ELSEVIER Physica D 115 (1998) 293–312

PHYSICA D

Problem solving during artificial selection of self-replicating loops ★

Hui-Hsien Chou [a,*], James A. Reggia [b,1]

[a] The Institute for Genomic Research, 9712 Medical Center Drive, Rockville, MD 20850, USA
[b] Department of Computer Science and Institute for Advanced Computer Studies, A.V. Williams Bldg., University of Maryland, College Park, MD 20742, USA

Received 21 July 1997; received in revised form 5 September 1997; accepted 24 September 1997
Communicated by A.M. Albano

Abstract

Past cellular automata models of self-replication have generally done only one thing: replicate themselves. However, it has recently been demonstrated that such self-replicating structures can be programmed to also carry out a task during the replication process. Past models of this sort have been limited in that the "program" involved is copied unchanged from parent to child, so that each generation of replicants is executing exactly the same program on exactly the same data. Here we take a different approach in which each replicant receives a distinct partial solution that is modified during replication. Under artificial selection, replicants with promising solutions proliferate while those with failed solutions are lost. We show that this approach can be applied successfully to solve an NP-complete problem, the satisfiability problem. Bounds are given on the cellular space size and time needed to solve a given problem, and simulations demonstrate that this approach works effectively. These and other recent results raise the possibility of evolving self-replicating structures that have a simulated metabolism or that carry out useful tasks. Copyright © 1998 Elsevier Science B.V.

Keywords: Self-replication; Self-organization; Cellular automata; SAT problem; Artificial selection

1. Introduction

Cellular automata can be viewed as dynamical systems in which both time and space are discrete. John von Neumann and Stanislaw Ulam first conceived of using cellular automata to study the logical organization of self-replicating structures [14]. In von Neumann's and subsequent cellular automata models two-dimensional space is divided into cells, each of which can be in one of n possible states. At any moment most cells are in a distinguished "quiescent" or inactive state whereas the other cells are said to be in an active state. A self-replicating structure or "machine" is represented as a configuration of contiguous active cells. At each instance of simulated time, each cell (each component) follows a set of cellular automata rules to determine its next state as a function of its current state and the state of its immediate neighbor cells. Thus, any process of self-replication captured in a model like this must be an emergent behavior arising from the strictly local interactions that occur. Based solely on these concurrent local interactions, a self-replicating structure goes through

★ Supported in part by NASA award NAGW-2805.
* Corresponding author. E-mail: hhchou@tigr.org.
[1] Tel.: 301 405 2686; fax: 301 405 6707; e-mail: reggia@cs.umd.edu.

PII S0167-2789(97)00237-6

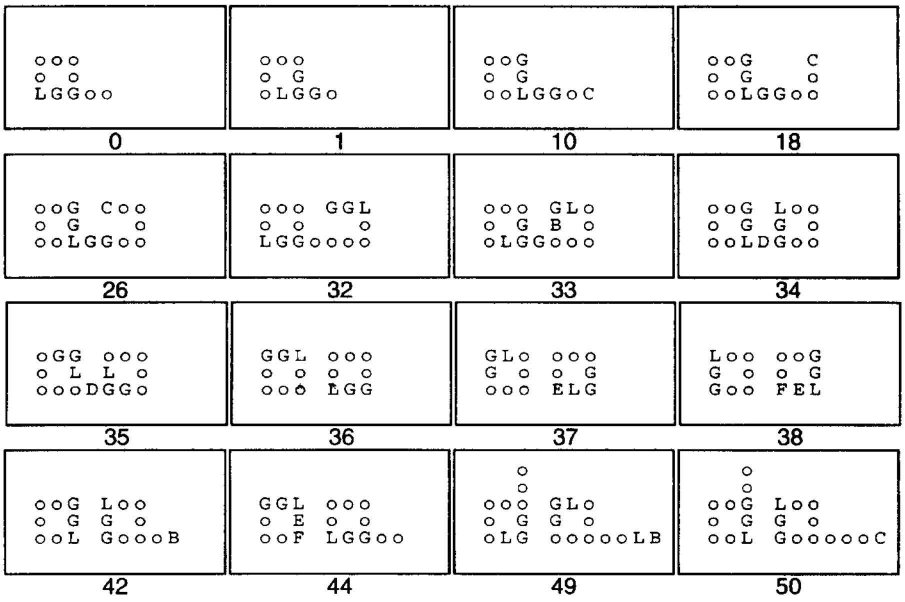

Fig. 1. A typical self-replicating loop. At time 0 (upper left), the 10-component loop is introduced into the otherwise empty cellular space. Three of the loop elements are signals *G* for "grow" and *L* for "turn left", embedded in the loop of o's that form a signal path over which signals flow. The signals represent instructions on how to self-replicate, forming the program GGL (read off clockwise from the loop). These instructions repeatedly circulate counterclockwise around the loop, with a copy going out of the loop's lower right "arm" each time the instructions go by. As the instructions reach the rightmost tip of the arm, they are "executed". In this case the instruction sequence GGL indicates to grow twice (i.e., extend the signal path two cells to the right), and then turn left. After executing these instructions four times, a second duplicate loop appears by time 44. Over the next several time steps, each of these loops begins the process of replicating again in adjacent cells. This replication process continues indefinitely, causing a growing colony of loops to form in the cellular space.

a sequence of steps to construct a duplicate copy of itself, possibly displaced and perhaps rotated.

Von Neumann's original self-replicating structure is a complex "universal computer-constructor". Much subsequent work has focused on creating simpler replicants [4], including qualitatively simpler structures referred to as *self-replicating loops* [6]. Self-replicating loops have a readily identifiable stored "instruction sequence" that serves two purposes: as instructions that are interpreted to direct the construction of a replica, and as uninterpreted data that are copied onto the replica. Subsequent work led to progressively simpler and smaller loops [2,11], co-evolving architectures [12], automatic programming of the rules [8], and emergence of self-replication from a randomly initialized cellular automata space [3]. An example of self-replicating loop is illustrated in Fig. 1.

Past cellular automata models of self-replication have solely been concerned with replication. Recently, however, the idea of programming self-replicators to do more than just replicate has been proposed [10,13]. The basic idea is that the signal (instructions) directing a structure's replication can be extended in some fashion to solve a specific class of problems while replication occurs. The motivation for such *programmed replicators* is that they provide a novel, massively parallel computational environment that may lead over the long term to powerful, very fast computing methods. However, past models of this sort have been limited in that the "program" involved is copied unchanged from parent to child, so that each generation of replicants is executing exactly the same program on exactly the same data. Here we take a different approach in which each replicant receives a distinct partial solution that is modified during replication. Under artificial selection, replicants with promising solutions proliferate while those with failed solutions are lost. We show for the first time that this approach can be applied successfully to solve an NP-complete problem. Bounds are given on the cellular space size and the time needed to solve a given problem, and simulations demonstrate that this approach works effectively.

The specific problem we consider is the satisfiability problem (SAT problem), a classic example of an NP-complete problem [5]. Given a boolean predicate like

$$\mathcal{P} = (\neg x_1 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3),$$

the SAT problem asks "what assignment of boolean values to the binary variables x_1, x_2 and x_3 can satisfy this predicate", i.e., can make this predicate evaluate to True? The predicate $\mathcal{P}$ here is in conjunctive normal form (CNF), where each part of the predicate surrounded by a pair of parentheses is called a *clause*. A SAT problem is usually called as an m-SAT problem if there are m boolean variables in a clause of its predicate. Therefore, the above example is a 2-SAT problem.[2]

The specific use of SAT problems here is motivated by recent suggestions that recombinant DNA methods can be applied to this problem [1,7]. This *DNA computer* technique separates DNA sequences representing possible SAT assignments based on the clauses of a given SAT predicate. During each separation cycle all current DNA sequences are tested against one clause of the predicate, and unsatisfying sequences are removed. In a limited number of steps, the satisfying assignment, if any, can be found in the final DNA strands. The DNA computer technique suggests that we may be able to use self-replicating loops to solve the SAT problem in a similar manner. If many self-replicating loops are generated, each carrying a different trial boolean assignment for a SAT problem, and selection pressure is imposed upon these loops so that only those carrying satisfying variable assignments will survive, it should be possible to achieve the same results that a DNA computer can produce. That is the idea behind using self-replicating loops to solve the SAT problem described in this paper.

For example, the self-replicating loop shown in Fig. 1 requires only a quarter of its cells to encode its replication steps, leaving the rest of its state o cells unused. These unused cells will be used to encode bit sequences representing boolean assignments for a SAT problem. A selection mechanism is built into the cellular automata rule set so that only those loops which carry satisfying bit sequences will survive in the cellular space through time.

The basic procedure is as follows. Let the self-replicating loops use their extra o cells to carry bit sequences representing variable assignments for a given SAT problem. The replication process is designed so that each new child loop will have one bit explored which differs from its parent. Initially, there will be one loop in the cellular space carrying a completely unexplored bit sequence. In the end there will be one or more loops in the cellular automata space, each carrying a different fully explored bit sequence, assuming a solution exists. During this *enumeration process* possible assignments for the given SAT predicate are generated, while monitoring mechanisms built into the cellular space look for loops carrying unsatisfying assignments, removing them. This provides for a *selection process*. Combining the enumeration and selection processes, many SAT assignments can be explored in a short time using self-replicating loops in a fashion somewhat analogous to the DNA technique.

