Symmetry in Gardens of Eden

Christiaan Hartman
Delft University of Technology
The Netherlands
c.hartman.cit@gmail.com

Marijn J. H. Heule*
The University of Texas at Austin
United States
marijn@cs.utexas.edu

Kees Kwekkeboom
Delft University of Technology
The Netherlands
keeskwekkeboom@gmail.com

Alain Noels
Delft University of Technology
The Netherlands
alainnoels@gmail.com

Submitted: Aug 9, 2012; Accepted: Jul 28, 2013; Published: Aug 9, 2013

Abstract

Conway's Game of Life has inspired enthusiasts to search for a wide range of patterns for this classic cellular automaton. One important challenge in this context is finding the smallest Garden of Eden (GoE), a state without a predecessor. We take up this challenge by applying two techniques. First, we focus on GoEs that contain a symmetry. This significantly reduces the size of the search space for interesting sizes of the grid. Second, we implement the search using incremental satisfiability solving to check thousands of states per second. By combining these techniques, we broke several records regarding GoEs: the fewest defined cells, the smallest bounding box, and the lowest living density. Furthermore, we established a new lower bound for the smallest GoE.

1 Introduction

Conway's Game of Life (Life) is a highly popular zero-player game that generates fascinating patterns using a simple cellular automaton. Cells are either "alive" or "dead" and they are updated based on the state of their neighbours. Life enthusiasts have been searching for patterns such as still lifes, oscillators, and space ships[1]. Many of these patterns can be detected with a random initial state. A state that can only exist as an initial state is called a *Garden of Eden* (GoE).

*Supported by DARPA contract number N66001-10-2-4087.
[1]see http://www.conwaylife.com/wiki/

The quest for the smallest GoE started shortly after Life was designed in the early '70s. The first published GoE was found by a team from MIT in 1971 and consists of 226 alive cells and fitted in a $33{\times}9$ bounding box [25]. The GoE with the smallest width fits in a bounding box of $117{\times}6$ [16]. Flammenkamp discovered many small GoEs; His latest improvement (2004) is a GoE with 72 alive cells fitting within a $11{\times}12$ bounding box. Beluchenko made the latest improvements (2009) to this area by showing that there exists a GoE with 45 alive cells within a $11{\times}11$ bounding box. In this paper, we further improve upon these records by applying two satisfiability (SAT) solving techniques.

First, we exploit symmetries to significantly reduce the search space. This includes breaking symmetries between solutions as well as enforcing symmetries within solutions, or *internal symmetries* [17]. The latter technique is more useful, but no longer guarantees that all solutions will be found: the search is restricted to solutions that contain the forced symmetry. Several record-breaking patterns regarding GoEs reveal internal symmetries [11]. This also holds for other patterns in Life such as maximum density still life [5].

Second, we use the latest SAT solving techniques to systematically explore the search space. In most cases, even after reduction, over a billion states need to be explored. This can only be accomplished in a reasonable amount of time if the check for a single state requires less than a fraction of a second. We realize this by translating the check into SAT, equivalent to [2], and solve the translated problem using a SAT solver.

The existence of a GoE in an $n{\times}m$ grid is most compactly represented as a two level Quantified Boolean Formula (2QBF). Given a SAT translation F of the Life transition relationship with a certain bounding box, one simply adds quantifiers $\forall x' \, \exists x \, F$, with x' referring to the parent (output) state variables and x to the predecessor (input) state variables. However, existing QBF solvers are not efficient on these problems.

Alternatively, one could translate the entire search problem into a single large SAT instance by eliminating the universal quantified variables in the 2QBF representation. A recent trend in SAT solving suggests that solving millions of simple SAT problems is more efficient. Two techniques in this context are currently used in the field of Formal Verification [6, 7]. We follow this trend by translating only the check whether there exists a predecessor state into SAT and include it in a search procedure that enumerates the entire search space after symmetry breaking and forced internal symmetries.

We use *incremental* SAT solving [10] to implement this search. With this technique, each state of the grid is assigned as an *assumption* (decision before search). The advantages of this approach are that 1) the SAT solver has to be initialized only once, avoiding the overhead of repetitive parsing; 2) heuristics can be reused to solve similar states; 3) learned clauses can be kept; and 4) in the case that a GoE is found, then the *orphan* (minimal sized GoE) can be extracted cheaply by checking which assumptions were required. Incremental SAT solving facilitates checking thousands of states per second on a single core machine.

