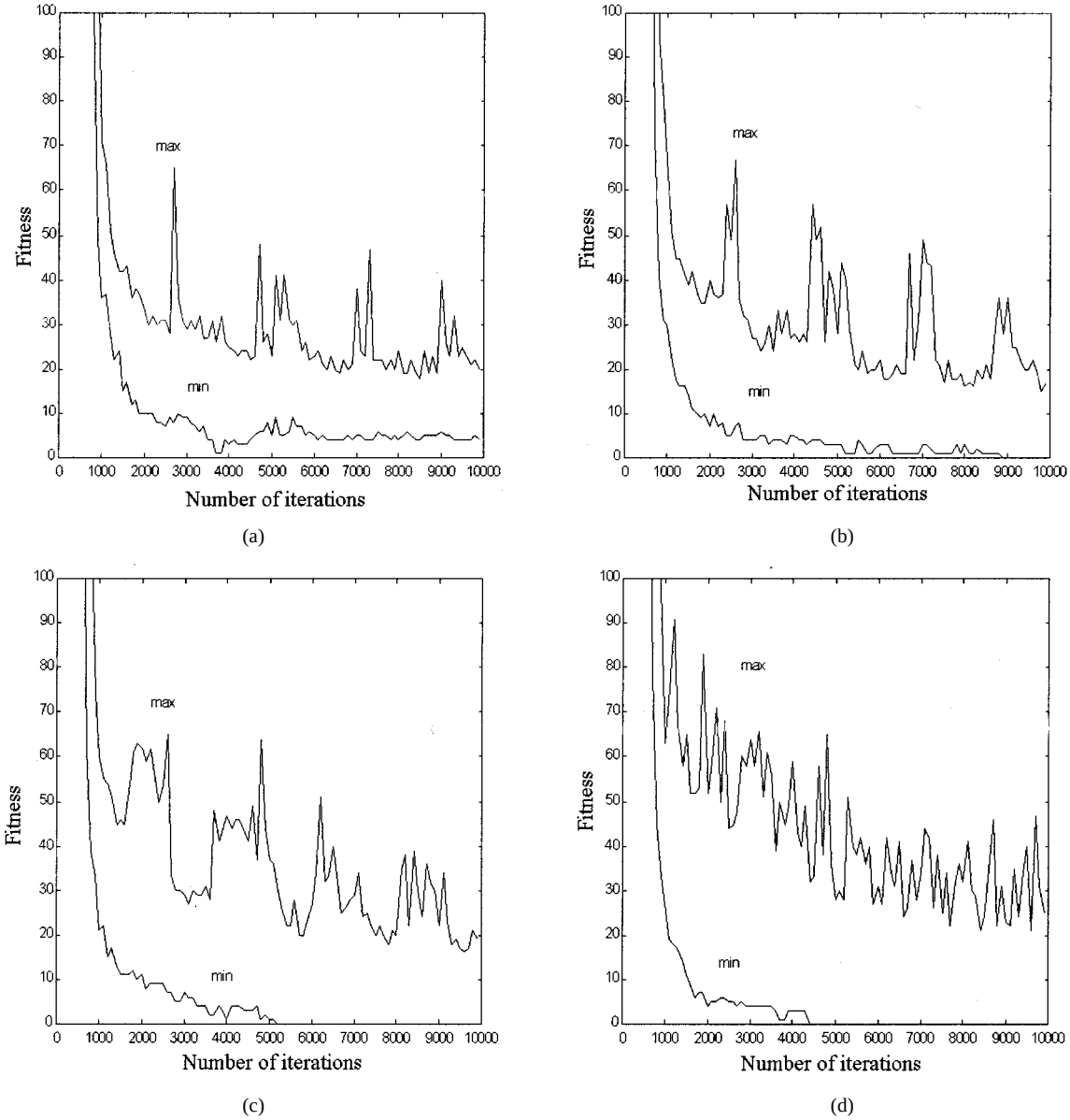


Fig. 8. Fitness values of a generic run for bw_large.b with population sizes (a) 80, (b) 160, (c) 320, and (d) 640.

approach is not possible because of the lack of available common results.

In [25] and [27], a greedy randomized adaptive search procedure (GRASP) for solving SAT [27] and MAX-SAT [25] problems is proposed. This heuristic consists of two phases, a construction phase and a local search phase. The construction phase generates a feasible solution by choosing the next element to be added from an ordered candidate list. The order is determined by a greedy function that measures the benefit of selecting an element. The local search phase tries to improve the constructed solutions. The method has been implemented in parallel and tested for MAX-SAT, but not for SAT.