2. Examples

Several examples of solving SAT problems using self-replicating loops are now given; the underlying rules are summarized in Section 3. As is often done with models of this type to avoid edge effects, these examples are in a cellular space with periodic boundary conditions, where opposite edges are considered to be adjacent (i.e., the cellular space forms a torus).

Fig. 2(a) shows the enumeration results for a self-replicating loop carrying three binary bits. In the initial loop, unexplored bits are represented by the symbol A. These A's replace some of the o's in the old

[2] Note that without loss of generality, 2-SAT problems are chosen as examples due to their simplicity. Although 2-SAT problems can actually be solved in polynomial time and are not truly NP-complete, the method presented in this paper does not take advantage of that and is equally applicable to 3-SAT or other hard problems.

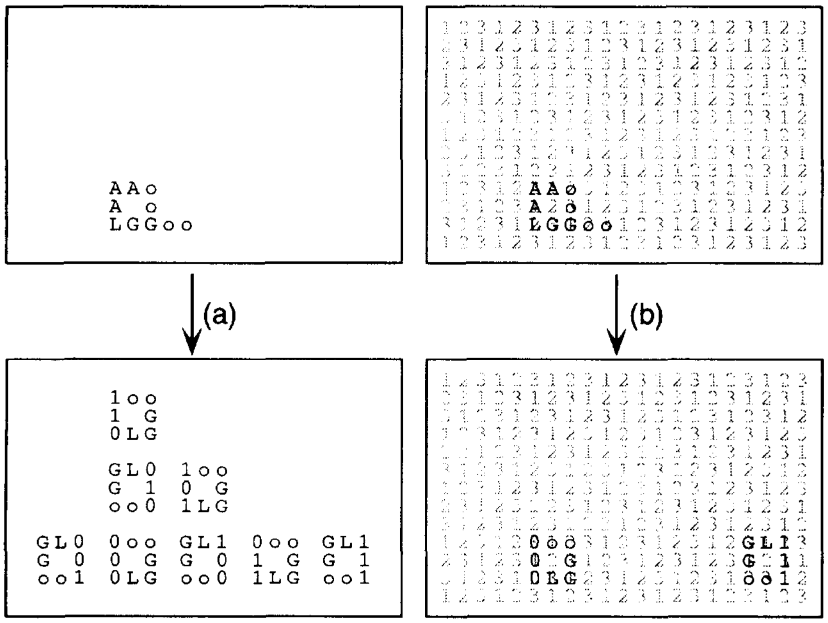

Fig. 2. The enumeration and selection of satisfying boolean assignments by self-replicating loops. (a) A 3 × 3 self-replicating loop can carry three binary bits (locations marked by AAA) and enumerate all eight possible boolean assignments in three generations. (b) When "monitors" are introduced into the cellular space, an artificial selection process takes place. In three generations only those loops carrying satisfying assignments to the boolean predicate $\mathcal{P}$ are immune from termination by monitors and survive in the cellular space. Monitors are represented by light gray digits. There are three different kinds of monitors, each representing one clause of the original predicate $\mathcal{P}$.

self-replicating loops (compare Fig. 1). Explored bits are represented by either digit 0 or 1 in the loops. The bit sequence a loop carries is read off starting right after the L symbol; therefore, the lower left loop in Fig. 2(a) carries the sequence 001 and the top-most loop carries 011. The sequence 001 means "assign $x_1 = 0$, $x_2 = 0$, and $x_3 = 1$". In Fig. 2(a), only the enumeration process is at work. Without the selection process, in three generations all eight possible boolean assignments for a three-variable SAT predicate are generated and carried by eight loops in the cellular space. Loops will stop growing once they have explored all of their bits (they contain no A's). Since exploration of bits is done one bit each generation, and with each exploration step we get a different bit in the parent and child loops while inheriting previously explored bits from the parents unmodified, all possible boolean assign will be found with the enumeration process, if no collisions occur between loops.

To remove those loops that do not satisfy a SAT predicate, "*monitors*" are spread throughout the cellular space. Each monitor tests a particular clause of the SAT predicate. If the condition a monitor is looking for is verified, it will "destroy" the loop on top of it. For the example predicate $\mathcal{P}$, three classes of monitors, each testing for one of the following conditions, are planted in the cellular space: $x_1 \wedge \neg x_3$, $\neg x_1 \wedge x_2$, and $\neg x_2 \wedge x_3$. These are just the negated clauses of $\mathcal{P}$. If any one clause of $\mathcal{P}$ is *not* satisfied, then the whole predicate will not be satisfied. A monitor will destroy/remove a loop on top of it if its corresponding clause is found to be unsatisfied by the bit sequence carried by the loop. This detection process is done in linear time since essentially each monitor is just a finite automata machine; the bit sequence passing through its cell can be viewed as a regular expression that is to be recognized. With enough monitors in the cellular space, they can remove all unsatisfying solutions. Note that a self-replicating loop must cover each

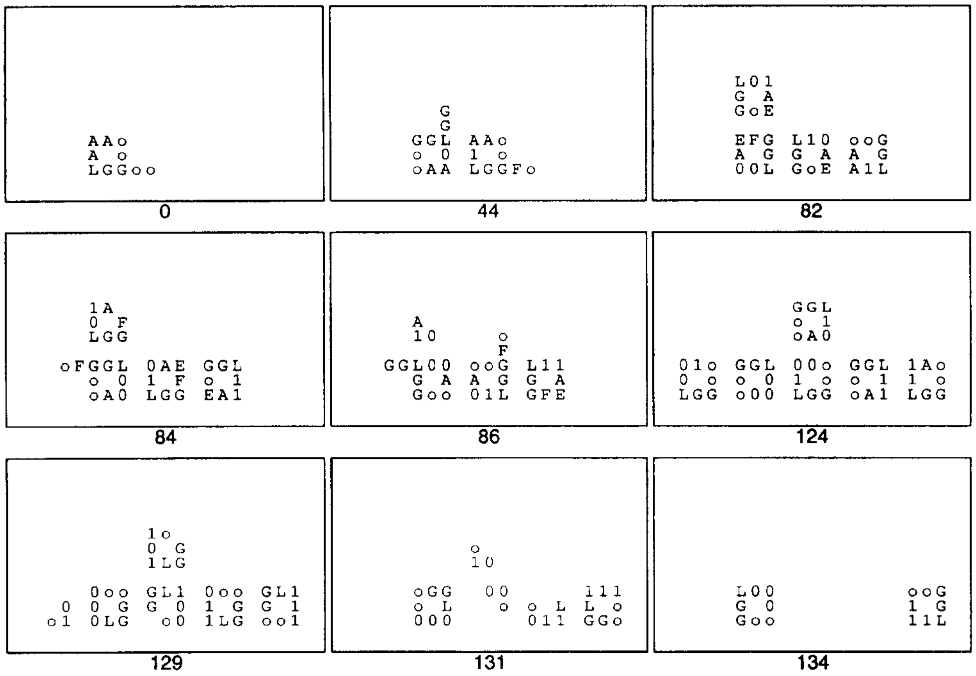

Fig. 3. Progressive stages of the enumeration and selection process of 3 × 3 loops. At time 0 the initial loop is placed in the cellular space, which carries unexplored binary bits represented as AAA. Monitors checking for the three clauses of predicate $\mathcal{P}$ are spread in the cellular space but are not shown. At time 44 the first replication cycle has completed and two loops are now present in the cellular space. The first binary bit has been changed into 0 and 1 in the two descendent loops by the enumeration process. At time 82 the second replication cycle has completed and there are four loops in the cellular space. Starting at time 84 the top loop is being destroyed (note the missing corner cell of the loop). Its bit sequence '01A' does not satisfy $x_1 \vee \neg x_2$, the second clause, and therefore it is being erased by the monitor underneath its top-right corner. At time 86 the erasing process continues while the other loops start their next replication cycle. At time 124 the third (also the last) replication stage is completed and there are six loops in the cellular space. Four of these do not survive the monitors very long and are erased (times 129 and 131). Finally, we are left with two satisfying assignments 000 and 111 at time 134.

monitor at least once, otherwise some clauses will not be checked against the loop. This is maintained in our model.

Fig. 2(b) displays the result of replication starting from the same initial loop as in Fig. 2(a), but this time the selection process is turned on. Monitors looking for the three conditions above are spread throughout the cellular automata space. They are shown in light gray in the background of the figure. There are three types of monitors designated by symbols 1, 2 and 3 for the three conditions to test. The symbol of a monitor tells which condition it is testing. Like all the other cellular automata activities, the testing procedures are defined in the cellular automata rule set. Basically, a monitor keeps track of the boolean bit sequence flowing through its position, and destroys/removes a loop if its condition is recognized. The monitor rule set works independently of the self-replication and enumeration rule sets for self-replicating loops. The monitor will interfere with the self-replicating process only if an unsatisfying loop is found and the loop is to be destroyed. Thus, in Fig. 2(b) after three generations there are only two loops left in the cellular automata space instead of eight as seen in Fig. 2(a). These two loops carry the only two satisfying boolean assignments for the original SAT predicate $\mathcal{P}$, which are 000 and 111.