Our compact incremental SAT based 2QBF solver shows strong performance on GoE benchmarks and could be effective for other small hard 2QBF problems. Using our solver, we were able to establish several new records regarding GoE's: the smallest bounding box of 10×10, the fewest defined cells of 92, and the lowest alive density of 0.320. Additionally, we show that there is no GoE within a bounding box of 6×6.

The rest of this paper is organized as follows. As preliminaries, Conway's Game of Life is explained, focussing on orphans and the basics of SAT and QBF solving. Then, we discuss how the predecessor problem can be translated into 2QBF and SAT. Next, we present how symmetries can be exploited to reduce the search space. In the experimental results section, we show that these techniques are sufficient to break several records regarding GoEs. Finally, we draw some conclusions.

2 Preliminaries

2.1 Conway's Game of Life

The *Game of Life* is a cellular automaton invented by John Conway [12]. Life consists of a two-dimensional grid of cells that are either "alive" or "dead" called the *cellular space.* Each cell in Life is a finite state automaton with update rules that are based only on local information: i.e., the states of its neighbours and the current state of the cell. Neighbours are cells that are either horizontally, vertically or diagonally adjacent; the *neighbourhood* of a cell therefore consists of eight cells.

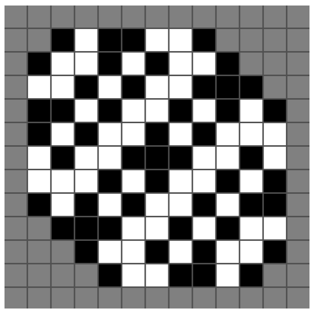
Analogous to biological evolution, Life implements well-known mechanisms like survival-of-the-fittest. This means that the game evolves in time, where the state of all cells in each discrete time step is a called a *generation.* Each cell is alive or dead in the next generation depending on four simple rules:

1. *Under-population*: any cell with less than two alive neighbours will be dead in the next generation.
2. *Overcrowding*: any cell with more than three alive neighbours will be dead in the next generation.
3. *Status-quo*: any alive cell with exactly two alive neighbours will survive in the next generation; and any dead cell with exactly two alive neighbours will stay dead.
4. *Birth*: any cell with exactly three alive neighbours will be alive in the next generation.

This update process is done synchronously and can be repeated infinitely.

2.2 Orphans

Life begins by setting the cells in a state called the *initial generation.* The next state of a cell only depends on its current state and the states of the surrounding neighbours. This implies that every generation has a successor. However, the contrary is not true: not every generation has a predecessor. That is, Life is a non-surjective cellular automaton.



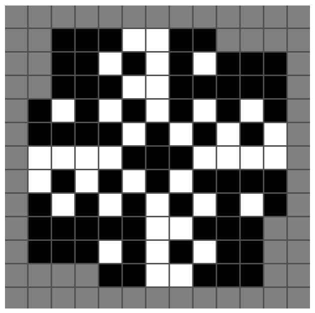


Figure 1: On the left is the smallest orphan containing 51 alive (black) cells, 56 dead (white) cells and fits in an 11×11 bounding box. Grey cells are undefined. On the right is the quarter-rotational symmetric orphan known as the Flower of Eden.

A generation that cannot evolve from a previous generation by following the rules of Life is called a *Garden of Eden*. An *orphan* is a minimal subset of cells of a generation that has no predecessor. In other words, each generation that contains an orphan is a GoE.

Several orphans have been discovered [11]. The smallest known orphan was found by Beluchenko and is shown in Figure 1. There exists no orphan fitting in a 6×5 bounding box[2].

Checking whether a given generation is a GoE is difficult because Life is not reversible; instances of Life can have multiple predecessors or none. No automaton is known which can perform the reverse step. The reversibility problem scales exponential with the size of the input grid: a given generation of an $n \times m$ grid has $2^{(n+2)(m+2)}$ possible previous generations. Of course, only a fraction of these generations are predecessors. This explosion of possible previous generations makes it difficult to verify an orphan.

Several techniques have been developed to find GoEs. A non-deterministic finite automaton can accept states that have a predecessor [16]. Reversing this automaton gives an automaton that only accepts states without predecessor. Searching for states that are accepted by the automaton gives orphans. Beluchenko [3] uses predecessor counts to successively approximate GoEs. In each step, one cell is added to the pattern which grows spiral-wise. The state of all prior cells remains the same. Whether the new cell in each step is alive or dead depends on which state in combination with the existing pattern has the fewest predecessors. The process is repeated until adding a cell reduces the number of predecessors to zero.