Eiben and van der Hauw [7] present a GA to solve 3-SAT problems. Each clause is assigned a weight and the fitness of a chromosome is evaluated as the weighted sum of the evaluation value of the clauses. Furthermore, a mechanism called stepwise adaptation of weights (SAW), which enables the GA to modify the weights during the execution, is introduced. The SAW mechanism increases the GA performance and the authors show that their SAW-ing GA is superior to WSAT. Notice that the hard 3-SAT instances they use for testing range from only 30 to 100 variables. Similar approaches are proposed in [1] and [12]. In particular, in [1], the SAW mechanism is applied into an evolutionary algorithm instead of a GA, while in [12], the evolutionary algorithm uses a *refined fitness function* that tries to capture the distance of solution candidates from an optimum. They define a *refining function* $r{:}\{0, 1\}^n \rightarrow [0, 1)$ and add it to the fitness function. In such a way, the value of the fitness function is not just the number of unsatisfied clauses, but ranges over the interval $[0 \cdots m]$. The refining function is $adaptive$ in the sense that it depends on the progress of the evolutionary search. The authors define different refining functions and compare the results with WSAT and the SAW-ing GA of Eiben and van der Hauw [7] on the same test problems.

TABLE III
SASAT AND CGWSAT ON THE DIMACS TEST SUITE

Problem	Variables	Clauses	Iterations SASAT	Iterations CGWSAT
ii32b3	348	5734	2.2E6	237
ii32c3	279	3272	7.5E6	195
ii32d3	824	19478	4.6E5	1635
ii32e3	330	5020	2.0E6	192
g125.18	2250	70163	43007	6576
g250.15	3750	233965	6193	5153
f1000	1000	4250	1.5E8	26902

In [23], a GA FLIPGA combined with a local search procedure is presented. At each iteration, the local search procedure evolves the population in order to obtain a population of local optima after which the standard crossover and mutation operators are applied. The heuristic is similar to that of WSAT: it flips a variable if the number of satisfied clauses increases. However, the number of variables flipped is not one, rather the flipping process is stopped only when the number of satisfied clauses does not increase any more. The authors compare their method with those of [1] and [12], again on the same test problems, and show that their method is better than both [1] and [12]. The comparison is done with respect to the average number of evaluations to solution (AES), i.e., the average number of fitness evaluations. Because of the flip heuristic included inside the GA, the cost of fitness evaluation after a flip must be added to AES. Such a cost, denoted by FES (flip cost in terms of number of fitness evaluations to solution), has been computed as FES = (number of flips * 3)/n. Thus, the number of flips can be obtained as FES $* n/3$. A comparison between our approach and FLIPGA is not very sound because of the small size of test problems. However, Table IV reports the number of flips of FLIPGA and those of CGWSAT on these instances of 3-SAT. The population used by FLIPGA is ten, while we used a population of 80 chromosomes, i.e., a grid 10×8. This dimension is necessary because otherwise the neighborhood relation is not meaningful and the niche effect could be lost. Considering that we used a bigger population to obtain the number of flips of CGWSAT, we have multiplied the number of iterations by 80, i.e., the number of chromosomes in the population. The results show that CGWSAT outperforms FLIPGA. Notice that the authors observed a minor role of crossover for the performance of their method: for some instances, it is essential to find a solution for others it has no influence. With regard to CGWSAT, on the contrary, we already showed the crucial role of crossover to speed up the method.

We must remark that the comparison of our method with the approaches reported is not sound because our method has been realized on a parallel machine with the aim to deal with large-sized problems.

TABLE IV
FLIPGA AND CGWSAT ON THE 3-SAT INSTANCES

Problem	Variables	Clauses	Iterations FLIPGA	Iterations CGWSAT
1	30	129	1100	640
2	30	129	17810	3200
3	30	129	7120	1280
4	40	172	2373	1120
5	40	172	2066	880
6	40	172	20186	6720
7	50	215	3483	1040
8	50	215	6916	1680
9	50	215	185050	38640
10	100	430	4322500	24960
11	100	430	52333	5440
12	100	430	52133	8720

REFERENCES

[1] T. Bäck, A. E. Eiben, and M. E. Vink, "A superior evolutionary algorithm for 3–SAT.," in *Proceedings of the 7th Annual Conference on Evolutionary Programming*, W. Porto, N. Saravanan, D. Waagen, and A. E. Eiben, Eds. Berlin, Germany: Springer-Verlage, 1998, pp. 125–136.

[2] M. Cannataro, S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia, "A parallel cellular automata environment on multicomputers for computational science," *Parallel Comput.*, vol. 21, no. 5, pp. 803–824, 1995.