Some intermediate steps of the enumeration and selection process for the same 3 × 3 loop are shown in Fig. 3. Since monitors do not change once distributed, for the sake of clarity, they are not explicitly displayed in this figure. Starting with one initial loop carrying a totally unexplored bit sequence AAA, sequences 0AA

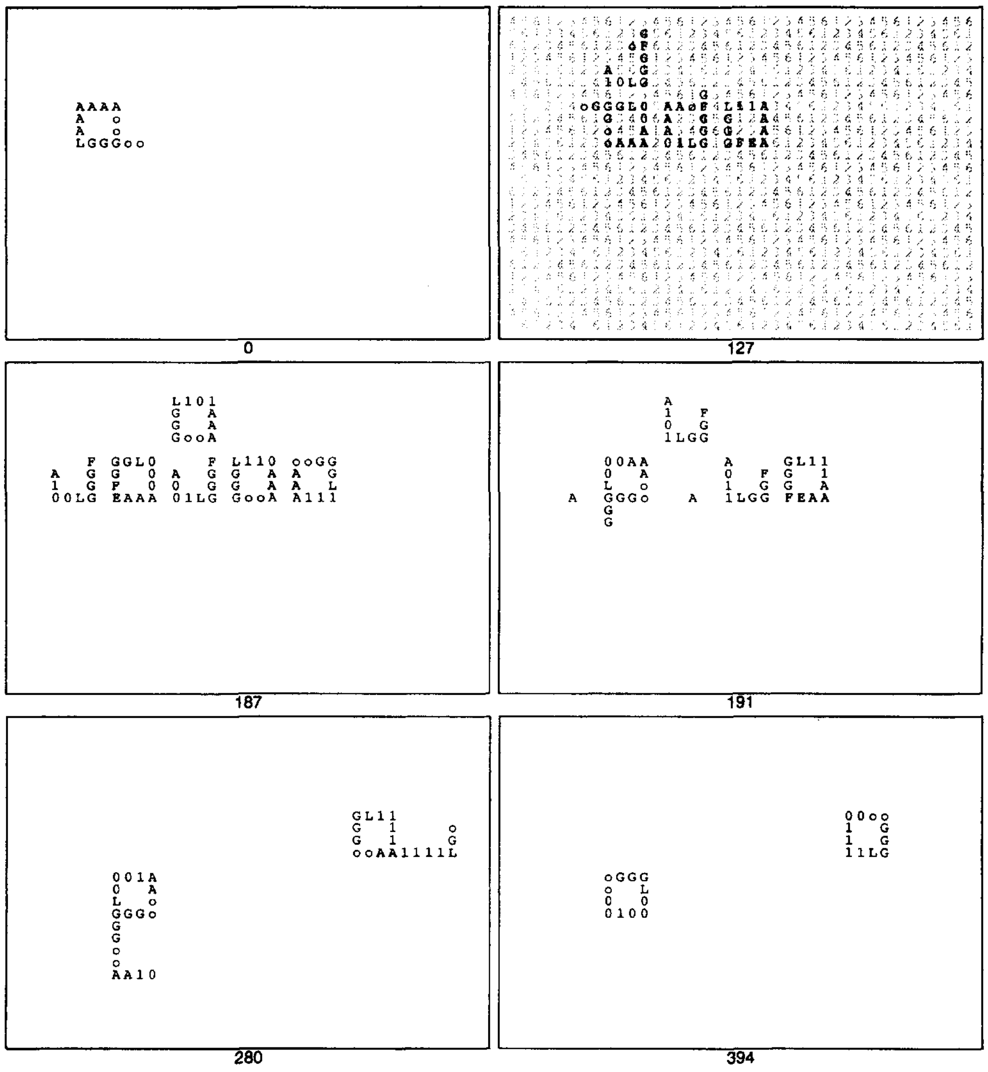

Fig. 4. Solving a six variable SAT problem using 4 × 4 loops. The initial loop carrying six unexplored bits AAAAAA for predicate $Q = (\neg x_1 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_4 \vee x_4) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_6)$ is placed in the cellular space at time 0. At time 0 a single monitor also exists in the cellular space outside the viewed region; it results in a population of monitors by time 127 as shown (these monitors persist unchanged but are not shown at later times). New loops are generated and some are removed by monitors if they carry unsatisfying bit sequences. There are only two satisfying boolean assignments for predicate Q, 000100 and 111100. They are found at time 394, about six generations of the 4 × 4 loop, read off in a clockwise manner from the final loops. Monitors (six different ones; one per cell) are everywhere in this cellular space but are not shown for clarity.

in the parent loop and 1AA in the child loop appear in the first generation of descendent loops. In the second generation, two new child loops carrying 01A and 11A appear; the two parents (from the first generation) now carry 00A and 10A. In the third and final generation, we potentially should get four more new loops 011, 111, 001 and 101. The four parents would carry 010, 110, 000 and 100, so all together we should get all eight possible values for a 3 bit binary sequence. However, it can be seen that some of these are never generated after the second generation. The topmost loop in Fig. 2(a) was never generated in Fig. 3 because its parent loop (the one below it in Fig. 2(a)), which still carries unexplored binary bits, is removed. Since it has been found (by the monitors) that the partially explored bits 01A in this parent loop do not satisfy one of the clauses, there is no need to explore further since all of its descendents will carry the same binary bits.

Another example, this time or solving a six-variable SAT problem using 4 x 4 loops, is illustrated in Fig. 4. The goal is to satisfy the six variable predicate

$$\mathcal{Q} = (\neg x_1 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_4 \vee x_4) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_6).$$

Self-replicating loops are able to find the only two satisfying boolean assignments out of a total of 64 possible assignments in 394 iteration steps, or six generations. It generally takes n generations to solve an n-variable SAT problem.

The two examples above are for solving 2-SAT problems, i.e., their clauses have two boolean variables. The cellular automata rule set can be easily modified to solve SAT problems with longer clauses; there is no inherent limitation on the clause length. Fig. 5 shows the result of solving a four-variable, 4-SAT problem with four clauses:

$$\begin{aligned}\mathcal{R} = {} & (x_1 \vee x_2 \vee x_3 \vee x_4) \\ & \wedge (x_2 \vee \neg x_2 \vee x_3 \vee \neg x_4) \\ & \wedge (\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4) \\ & \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4).\end{aligned}$$

Here 12 of the 16 possible variable assignments are found to satisfy the predicate in four generations.

3. A behavioral look at the cellular automata rules

We now consider a behavioral description of the cellular automata rule set for self-replicating loops that solve SAT problems. The complete rule set and further information, including the simulation environment for this work, can be obtained from the Internet.[3] Each

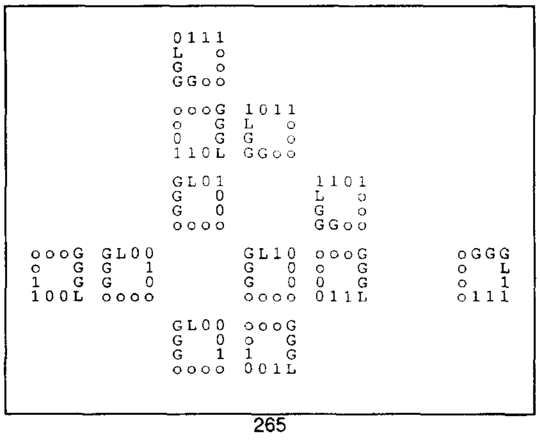

Fig. 5. Solving a four variable, 4-SAT problem using 4 × 4 loops. There are only four boolean assignments that fail to satisfy predicate $R = (x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4)$. These are 0000, 0101, 1010 and 1110. They are removed before time 265, about four generations of the 4 × 4 loop.

cell is split into *fields* to simplify rule set programming. The SAT rule set uses the Moore neighborhood, where a cell determines its next state based on its current state and that of its eight surrounding neighbors. Note that all loops are larger than such a neighborhood, so the self-replication and problem-solving capabilities of the loops considered here are emergent properties of the cellular automata space.

3.1. Fields

The cellular automata rule set for solving SAT problems is based on cells that are split into six *fields* (see Fig. 6). A four-bit *code* field carries the primary instructions directing self-replication and problem-solving. It is primarily the values of this field that have been displayed in figures so far. The *code* field signals used to direct replication are:

- · : the quiescent state; an empty cell;
- o: the building block of the loop; these form a pathway over which signals move;
- G: the growth signal directing expansion of the signal pathway;
- L: the left turn command, indicating that the expanding signal path should turn left;

[3] http://www.cs.umd.edu/~hhchou/download.html.