2.3 Satisfiability solving

The *Boolean satisfiability problem* (SAT) decides if a given CNF formula has a solution. For a Boolean variable x, there are two *literals*: the positive literal x and the negative literal $\bar{x}$. A *clause* is a finite disjunction of literals and a CNF formula is a finite conjunction of clauses. A truth assignment for a CNF formula F is a function τ that maps literals in

[2] http://www.conwaylife.com/wiki/Garden_of_Eden

F to the set $\{0,1\}$. If $\tau(x) = v$, then $\tau(\bar{x}) = 1 - v$. A clause C is satisfied by τ if $\tau(l) = 1$ for some literal $l \in C$. A clause C is falsified by τ if $\tau(l) = 0$ for every literal $l \in C$. An assignment τ satisfies F if it satisfies every clause in F. A CNF formula F is *satisfiable* if there exists an assignment that satisfies F, otherwise F is called *unsatisfiable*.

2.4 Quantified Boolean Formula solving

The *Quantified Boolean Formula problem* (QBF) is a generalization of SAT. An existential ($\exists$) or a universal ($\forall$) quantifier can be applied to each Boolean variable. A QBF formula in prenex normal form $Q_1 x_1 Q_2 x_2 \dots Q_n x_n F$ consists of two parts: a sequence of quantifiers $Q_i \in \{\exists, \forall\}$ and CNF formula F. Solving a QBF formula in prenex normal form can be viewed as a two player game with a universal and an existential player. The universal player tries to falsify F, while the existential player tries to satisfy F. The variables are assigned from left (or outer scope) to right (or inner scope). If a variable is universally quantified, then the universal player assigns it, if it is existentially quantified, the existential player assigns it. A QBF formula is satisfiable if the existential player can always win. Otherwise the formula is unsatisfiable.

A special class of QBF is 2QBF. In 2QBF, formulas have only two levels of quantifiers. These formulas are of the form $\forall x_1, x_2, \dots, x_r \, \exists y_1, y_2, \dots, y_s \, F$. *Expansion* [1] of universal variables works as follows. Let G denote a copy of F in which all existential variables y_i are replaced by z_i and let $F(x = 0)$ denote that x is assigned to 0. Now expansion of x_1 equals

$$\forall x_1, x_2, \dots, x_r \, \exists y_1, \dots, y_s \, F \Rightarrow \forall x_2, \dots, x_r \, \exists y_1, \dots, y_s, z_1, \dots, z_s \, F(x_1 = 0) \wedge G(x_1 = 1)$$

Expansion of all universal variables in a 2QBF formula yields a corresponding SAT problem.

2.5 Incremental SAT solving

Conflict driven clause learning (CDCL) [21] is the most successful SAT solver architecture. The CDCL algorithm repeats the following: (1) If not all variables are assigned, then a free variable is selected (decide) and assigned a truth value followed by simplifying the formula; otherwise a solution is returned; (2) In case the current assignment falsifies a clause, then analyze the conflict —producing a learned clause which is added to the formula— to avoid this conflict in the future; otherwise go to step 1; (3) If the learned clause has no literals return unsatisfiable; (4) Backtrack to the point where the newly learned clause becomes a unit clause (one unassigned literal and all other literals are falsified). Simplify the formula and go to step 2.

Heuristics are crucial to achieve fast performance in CDCL SAT solvers. The two main heuristics of these solvers are variable selection and value selection heuristics. In short, variable selection heuristics prefer variables that were frequently involved in recent conflicts. These heuristics are known as Variable State Independent Decaying Sum

(VSIDS) [22]. The most commonly used value selection heuristics are called phase-saving [23]. These heuristics store for each variable the truth value to which it was forced most recently during simplification. When a variable is selected by VSIDS, it is assigned to the stored truth value.

Incremental SAT solving [10], a technique supported by most CDCL solvers, can be used to solve *similar* SAT instances. It allows addition and removal of clauses to the formula. The addition of unit clauses is realized by *assumptions*: decisions at level 0 (before the search). The solver runs and returns the solution, after which the assumptions are removed. Learned clauses are independent of the assumptions under which they have been recorded. These learned clauses can be reused to help solving the other instances. There is also another big advantage: because new instances are provided as assumptions to the solver instead of a CNF formula, variable and value selection heuristics can be cached. Again, this helps solving *similar* SAT instances.