[3] E. Cantú-Paz, "A summary of research on parallel genetic algorithms," Illinois Genetic Algorithm Lab., Univ. Illinois Urbana-Champaign, Urbana, IL, Tech. Rep. 950076, July 1995.

[4] C. Chang and R. C. Lee, *Symbolic Logic and Mechanical Theorem Proving*. New York: Academic Press, 1973.

[5] S. A. Cook and D. Mitchell, "Finding hard instances of the usatisfiability problem: A survey," in *Satisfiability Problem: Theory and Applications*, D. Du, J. Gu, and P. Pardalos, Eds. Providence, RI: American Mathematical Soc., 1997, vol. 35, pp. 1–18.

[6] K. A. De Jong and W. M. Spears, "Using genetic algorithms to solve NP-complete problems.," in *Proceedings of the Third International Conference On Genetic Algorithms*, J. D. Schaffer, Ed. San Mateo, CA: Morgan Kaufmann, 1989, pp. 124–132.

[7] A. E. Eiben and J. K. van der Hauw, "Solving 3-SAT with adaptive genetic algorithms," in *Proceedings of the 4th IEEE Conference on Evolutionary Computation*. Piscataway, NJ: IEEE Press, 1997, pp. 81–86.

[8] G. Folino, C. Pizzuti, and G. Spezzano, "Solving the satisfiability problem by a parallel cellular genetic algorithm," in *Proc. 24th IEEE Euromicro Conf.*, Vasteras, Sweden, Aug. 1998, pp. 715–722.

[9] G. Folino, C. Pizzuti, and G. Spezzano, "Combining cellular genetic algorithms and local search for solving satisfiability problems," in *Proc. 10th IEEE Int. Conf. Tools with Artificial Intelligence*, Taipei, Taiwan, Nov. 1998, pp. 192–198.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability. A guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.

[11] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[12] J. Gottlieb and N. Voss, "Improving the performance of evolutionary algorithms for the satisfiability problem by refing functions.," in *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, A. E. Eiben *et al.*, Eds.. Berlin, Germany, 1998, pp. 755–764.