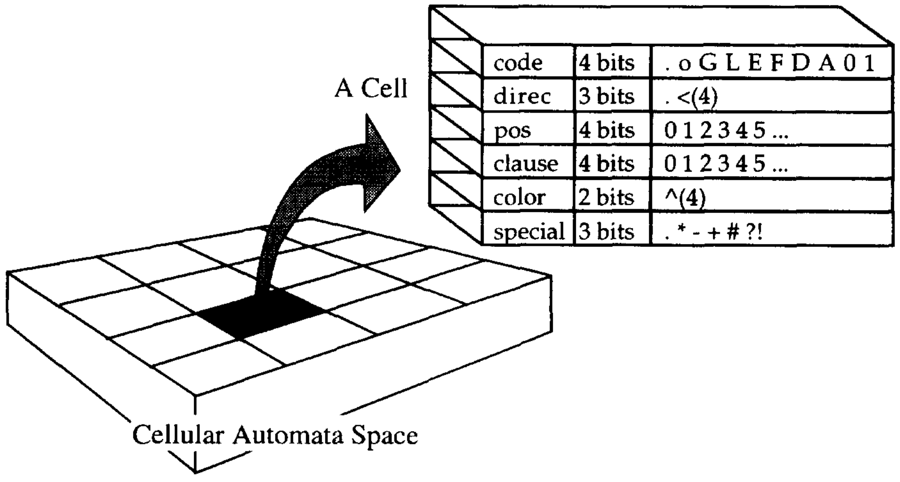
A Cell
code | 4 bits | . o G L E F D A 0 1
direc | 3 bits | . <(4)
pos | 4 bits | 0 1 2 3 4 5 ...
clause | 4 bits | 0 1 2 3 4 5 ...
color | 2 bits | ^(4)
special | 3 bits | . * - + # ?!
Cellular Automata Space

Fig. 6. The data fields used in solving SAT problems. Listed are the bit depth and valid state symbols of each field. Symbols with a "(4)" after them actually represent four states: the four orientations of the symbol. Quiescent states are represented by a period '.' in the rule set, but are not displayed in figures when showing the cellular space configuration.

– E, F: two signals that direct the sprouting of a new arm; and
– D: the detach signal that separates the parent and child loops.

The following code field states have no effect on the replication process, but instead direct the enumeration of SAT bit sequences:

– A: An unexplored SAT binary bit; when explored during a replication cycle it becomes 0 in the parent loop and 1 in the child loop.
– 0: The explored SAT binary bit 0 representing an assignment of zero/false to the corresponding boolean variable in the SAT predicate.
– 1: The explored SAT binary bit 1 representing an assignment of one/true to the corresponding boolean variable in the SAT predicate.

Five additional fields, usually not shown in the figures so far, are present. The 3-bit *direc* field indicates the direction that information flows over the signal pathway. Valid states for the *direc* field are the quiescent state plus the four different orientations of the signal >, which point so that signals in a loop always flow in a counterclockwise direction. A *pos* field (position field) that marks which boolean variable in a SAT predicate is represented by a bit in the SAT bit sequence carried in a loop. This field indicates the identity of solution bit variables and permits the monitors to check for unsatisfying bit sequences. Valid values for this field are the numbers 0, 1, 2, ..., representing variable one, variable two,... in the SAT predicate. Markers in this field flow with values in the code field. The *clause* field encodes which clause in the SAT predicate the monitor is to check. A cell with a particular non-quiescent clause value will look for any bit sequence passing over it which represents an unsatisfying boolean assignment for the clause it checks. Cellular automata rules are defined to control this checking process. If the monitor can find such an unsatisfying sequence, it will destroy the loop carrying that sequence, by setting a flag in the special field. A *color* field marks the expansion direction of each loop. It is used to detect collisions of self-replicating loops during their expansion. Since two loops expanding in the same direction (thus having the same color) never collide in this model, a collision of loops can be detected by the different color of cells. Finally, a 3-bit *special* field is used to encode miscellaneous situations that arise. For example, an $*$ indicates that a branching sequence EF is to be generated, # indicates that a loop is being erased (i.e., it carries an unsatisfying solution), and ! indicates that a collision between two loops has occurred.

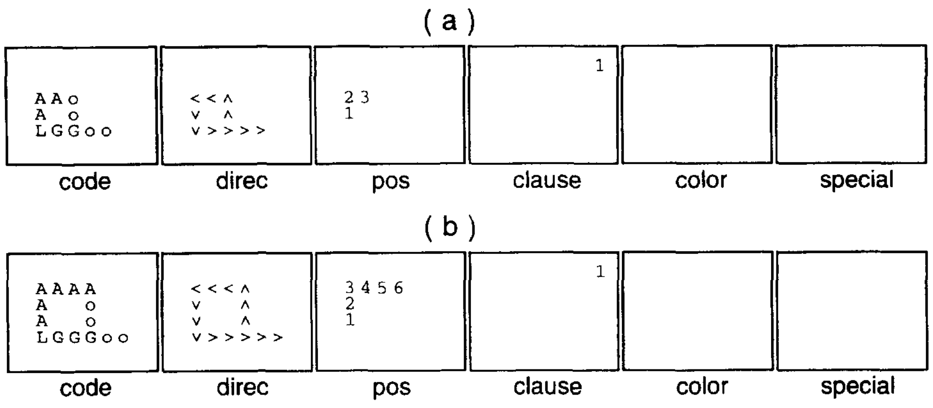

Fig. 7. Examples of the initial configuration (a) for solving a three-variable SAT problem and (b) for solving a six-variable SAT problem.

3.2. *Initialization*

At the beginning of a simulation, a single loop is put into the cellular automata space. The code field is initialized as in Fig. 2, the direc field is initialized to direct signal flow correctly, and A's representing an unexplored bit sequence are placed in the loop body. The pos field under these A bits is set to represent the corresponding variables in the SAT predicate. The clause field of a cell close to the initial loop is set to a non-zero value, usually one, to represent a *seed* monitor. After the simulation begins, the rule set will spread different monitors evenly throughout the entire cellular automata space starting from the seed monitor.[4] The color field and the special field in all cells are in the quiescent state at the beginning. Examples of the initial cellular space, one for solving a three-variable SAT problem and the other for a six-variable SAT problem, are given in Fig. 7. While these six fields are always present, only the code field is always displayed in subsequent examples of this paper for brevity. This is because the majority of the functions of the SAT rule set is controlled by the code field.

The SAT rule set has two independent functionalities which have been introduced above: the enumerating process by self-replicating loops to generate all possible boolean assignments to a SAT predicate, and the selection process by monitors that allow only those loops carrying satisfying assignments to survive at the end. The enumeration process is a straightforward extension of past rules governing self-replication [3,6,11] to include new rules that change an unexplored bit A into bit 0 in the parent and into bit 1 in the child loop after a replication. The selection process is new; its rule set essentially implements an automaton that recognizes regular expressions, and it has its own states to represent different stages of the checking process. The bit sequence carried by a loop is the string this automaton is checking against. We first look at the enumeration process and then the selection process in the following sections.

3.3. *Self-replication of loops*

The signal flow pattern in this model is similar to that used in previous self-replicating loops, except the direction of signal flow is now explicitly defined by the direc data field. Consider the code field and its associated direc field content as shown in Fig. 8. Each cell with an active direc field will copy the signal value from the neighbor "behind" its direc field pointer during each iteration. Therefore, the signal flow direction is the exact direction to which the direc field points.

When a signal G reaches the end of a signal path it will extend the path into the adjacent quiescent space

[4] For efficiency, the seed monitor should be placed in or near the initial loop structure, but its position is actually arbitrary.