3 Encoding

The existence of a GoE in a certain bounding box can naturally be encoded into 2QBF. A similar SAT encoding can be used to check if a certain generation has a predecessor. Highly specialized solvers exist for both representations.

3.1 General encoding

For a finite pattern in a grid of size $(1,1)-(n,m)$, the previous generations fit in a grid of size $(0,0)-(n+1,m+1)$ as only information from the neighbourhood is needed by the rules of Life.

A Boolean variable $x_{i,j}$ naturally encodes the state of a single cell on position (i,j). A cell on (i,j) is alive if and only if $x_{i,j}=1$; and dead otherwise. Variable $x'_{i,j}$ denotes the truth value of the cell on (i,j) in the next generation. The neighbourhood $N_{i,j}$ of a cell at (i,j) is the set of variables corresponding to the adjacent cells of $x_{i,j}$. Thus $N_{i,j}=\{x_{i+1,j+1},\ x_{i+1,j},\ x_{i+1,j-1},\ x_{i,j+1},\ x_{i,j-1},\ x_{i-1,j+1},\ x_{i-1,j},\ x_{i-1,j-1}\}$. Let $N^k_{i,j}$ denote the set containing all subsets of $N_{i,j}$ of size k. Cells in $(1,1)-(n,m)$ are called inner cells. Notice that $N^8_{i,j}=N_{i,j}$ for inner cells. One could add clauses for corner and border cells, but preprocessing techniques used in QBF and SAT solvers such as variable elimination [8] and blocked clause elimination [19] will identify these clauses as redundant (and remove them). The rules of Life can be expressed as follows:

Under-population

$$\Big(\bigvee_{S\in N^7_{i,j}}\Big(\bigwedge_{x_{k,l}\in S}\bar{x}_{k,l}\Big)\Big)\rightarrow \bar{x}'_{i,j} \tag{1}$$

Over-crowding

$$\Big(\bigvee_{S\in N^4_{i,j}}\Big(\bigwedge_{x_{k,l}\in S}x_{k,l}\Big)\Big)\rightarrow \bar{x}'_{i,j} \tag{2}$$

Status-quo

$$\Big(\bigvee_{S\in N^2_{i,j}}\Big(\bigwedge_{x_{k,l}\in S} x_{k,l} \wedge \bigwedge_{x_{k,l}\in N_{i,j}\setminus S} \bar{x}_{k,l}\Big)\Big) \rightarrow (x_{i,j} \leftrightarrow x'_{i,j}) \tag{3}$$

Birth

$$\Big(\bigvee_{S\in N^3_{i,j}}\Big(\bigwedge_{x_{k,l}\in S} x_{k,l} \wedge \bigwedge_{x_{k,l}\in N_{i,j}\setminus S} \bar{x}_{k,l}\Big)\Big) \rightarrow x'_{i,j} \tag{4}$$

These implications are then translated into a conjunctive normal form (CNF) formula, the input format for QBF / SAT solvers. The resulting encoding requires per inner cell $8 = \binom{8}{7}$ clauses of length 8 for under-population, $70 = \binom{8}{4}$ clauses of length 5 for overcrowding, $56 = 2\binom{8}{2}$ clauses of length 10 for status quo, and $56 = \binom{8}{3}$ clauses of length 9 for birth. In total 190 clauses per (inner) cell. The status quo clauses can be reduced to either length 8 or 9 by resolving them with the other clauses.

3.2 QBF and SAT encoding examples

To create a 2QBF encoding, quantifiers need to be added to the encoding above. The universal quantifier is applied to all $x'_{i',j'}$ variables, while the existential quantifier is applied to all $x_{i,j}$ variables. If the formula is unsatisfiable it means that there is an assignment to the universal variables that eliminates all solutions. Such an assignment corresponds to a GoE.

Example 1. Consider the problem of checking whether there exists an orphan within a bounding box of 3×3. The corresponding grid is $(0,0) - (4,4)$ and the CNF formula has 25 $x_{i,j}$ and 9 $x'_{i',j'}$ Boolean variables, and $9 \cdot 190 = 1710$ clauses. This encoding can be transformed into a 2QBF problem by adding the quantifier prefix: $\forall x'_{i',j'}\, \exists x_{i,j}\, F$ for $i', j' \in \{1,2,3\}$ and $i, j \in \{0,1,2,3,4\}$. This formula is satisfiable, so there is no GoE within a 3×3 grid.