[13] J. Gu, "Global optimization for satisfiability (SAT) problem," *IEEE Trans. Knowledge Data Eng.*, vol. 6, pp. 361–381, June 1994.
[14] J. K. Hao and R. Dorne, "A new population—based method for satisfiability problems," in *Proceedings of the 11th European Conference on Artificial Intelligence*. New York: Wiley, 1994, pp. 135–139.
[15] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: Univ. Michigan Press, 1975.
[16] D. S. Johnson and M. A. Trick, Eds., "Second DIMACS implementation challenge," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. Providence, RI: American Mathematical Soc., 1996, vol. 26.
[17] A. P. Kamath, N. K. Karmarkar, K. G. Ramakrishnan, and M. G. C. Resende, "A continuous approach to inductive inference," *Math. Program.*, vol. 57, no. 2, pp. 215–238, Nov. 1992.
[18] H. A. Kautz and B. Selman, "Planning as satisfiability," in *Proc. Eur. Conf. Artificial Intelligence* , Aug. 1992, pp. 359–363.
[19] H. A. Kautz, D. McAllester, and B. Selman, "Encoding plans in propositional logic," in *Proc. Int. Conf. Knowledge Representation*, Nov. 1996, pp. 374–384.
[20] SATLIB: The satisfiability library [Online]. Available: http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/
[21] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 4–15, Jan. 1992.
[22] W. N. Martin, J. Lienig, and J. P. Cohoon, "Island (migration) models: Evolutionary algorithms based on punctuated equilibria," in *Handbook of Evolutionary Computation*, T. Bäck, D. B. Fogel, and Z. Michalewicz, Eds. Bristol, U.K.: Inst. of Physics, 1997, p. C.6.4.
[23] E. Marchiori and C. Rossi, "A flipping genetic algorithm for hard 3-SAT problems," in *Proceedings of the Genetic and Evolutionary Computation Conference*. San Mateo, CA: Morgan Kaufmann, 1999, pp. 393–400.
[24] D. Mitchell, B. Selman, and H. Levesque, "Hard and easy distributions of SAT problems," in *Proc. 10th Nation Conf. Artificial Intelligence*, July 1992, pp. 459–465.
[25] P. M. Pardalos, L. Pitsoulis, and M. G. C. Resende, "A parallel GRASP for MAX-SAT problems," in *Proc. Workshop Applied Parallel Computing Industrial Problem Optimization*, Lyngby, Denmark, Aug. 1996, pp. 575–585.
[26] C. Pettey, "Diffusion (cellular) models," in *Handbook of Evolutionary Computation*, T. Bäck, D. B. Fogel, and Z. Michalewicz, Eds. Bristol, U.K.: Inst. of Physics, 1997, p. C.6.3.
[27] M. G. C. Resende and T. A. Feo, "A GRASP for satisfiability," in *Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, D. S. Johnson and M. A. Trick, Eds. Providence, RI: American Mathematical Soc., 1996, vol. 26, pp. 499–520.
[28] G. Rudolph and J. Sprave, "A cellular genetic algorithm with self-adjusting acceptance threshold," in *Proceedings of the First IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*. Stevenage, U.K., 1995, pp. 365–372.
[29] B. Selman, H. Levesque, and D. Mitchell, "A new method for solving hard satisfiability problems," in *Proc. 10th Nation Conf. Artificial Intelligence*, 1992, pp. 440–446.
[30] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *Proc. 12th Nation Conf. Artificial Intelligence*, 1994, pp. 337–343.
[31] Y. Shang and B. W. Wah, "A discrete Lagrangian-based global-search method for solving satisfiability problems," *J. Global Optimiz.*, vol. 12, no. 1, pp. 61–99, Jan. 1998.
[32] W. M. Spears, "Using neural networks and genetic algorithms as heuristics for NP-complete problems," M.S. thesis, Dept. Comput. Sci., George Mason Univ., Fairfax, VA, 1990.
[33] W. M. Spears, "Simulated annealing for hard satisfiability problems, in cliques, coloring, and satisfiability 'Second DIMACS Implementation Challenge'," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. Providence, RI: American Mathematical Soc., 1996, vol. 26, pp. 533–558.
[34] G. Spezzano and D. Ialia, "A high-level cellular programming model for massively parallel processing," in *Proceedings of the 2nd International Workshop on High-Level Programming Models and Supportive Environments*. Los Alamitos, CA: IEEE Comput. Soc. Press, 1997, pp. 55–63.
[35] M. Tomassini, "Parallel and distributed evolutionary algorithms: A review," in *Evolutionary Algorithms in Engineering and Computer Science*, K. Miettinen, M. Mkel, P. Neittaanmki, and J. Periaux, Eds. New York: Wiley, 1999, pp. 113–133.
[36] T. Toffoli and N. Margolus, *Cellular Automata Machines A New Environment for Modeling*. Cambridge, MA: MIT Press, 1986.
[37] D. Whitley, "Cellular genetic algorithms," in *Proc. Fifth Int. Conference on Genetic Algorithms*, R. K. Belew and L. B. Booker, Eds. San Mateo, CA: Morgan Kaufmann, 1993, p. 658.
[38] H. Zhang, "SATO: an efficient propositional prover," Tech. Rep., Dept. Comput. Sci., Univ. Iowa, Iowa City, IA, 1994.

Gianluigi Folino was born in Catanzaro, Italy, on May 27, 1972. He received the Laurea degree in engineering from the University of Calabria, Calabria, Italy, in 1997.

Since 1999, his research has been supported by an INFN fellowship at the Institute of Systems and Informatics of the Italian National Research Council, Rende, Italy. His current research interests include cellular automata, parallel computing, genetic algorithms, and genetic programming.

Clara Pizzuti received the Laurea degree in mathematics from the University of Calabria, Calabria, Italy.

She was with the Consortium for Research and Applications of Information Technology, where she was Member of the Research Division Staff participating in Esprit projects on advanced logic based systems. Since 1994, she she has been with the Institute of Systems and Informatics of the Italian National Research Council as a Researcher. She is also a Lecturer with the Department of Computer Science, University of Calabria, Calabria, Italy. Her research interests include deductive databases, abduction, automated reasoning, genetic algorithms, genetic programming knowledge discovery in databases, and data mining.

Giandomenico Spezzano received the Laurea degree in engineering from the University of Calabria, Calabria, Italy, in 1980.

From 1981 to 1996, he was a Researcher with the Consortium for Research and Applications of Information Technology, where he led various research projects in the distributed and parallel computing area. Since 1996, he has been a Senior Researcher with the Institute of Systems and Informatics of the Italian National Research Council in the area of parallel computation. He is also a Lecturer with the Department of Electronics, Computer Science, and Systems, University of Calabria, Calabria, Italy, since 1994. He has authored or coauthored over 90 papers and two books. His current research interests include tools for parallel computers, computational steering, parallel genetic programming, cellular automata, and their implementation on parallel architectures.

Dr. Spezzano is a member of the IEEE Computer Society and the ACM.