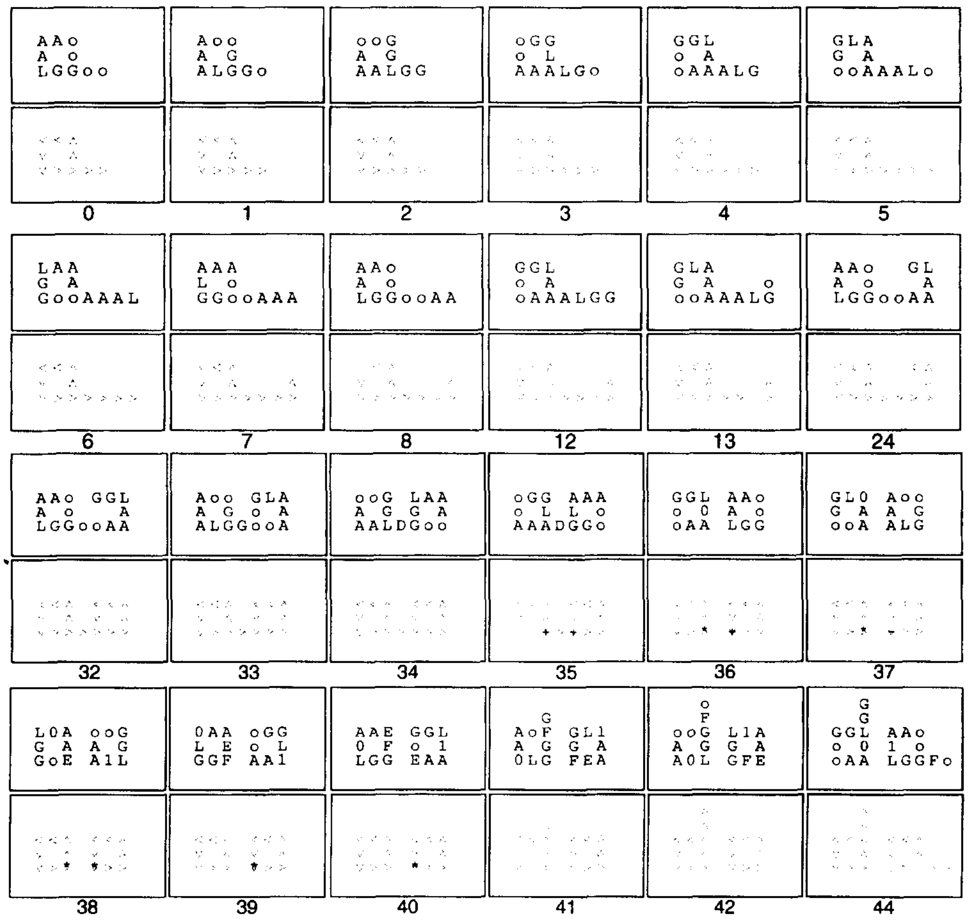

Fig. 8. The flow, extension and turning of signals. The code field (top) is displayed with its associated direc field (bottom, in light gray) for a number of time steps. The direc field defines the signal flow direction. These two fields actually overlay each other at each cell; they are displayed separately only for clarity. The special field in black is also shown overlaying the direc field in the bottom row.

and produce an 'o' there. Signal G will also set a new direc pointer there to point to the same extension direction, as shown at times 2 and 3 in Fig. 8. Turning of the extension direction is triggered by the signal *L*. When reaching the end of the signal path, it will set a new direc field pointer at its left neighbor cell pointing at the new extension direction. The code field does not get extended by the *L* signal; only the direc field is influenced, as is shown at times 6 and 7 in Fig. 8. When signal *G* reaches the end at time 12, it will cause an extension toward the new direction set by the *L* signal, making a new 'o' in the code field, as seen at time 13. Proceeding in this fashion, the replication arm continues turning counterclockwise as seen at time 24. Only replication control signals *G* and *L* change quiescent space into active cells. The other signals are ignored when they reach the end of the signal path. These signals, such as A, 0 or 1, carry the SAT bit sequence and do not play a role in replication control, as seen at times 7 and 8 in Fig. 8, where the A's disappear at the end of the extending path.

When the replicating arm finally completes a new loop, a detach signal *D* is formed. This in turn closes the new signal path, and separates the two loops. When signal *D* is generated it triggers the setting of two special flags in its neighbors. Each of the flags belongs to one of the loops. These two special flags control the enumeration of the SAT bit sequences and generation of new replicating arms.

Consider Fig. 8. At time 33 the arm is closing, which triggers the formation of detach signal D at time 34, and the setting of two special field flags at time 35. Signal D also modifies the direc pointer in the new loop at time 35. Signal D then disappears at time 36 together with its underlying direc pointer, completely separating the two loops. The two special flags in corners of the two loops trigger the formation of the signal sequence EF as seen at times 38–41. The EF signal sequence causes a new replication arm to appear in the following corner at times 41–44.

3.4. *Enumerating the SAT sequences*

As seen in Fig. 8, at time 35, two '+' flags form beside signal D. The left '+' flag immediately causes the code signal A atop it to be changed to binary bit 0, while itself changing to the '$*$' flag for generating an arm extrusion sequence. In the meantime, the signal L atop the right '+' flag changes it to a '$-$' flag during times 36 and 37. The '$-$' special flag then causes the signal A arriving at it to change to binary bit 1 at time 38, while itself changing to the '$*$' flag for generating an arm extrusion sequence. At time 39 the left '$*$' special flag disappears since it has generated the EF sequence. The right '$*$' flag will also disappear at time 41 when its EF sequence has been generated.

The function of the '+' special flag is to make the code signal A change to binary bit 0, and the function of the '$-$' special flag is to make the same signal A change to binary bit 1. Because of the timing difference in the parent and child loop, the special flag '+' in the child loop will be changed to flag '$-$' before it influences the following signal A. Therefore, the explored SAT sequences will be different in the parent and child loop. This is how the rule set enumerates the SAT bit sequences.

Explored binary bits 0 or 1 will stay unchanged and be copied into all descendent loops, while unexplored signal A's will be gradually reduced to either 0 or 1 during each replication cycle, depending on whether they are in the parent or child loop. Starting with only one initial loop carrying all unexplored A bits, the replication/enumeration process tries to produce all possible SAT bit sequences with the same number of bits as in the original loop. These bit sequences have the potential to represent all possible boolean assignments to the variables of a SAT predicate. Recall that Fig. 2(a) provides an example where an initial loop with three A's generates all eight possible SAT bit assignments in the end. When the enumeration process ends, all A's will have been explored if there are no collisions between loops and no selection process occurs. If two different loops try to grow into the same space and collide with each other, the rule set makes sure that the collision is resolved such that no information is lost. After a collision, a loop tries to replicate in a different direction; it continues to try until empty space appears.

3.5. *SAT clause checking, unsatisfying loops detection and deletion*

We now discuss how monitors remove loops carrying unsatisfying bit sequences for a SAT predicate. The clause field determines if there is an active monitor in a cell, and if so which SAT clause the monitor is checking for. The clause field is independent of all of the other fields: monitors work independently from the self-replication process and the enumeration process and have their own governing rules. Monitors are distributed throughout the space early in a simulation. Once distributed, the particular clause each of them checks for is fixed and does not change over time. The distribution of monitors is arbitrary as long as each loop overlaps all different monitors at least once. This requirement ensures that all clauses of the SAT predicate will be checked in all loops. Our model satisfies this condition.

The rules that generate additional monitors throughout the cellular space from the seed monitor are simple. If there is no active monitor in a cell, then the monitor value from either its north or west neighbor is referenced, if either of these exists. This value is increased by one modulo the total number of clauses, and the new calculated monitor value is stored in the cell. The entire cellular space is filled with monitors by this procedure due to the adjacency of opposite boundaries.

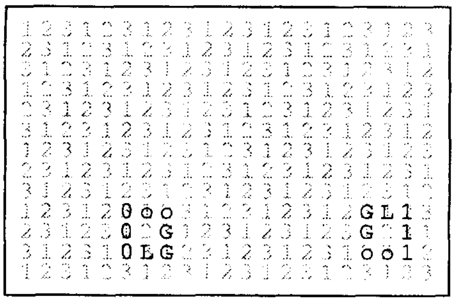

Fig. 9. Monitors (light gray) in the clause field. After monitor distribution is done, monitor values are distributed in such a way that all loops will overlap all different monitors at least once. There are two loops here, exactly as in Fig. 2(b). These two loops overlap all three different monitor values multiple times.

A typical situation after monitor distribution has been done is shown in Fig. 9, which is the same as Fig. 2(b). The clause field contains three different monitors numbered 1,2 and 3 for the three clauses of predicate $\mathcal{P}$ discussed in Section 1. The distribution of the clause values (the monitors) is even and each loop overlaps all three monitors at least once. During problem solving, each monitor keeps checking the bit sequence passing through it for unsatisfying ones. If an unsatisfying sequence (and the loop which carries it) is found by a monitor, that monitor will set the destruction flag '#' in the special field to destroy the loop. The use of monitors like this implies that to initialize the model to solve a specific SAT problem stated as predicate $\mathcal{P}$, not only must an initial loop be placed in the cellular space, but also a small number of rules must be specified stating the logical conditions associated with each monitor (i.e., this is how predicate $\mathcal{P}$ is specified).

A detailed example of how monitors work is shown in Fig. 10. For illustrative purposes, monitor distribution rules are disabled for this example and only one monitor is in the cellular automata space. This monitor is located at the upper right corner of the upper loop. This monitor checks for condition 2 of predicate $\mathcal{P}$. If the condition $\neg x_1 \wedge x_2$ is found to be true by the monitor, clause 2 of the predicate is unsatisfied by the loop and removal of the loop will begin.