A similar SAT encoding can be used to check if a given generation has predecessors: simply assign the variables $x'_{i',j'}$ of the generation to the corresponding truth values. A solution in this context means that the generation has a predecessor. If the formula is unsatisfiable, then the generation is a GoE. Additionally, by enumerating all generations of a certain grid, SAT solvers can also be used to find an orphan in the dimensions for that grid.

Example 2. Again, consider the problem of checking whether there exists an orphan within a bounding box of 3×3. The corresponding grid is $(0,0) - (4,4)$ with 25 $x_{i,j}$ and 9 $x'_{i',j'}$ Boolean variables. This problem can be partitioned into 2^9 states that cover all possible truth values for the variables $x'_{i',j'}$ with $i', j' \in \{1,2,3\}$. For each state, we add unit clauses $(x'_{i',j'})$ or $(\bar{x}'_{i',j'})$ to the encoding depending on the truth value in the state.

The same SAT encoding has been proposed in [2] to check how many random generations are GoEs. Experimental results based on grids of size 15×15 to 25×25 showed that there are many GoEs where $48-88\%$ of the cells are alive. Instances generated from grids of size 15×15 are rarely unsatisfiable. Alternatively, we perform systematical search for GoEs and focus on smaller grids. As we will show in Section 5, the combination of systematic search and incremental SAT boosts performance which makes it possible to explore many more generations per second.

3.3 Implementation and parallelization

Given both encodings above, there are three possible methods to find GoEs. First, one can solve the 2QBF representation with a QBF solver. However, QBF solvers have trouble with performance (see Section 5.2). Second, one can transform the 2QBF instance into a single SAT instance by expansion of the universal variables. Although the 2QBF representation is small, expansion for interesting grid sizes increases the formula size by a factor million, making it too large for contemporary SAT solvers. Even for small grids this approach yielded poor results. Third, perform enumerative induction over the entire search space of 2^{nm} possible states and prove or disprove GoE existence: assign the variables $x'_{i',j'}$ with $i' \in \{1, \dots n\}, j' \in \{1, \dots m\}$ to all possible truth values and check each distinct generation for an orphan.

One way to enumerate the entire search space is to generate 2^{nm} copies of the encoding and add unit clauses $(x'_{i,j})$ or $(\bar{x}'_{i,j})$ depending on the truth value of the corresponding cells for each generation, as described in Example 2. This is very inefficient: the SAT solver has to parse the same encoding for every instance and the entire search problem is enumerated, resulting in similar instances. With incremental SAT solving [10], each generation can be provided as a set of *assumptions* to the solver and the encoding needs to be parsed only once. Reuse of conflict clauses helps to solve *similar* SAT instances, as we will show in Section 5.1.

Enumeration of the search problem has one inherent advantage: this process is ideal for exploiting multiple cores in contrast to solving one large SAT instance. In CDCL, parallelization of one large SAT instance is non-trivial, as sharing learned clauses between processors easily increases the communication between processors [15]. But with incremental SAT solving, partitioning of the search space is trivial. The generations — converted to assumptions— can be spread over multiple cores, realizing a linear speed-up of the search process in practice.

4 Symmetries

4.1 Symmetry Breaking

Several symmetries exist when searching for an orphan O within a square grid. For instance, if O has no predecessor, then the generation O^*, which is a rotation of O by 180 degrees, is also not an orphan. By identifying and breaking solution symmetries, the search space can be reduced, while it still guaranteeing that all solutions can be found.

The predecessor problem for a given generation in Life has at most seven distinct symmetrical instances. Those instances can be obtained by flipping the problem horizontally, vertically, diagonally, or combining these reflections.

Example 3. Recall Example 1. Each state can be represented by a string of 9 Boolean values that correspond to the truth values for the variables $x'_{i,j}$ with $i, j \in \{1, 2, 3\}$. Consider the state `100001000` which assigns the variables $x'_{1,1}$ and $x'_{2,3}$ to true and the others to false. The states `100100000`, `001100000`, `001000010`, `010000100`, `000100100`, `010000001`, `000100001` are symmetric to `100001000`.