The bottom loop in Fig. 10 is the parent loop in this example. At time 77 the detach signal D causes

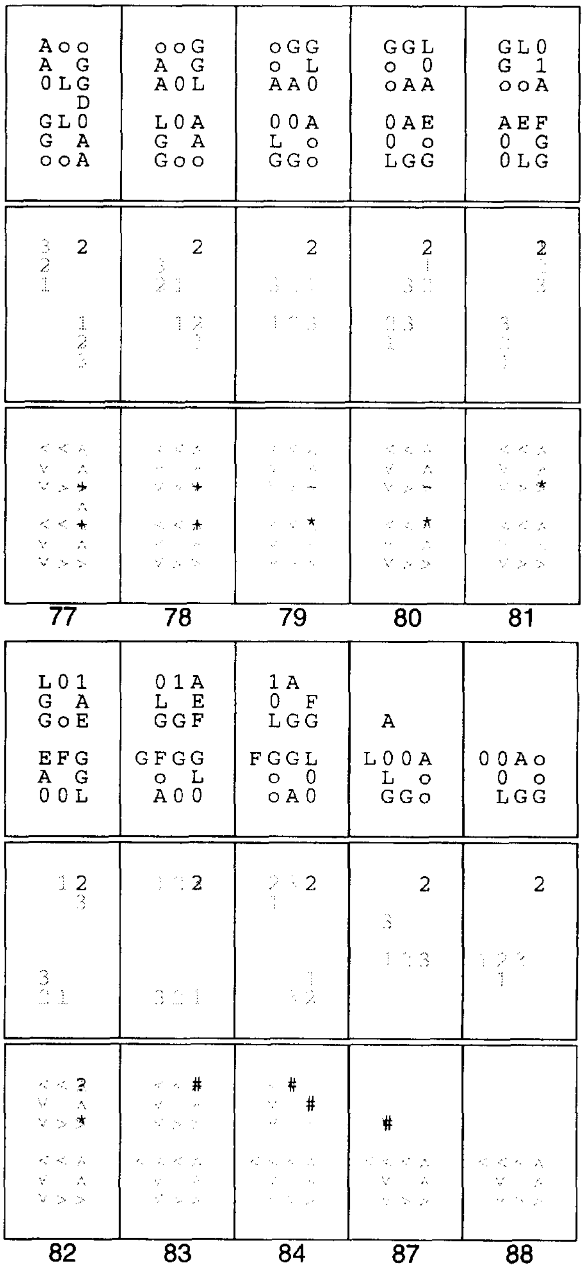

Fig. 10. Checking and deletion of unsatisfying loops by a monitor. The upper row shows the code field, the middle row shows the clause field (monitors) plus the pos field in gray, and the lower row shows the special field plus the direc field in gray. There is only one monitor in the upper right corner of the top loop which checks for clause 2 of the predicate $\mathcal{P}$ discussed earlier.

the two special flags '+' to be set in both loops, as discussed previously. The '+' flag in the child loop (top one) is later changed to the '−' flag by the L signal at time 79, which in turn causes the following signal A to be changed to binary bit 1 at time 81. Since the first variable (x_1) is verified to be '0' by the monitor at time 81, the monitor sets the special flag to '?' at time 82, which causes checking for the second variable (x_2). At time 82 the second bit (which has just been converted from signal A at time 81) is passing over the monitor embedded cell. since its value is '1', the second part of the condition is also verified by the monitor, proving that clause 2 of predicate $\mathcal{P}$ is not satisfied by the current loop. Therefore, at time 83 the special destruction flag '#' is set by the monitor, destroying the whole child loop by time 88. Note that the third bit in this loop was never explored.

By looking at the pos data field, the monitor knows that bit 0 on top of it at time 81 is assigned to variable x_1 of predicate $\mathcal{P}$ and bit 1 on top of it at time 82 is assigned to variable x_2. The pos field content is always translated with bits in the code field so that any monitor will know which values are assigned to which variables in the predicate.

In actual simulations, there are monitors throughout the entire cellular space, not just one monitor as in this example. As long as a loop is covered by all different monitors at least once, it is guaranteed that unsatisfying loops, once generated, will all be removed by monitors.

4. Analysis

In this section, a limited and elementary analysis is done of the space and time requirements of the model described above.

Proposition 1. The smallest n for which an $n \times n$ loop can solve an x-variable SAT problem is $n = \lceil (x+6)/3 \rceil$.

Proof. Each cell of the cellular space can accommodate one instruction in its code field. A self-replicating loop which has n cells on one side (an $n \times n$ loop) needs n instructions for its own replication control, plus two instructions for the arm extrusion sequence EF. The rest of its cells can be used to carry the SAT bit sequence. Therefore, an $n \times n$ loop can carry $4 \times (n-1) - (n+2) = 3n - 6$ SAT bits. For example, the 3×3 loop in Fig. 10 can carry up to $3 \times 3 - 6 = 3$ SAT bits. Thus the smallest $n \times n$ loop which can solve an x-variable SAT problem must satisfy the inequality $x \leq 3n - 6$, which gives the integer solution $n = \lceil (x+6)/3 \rceil$. □

Lemma 1. The number of iteration steps needed for the replication of an $n \times n$ loop is $21(n-1) + 2$.

Proof. This can be calculated based on the fact that an $n \times n$ loop has $4(n-1)$ cells and it takes four cycles of the signal sequence in the parent loop to replicate the child loop. It takes one more cycle to extrude the new arm and $n - 1$ more steps to move the starting signal G to the new arm position. The child loop is two steps behind the parent loop, so altogether it takes $5 \times 4 \times (n-1) + (n-1) + 2 = 21(n-1) + 2$ steps to complete a replica. □

Proposition 2. The maximum two-dimensional cellular space size needed to solve an x-variable SAT problem is $2x(n+1) + n$, or $(2x+1)\lceil x/3 \rceil + 6x + 2$, along each dimension.

Proof. During each generation, one solution bit is explored. For an x-variable SAT problem, x generations are needed to explore all possible bit sequences. Thus the maximum expansion along any one direction in the cellular space for x generations is $x(n+1)$, which is the width of each loop plus one boundary cell, multiplied by generations. Therefore, the maximum cellular space size needed to solve an x-variable SAT problem along one dimension is $2x(n+1) + n$ where the extra n is the original loop width. By Proposition 1, this can be expressed solely in terms of x. □

Proposition 3. The number of time steps needed to determine whether a SAT problem is satisfiable or not, assuming no collisions occur, is $21x(n-1) + 2x$, or $21x\lceil x/3 \rceil + 23x$.

Table 1
Mathematical property for some self-replicating loops

Loop size n $O(n)$	Total cells $4(n-1)$ $O(n)$	Maximum SAT bits $x=3n-6$ $O(n)$	Rep. steps (parent) $21(n-1)$ $O(n)$	Rep. steps (child) $21(n-1)+2$ $O(n)$	Maximum CA width $2x(n+1)+n$ $O(n^2)$	Projected iterations $21x(n-1)+2x$ $O(n^2)$
3	8	3	42	44	27	132
4	12	6	63	65	64	390
5	16	9	84	86	113	774
6	20	12	105	107	174	1284
7	24	15	126	128	247	1920
8	28	18	147	149	332	2682
9	32	21	168	170	429	3570
10	36	24	189	191	538	4584
11	40	27	210	212	659	5724
12	44	30	231	233	792	6990
13	48	33	252	254	937	8382
14	52	36	273	275	1094	9900
15	56	39	294	296	1263	11 544
16	60	42	315	317	1444	13 314
17	64	45	336	338	1637	15 210
18	68	48	357	359	1842	17 232
19	72	51	378	380	2059	19 380
20	76	54	399	401	2288	21 654
21	80	57	420	422	2529	24 054
22	84	60	441	443	2782	26 580

Proof. Assuming no collisions occur between loops, the number of time steps required is the number of generations to reach the solution times the number of replicating steps per generation. Since during each generation one SAT bit is explored, according to Lemma 1, for an x-variable SAT problem, this gives raise to $x \times (21(n-1)+2) = 21x(n-1)+2x$. □

Table 1 illustrates these results for small values of n. Note that one can relate the number of variables involved, x, to these results via column 3 of this table.

Proving the correctness of the rule set used in our model, as with correctness proofs for parallel algorithms in general, is a formidable task that will not be undertaken here. However, based on numerous simulations (see next section), we conjecture that the model of self-replicating loops described above can always determine whether a given predicate is satisfiable. To determine if a predicate is satisfiable or not, it is necessary to either find *one* satisfying bit sequence, or to see that there are no more self-replicating loops in the cellular space. Loops are generated when SAT bits are explored. A loop carrying fully explored bits will be removed when it is not satisfying. Peripheral loops always have space to expand into, assuming a space size not less than that of Proposition 2, so there is no stopping the exploration and removal process. When the expansion of a loop is temporarily stopped due to collisions, the loop will preserve its current status and wait for free space to appear and when it does, explore another SAT bit. Confirmation of this conjecture would require a formal proof that the rule set of this model will always perform as expected during every possible situation.

The last column of Table 1 lists the theoretically projected number of iterations required to solve a given SAT problem, assuming no collisions occur. In some cases, however, the actual steps needed to find a satisfying variable assignment can be more than that and depends on the characteristics of the SAT problem being solved. Note that the model here may fail to find *all* satisfying bit sequences for a predicate (rather than just one as required by the classic SAT problem).

5. Empirical study of selection and crowding

The last column of Table 1 lists the projected number of steps needed to determine if a given SAT predicate is satisfiable, assuming that there is no congestion of loops in the cellular space. The question that remains is whether this assumption is realistic, or whether the cellular space is usually crowded when solving SAT problems. If solutions are initially trapped in the central regions until space for their replication/exploration is freed up by the loss of loops carrying unsatisfying variable assignments, then the time required to find a solution can increase substantially.

If loops are never removed after appearing, within five generations loops in the central region of the population will start to have problems continuing replication. On the contrary, if only one loop survives during each replication (the other being removed by monitors, be it the parent or the child loop), self-replicating loops can finish exploring in exactly x generations for an x-variable SAT problem. This is the theoretical best case of this model for finding a satisfying variable assignment since it takes at least x generations to explore x bits (of course, detecting that a SAT problem is not satisfiable can occur much earlier if no loops remain).

To develop a quantitative understanding of how the selection process influences the efficiency of finding satisfying bit sequences, a series of simulations were run, each with a controlled selection behavior. To measure the progress toward finding satisfying bit sequences, the *generation index* of a loop is defined as the number of explored bits within that loop. Therefore, when solving an x-variable SAT problem, the initial loop, with no explored bits, is at generation index 0. The final, totally explored loop which no longer replicates, is at generation index x. The progress of the whole cellular automata space toward finding satisfying bit sequences can then be measured as the average value of the generation index over all loops in the space. For comparison purposes, the number of iterations is also measured in generations. To do this the current iteration number is divided by the number of steps in one full replication cycle of a loop.

Fig. 11 shows the progress of four different selection schemes, each trying to solve a different six-variable SAT problem using 4×4 loops. The different selection schemes arise as follows:

- Curve A represents the theoretical best case where during each generation only one loop is kept, so all bit sequences are explored in exactly six generations.
- Curve B represents the worst case where loops are never lost, so congestion prevents exploring all possible bit sequences (but it does not prevent correct solution of the SAT problem). An average generation index of 6 is never reached.
- Curve C represents a selection scheme where 50% of bit sequences are satisfying but selections occur only after the last bit is explored. Bit sequences are explored slowly in this case since loops in the congested region are trapped while waiting for peripheral loops to be erased to clear space for the trapped loops to replicate. This takes a much longer time to finish (12 generations).
- Curve D represents the same 50% satisfying ratio of all possible bit sequences but the removal of unsatisfying loops occurs at the fifth bit (i.e., one generation earlier than curve C). The time it takes to explore all bit sequences is faster than curve C, occurring at generation 8.

Fig. 11 suggests that the earlier selection occurs, the faster is the exploration of bit sequences. To examine this, two additional 50% curves with even earlier selection stages are plotted in Fig. 12 together with the two 50% curves of Fig. 11. In this new figure only the critical region is plotted to facilitate comparison. It can be seen that earlier selection at bit 4 does run faster than with selections at bit 5 or 6, but when selection occurs too early at bit 3 (curve D), congestion will still occur later, preventing exploration of all bit sequences. This is similar to the worst case in Fig. 11.

Additional simulations show that the satisfying ratio of loops also influences the speed of exploration. In general, the smaller the ratio of satisfying loops, the faster the exploration of bit sequences. Intuitively, this is because a smaller satisfying ratio allows more unsatisfying loops to be removed by the monitors and therefore decreases the congestion of the cellular space.

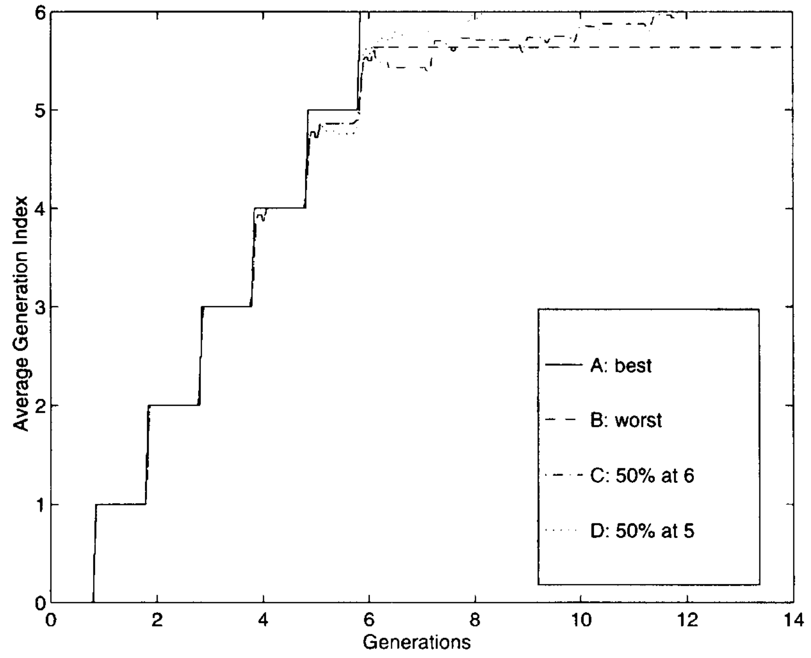

Fig. 11. Problem-solving progress with different selection schemes. Each selection scheme is controlled by a specially designed SAT predicate. The vertical axis shows the average generation index, which reveals the progress toward exploring *all* bit sequences. For six-variable SAT problems, the generation index is always between 0 and 6. The horizontal axis shows the iteration steps of the cellular automata measured in generations.

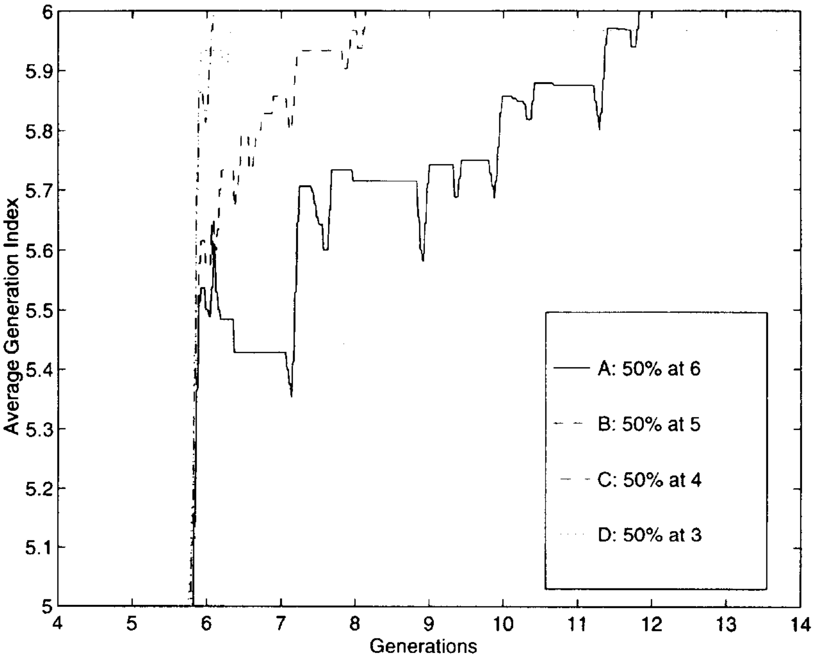

Fig. 12. Problem-solving progress with a 50% satisfying ratio occurring at different selection stages controlled by custom tailored SAT predicates. Same notation as in previous figure.

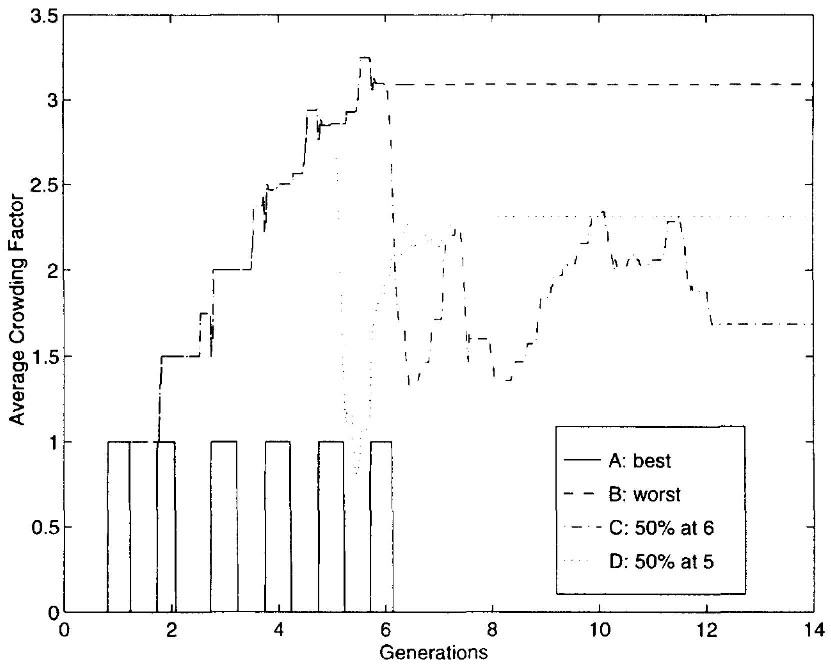

Fig. 13. Crowdedness curves for different selection schemes.