During the experiments, we skip all generations that have a symmetric equivalent with a large string representation. Thus, we only consider the generation with the largest string representation. We refer to this as *symmetry breaking* (SB).

4.2 Internal Symmetry

The smallest known orphan and the Flower of Eden, both shown in Figure 1, reveal a clear symmetry *within* the solution [17]: the first a reflection in both diagonals (left), the second a rotation by 90 degrees (right). Moreover, most of the currently known GoEs reveal internal symmetries [11]. Other patterns, like maximum density still lives [5] and oscillators in Life often expose internal symmetries as well. These symmetric patterns seem to naturally evolve in Life as a consequence of the uniform rules. We can use this observation to pragmatically reduce the search space by enforcing internal symmetries.

More formally, a generation G contains an internal symmetry or symmetry *within* the solution if there exist a (non-trivial) mapping σ that maps G onto itself: $\sigma(G) = G$. Enforcing internal symmetries reduces the search at the cost of being incomplete and makes it possible to systematically explore interesting parts of the search space for reasonably large grid sizes. Exploiting internal symmetries can reduce the search space by orders of magnitude when compared to symmetry breaking.

We propose three levels of internal symmetries that add increasing restrictions to the search space. The symmetries are presented for $n \times n$ grids, because those will be the focus of our experiments, but are applicable to $n \times m$ grids with $n \neq m$ as well. The first level (approximately) halves the number of distinctly defined cells, reducing the search space from 2^{n^2} to about $2^{n^2/2}$. With this first level of internal symmetry, SB can further reduce the search space by a factor of four.

- Horizontal reflection: σ_H. Enforce that $x'_{i,j} \leftrightarrow x'_{n+1-i,j}$ for $i, j \in \{1, \ldots, n\}$ reducing the number of distinctly defined cells to $n\lceil n/2 \rceil$.
- Diagonal reflection: σ_D. Enforce that $x'_{i,j} \leftrightarrow x'_{j,i}$ for $i, j \in \{1, \ldots, n\}$ reducing the number of distinctly defined cells to $n(n + 1)/2$.
- Rotation by 180 degrees: σ_{180}. Enforce that $x'_{i,j} \leftrightarrow x'_{n+1-i,n+1-j}$ for $i, j \in \{1, \ldots, n\}$ reducing the number of distinctly defined cells to $\lceil n^2/2 \rceil$.

If enforcing one internal symmetry does not sufficiently reduce the search space, we combine two internal symmetries to reduce the number of distinct cells by a factor of at most four. SB can further reduce the space by a factor of two.

- Rotation by 90 degrees: σ_{90}. Enforce that $x'_{i,j} \leftrightarrow x'_{j,n+1-i}$ for $i,j \in \{1,\dots,n\}$ reducing the number of distinctly defined cells to $\lceil n/2 \rceil \lfloor n/2 \rfloor + (n \bmod 2)$.
- Combining σ_H and σ_{180}. Enforce that $x'_{i,j} \leftrightarrow x'_{n+1-i,j} \leftrightarrow x'_{n+1-i,n+1-j}$ for $i,j \in \{1,\dots,n\}$ reducing the number of distinctly defined cells to $\lceil n/2 \rceil \lceil n/2 \rceil$.
- Combining σ_D and σ_{180}. Enforce that $x'_{i,j} \leftrightarrow x'_{j,i} \leftrightarrow x'_{n+1-i,n+1-j}$ for $i,j \in \{1,\dots,n\}$ reducing the number of distinctly defined cells to $\lfloor n/2 \rfloor \lceil n/2 \rceil + \lceil n/2 \rceil$.

For large sizes of the grid, we can enforce an internal symmetry to reduce the number of distinctly defined cells by a factor of at most eight. In this case, SB cannot be applied.

- Combining σ_D and σ_{90}. Enforce that $x'_{i,j} \leftrightarrow x'_{j,n+1-i} \leftrightarrow x'_{j,i}$ for $i,j \in \{1,\dots,n\}$ reducing the number of distinctly defined cells to $2\lfloor (n+1)/2 \rfloor \lceil (n+2)/2 \rceil$.

Enforcing that $a \leftrightarrow b$ in SAT can be done by adding clauses $(a \vee \neg b) \wedge (\neg a \vee b)$ or by replacing all occurrences of literal b by a and literal $\neg b$ by $\neg a$. In the 2QBF representation, only the latter is valid when the equivalent variables are universal. Symmetry breaking and enforcing rotational symmetry was applied by Barbara Smith [24] to find maximum density still-lifes in Life using the ILOG solver.