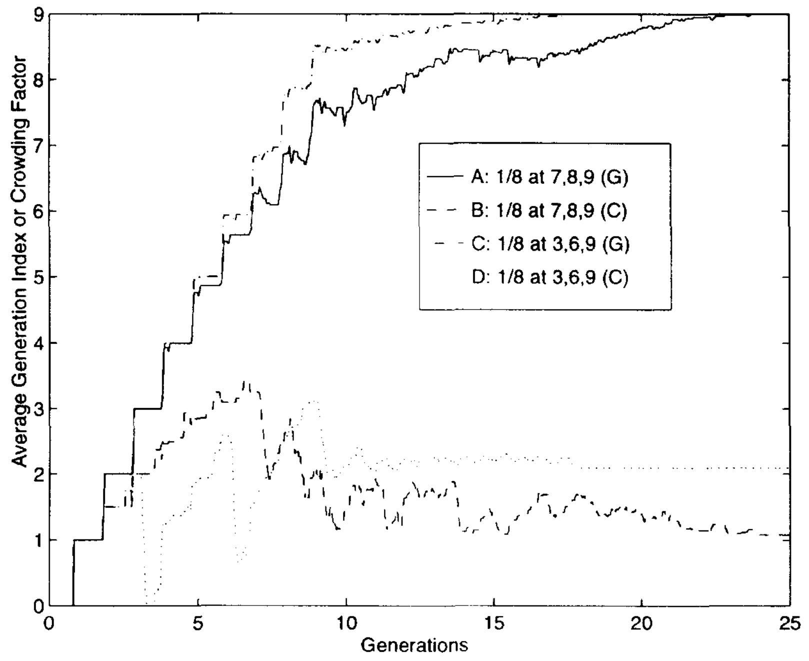

Fig. 14. Solving two 9-variable SAT problems with different selection schemes. The vertical axis shows the average generation index (for cruves A and C) or crowding factor (for curves B and D).

Table 2
Conjectural maximum satisfying ratio for different SAT problems that self-replicating loops can solve efficiently

Loop size n	Total cells $4(n-1)$	Maximum SAT bits $x = 3n - 6$	Solvable max. satisfying ratio $1/(2^{x/3}) \times 100\%$
3	8	3	50.000000%
4	12	6	25.000000%
5	16	9	12.000000%
6	20	12	6.250000%
7	24	15	3.125000%
8	28	18	1.562500%
9	32	21	0.781250%
10	36	24	0.390625%
11	40	27	0.195313%
12	44	30	0.097656%
13	48	33	0.048828%
14	52	36	0.024414%
15	56	39	0.012207%
16	60	42	0.006104%
17	64	45	0.003052%
18	68	48	0.001526%
19	72	51	0.000763%
20	76	54	0.000381%
21	80	57	0.000191%
22	84	60	0.000095%

This facilitates loop growth and exploration. Again, the congestion, or crowdedness of the cellular space, determines efficiency.

To better understand how the congestion of the cellular space correlates with the speed of exploration, we define a quantitative measure of crowdedness. The *crowding factor* for a loop is defined as the number of directions in which the loop cannot replicate. For a loop alone in the cellular space, its crowding factor is 0, i.e., it is not crowded at all. For a loop which is fully surrounded by other loops, its crowding factor becomes 4, the maximum value. the average crowding factor of all loops in the cellular space is taken as the crowdedness of the space as a whole.

The crowdedness curves for the same 6-bit SAT problems as in Fig. 11 are shown in Fig. 13. Curve A for the best case has a pulse-like pattern since only one loop is kept in each generation. Whenever a replication cycle is completed one of the two loops in the cellular automata space is removed, so the average crowding factor alternates between 0 and 1. Curve B for the worst case climbs rapidly to a crowding factor above 3 and then stays there. Curve C represents the case with a 50% satisfying ratio of loops and selection only at the last (6th) bit. Initially this curve shoots up rapidly just like curve B until a sudden drop in crowdedness occurs when the selection process starts at bit 6. This curve then fluctuates between the crowding factor of 1.25 and 2.3 while stalled inner loops get the chance to replicate. Curve D represents a case with 50% satisfying ratio and selection at the 5th bit. It behaves similar to curve C, but its drop in crowdedness occurs one generation earlier, and the drop is deeper.

Fig. 14 shows a longer simulation. In this example two 9-variable SAT problems are solved using 5×5 loops. The satisfying ratio is controlled at 1/8 while selections occur at bits 7, 8 and 9 for the first case, and bits 3, 6 and 9 for the second case. Early selections in the second case greatly helps in lowering cellular space congestion.

Based on many simulations like those described here we conjecture that for self-replicating loops to solve an x-variable SAT problem efficiently the ratio of satisfying bit sequences must be under $1/2^{x/3}$. The rationale for this is as follows. To efficiently explore all bit sequences of a SAT problem, the cellular automata space cannot become *too* crowded. The average crowding factor must be kept under 2 to avoid slowing of the search process. If the average crowding factor reaches 3 it can halt the whole exploration process. Since the average crowding factor climbs rapidly at first, it must be brought down at least once in three generations to keept it below 2. Because the satisfying ratio for a SAT problem is related to how many times selections are made by the monitors, for an x-variable SAT problem this roughly amounts to $1/2^{x/3}$. This can be seen as the heuristic upper bound on the satisfying ratio for any particular SAT problem to be solved efficiently by self-replicating loops. Table 2 shows the corresponding satisfying ratio bounds for the same loop size as in Table 1.

6. Discussion

In this paper, we have shown a new way in which self-replicating structures in a cellular space can be adopted to do problem-solving while replicating. Motivated by recent reports from molecular biology that recombinant DNA methods can be applied to solve SAT problems [1,7], we have demonstrated that replicating structures in a cellular space can do likewise. The method involved makes use of a similar enumeration-and-selection process as used in the biological system. Specific bounds have been given on the amount of space and time required to solve x-variable SAT problems as a function of x.

Recently, other efforts have been made to program self-replicating loops in cellular spaces to carry out tasks as they replicate. We briefly summarize these results here, contrasting them with our own. Perhaps the most straightforward approach to programming self-replicating loops to solve problems is simply to extend the sequence of signals circulating around the loop as was done here, adding additional signals representing a program that carries out some task. This application program is copied along with the replication program unchanged from generation to generation as the loop replicates, and is executed once by each loop in between replications. The viability of this approach was recently demonstrated by programming replicating loops to construct a pattern (the letters LSL) in the interior of each replicated loop [13]. Alternatively, arbitrarily long programs have been accommodated by adding "tapes" to loops [10]. Two vertically descending tapes of arbitrary size can be added to the bottom of each loop, one to store a signal sequence representing an application program, the other to store problem data. Using this approach, it has been shown that one can program a self-replicating loop to perform parentheses checking, a computation that corresponds to recognition by a non-regular language. It can be shown that tape-extended self-replicating loops are capable of executing any desired program [10]. Thus, in principle such extended self-replicating loops exhibit computational universality.

These past programmable self-replicating loops literally encode a set of instructions on the loop or an attached tape that directs solution of a problem. This application program is copied unchanged from parent to child, so each generation of loops is executing exactly the same program on exactly the same data. In contrast, with the model described in this paper, the initial problem solution is not copied exactly from parent to child but is modified from generation to generation. Each child loop gets a different partial problem solution. If a loop determines it has found a valid complete problem solution, it stops replicating and retains that solution as a circulating pattern in its loop. On the other hand, if a loop determines its partial solution is not useful, the loop is removed, erasing itself without descendents. This process can be viewed as a parallel state space search through the space of problem solutions. At the end of this process when replication has stopped, the cellular space contains one or more non-replicating loops, each with a circulating sequence of signals that encodes a valid problem solution (assuming such a solution exists). These and other recent results [8,10,13] suggest the possibility of evolving self-replicating structures in cellular spaces that not only replicate, but also develop a "metabolism" or carry out difficult computational tasks in a novel and efficient manner.

References

[1] L.M. Adleman, Molecular computation of solutions to combinatorial problems, Science 266 (1994) 1021–1024.

[2] J. Byl, Self-reproduction in small cellular automata, Physica D 34 (1989) 295–299.

[3] H.-H. Chou, J.A. Reggia, Spontaneous emergence of self-replicating structures in a cellular automata space, Physica D (1997), in press.

[4] E.F. Codd, Cellular Automata, Academic Press, New York, 1968.

[5] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.

[6] C.G. Langton, Self-reproduction in cellular automata, Physica D 10 (1984) 135–144.

[7] R.J. Lipton, DNA solution of hard computational problems, Science 268 (1995) 542–545.

[8] J. Lohn, J.A. Reggia, Automatic discovery of self replicating structures in cellular automata (1997), submitted.

[9] M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, Cambridge, MA, 1996.

[10] J.-Y. Perrier, M. Sipper, J. Zahnd, Toward a viable, self-reproducing universal computer, Physica D 97 (1996) 335–352.

[11] J.A. Reggia, S.L. Armentrout, H.-H. Chou, Y. Peng, Simple systems that exhibit self-directed replication, Science 259 (1993) 1282–1288.

[12] M. Sipper, E. Ruppin, Co-evolving architectures for cellular machines, Physica D 99 (1997) 428–441.

[13] G. Tempesti, A new self-reproducing cellular automaton capable of construction and computation, in: F. Moràn, A. Moreno, J.J. Merelo, P. Chacòn (Eds.), Lecture Notes in Computer Science, vol. 929, Springer, Berlin, 1995, pp. 555–563.

[14] J. von Neumann, The Theory of Self-Reproducing Automata, University of Illinois Press, Urbana, IL, 1966.