5 Experimental results

Three experiments have been performed. The first one shows the effectiveness of the incremental SAT approach. The second experiment evaluates the performance of (2)QBF solvers. The third describes our systematic search for GoEs and the results.

5.1 Efficiency of a predecessor check in SAT

We used the iCNF patch[3] for MiniSAT 2.2 [9] for our experiments. This patch allows us to easily use the assumption techniques in MiniSAT by adding a few lines to the input formula. To evaluate the SAT based predecessor check we compared our implementation with other software for GoE checks. Achim Flammenkamp was willing to provide us with the software he used to perform predecessor checks. This software is written in C and counts predecessors for a given generation. It must be noted that this program has never been optimized for speed. In order to do a fair comparison, a small adaptation has been made to the software, so that execution terminates when detecting one predecessor. The parsing load is comparable: the SAT solver accepts the iCNF format, and the C program accepts a pattern description of a single character per cell. The SAT solver has to parse the problem encoding once.

Compared to "*simple SAT*", i.e., not using assumptions, incremental SAT has several features that may help to reduce the computational costs. Simple SAT, as in [2], parses

[3]The patch is available on http://users.ics.tkk.fi/swiering/icnf/.

the problem for each generation, while incremental SAT parses only once. Simple SAT *initializes* (variable selection and value selection) heuristics before solving a generation, while incremental SAT *reuses* heuristics values of the last solved generation. The simple SAT approach starts without learned clauses, while incremental SAT can keep learned clauses obtained from solving earlier generations in its database.

The performance of the different approaches has been evaluated using $10,000$ generations of Life — without applying internal symmetry or symmetry breaking. We generated 100 random generations for a grid of size 10×10. Each cell of Life had a probability of 50% of being alive. Each random generation was used to generate the next 100 subsequent generations. The latter was performed to let the solver take advantage of the incremental SAT approach. The test has been executed on an Intel Xeon X5570 (@2,93 GHz) processor with 32 GB RAM. The results are shown in Table 1.

Table 1: Comparing runtimes for the C program and SAT approaches.

Checker	runtime (s)	generations/s
C program	$14,826.00$	0.67
simple SAT, i.e., solve a new CNF formula for each generation	224.00	44.64
incremental SAT + reset heuristics + remove learned clauses	211.35	47.31
incremental SAT + reuse heuristics + remove learned clauses	26.65	375.23
incremental SAT + reset heuristics + keep learned clauses	3.33	$3,003.00$
incremental SAT + reuse heuristics + keep learned clauses	2.18	$4,589.00$

The incremental SAT approach with all features enabled (i.e., reusing heuristics and keeping some learned clauses) is able to check thousands of generations per second, making it almost four orders of magnitude faster when compared to the C program. Passing on the heuristic values from each generation to the next is the most effective feature. Removing all learned clauses, while keeping the heuristics hardly influences the performance. Resetting both heuristics and the learned clause database significantly hurts performance. In fact, this version is hardly faster than the simple SAT approach. The cost of initializing the SAT solver (for each generation in the simple SAT approach) appears marginal. Due to this huge performance improvement, many more generations can be explored at the same time, making it possible to explore much larger grid sizes.

5.2 Gray code based 2QBF Solver

Finding a GoE is most naturally represented in 2QBF, so we experimented with QBF solvers to study to which extent QBF techniques can be used to improve performance. We experimented with three state-of-the-art QBF solvers: depQBF version 0.1 [20], QuBE version 7.2 [13], and areqs2 version 0.1 [18]. Additionally, the QBF preprocessor bloqqer [4] was applied because of its effectiveness on many other QBF problems.

For comparison, we transformed our approach into a 2QBF solver. Recall that reusing the heuristics was most important contribution of our incremental approach (Section 5.1).

Given that insight, our QBF solver uses Gray codes [14] to ensure that two consecutive generations solved by the incremental SAT solver differ on exactly one assumption.

Example 4. Consider the problem of finding a GoE in a 2×2 grid. As discussed in Example 2, each generation can be expressed as a Boolean string of length $2 \times 2 = 4$. Using Gray codes the 16 different generations can be evaluated in the following order: `0000`, `0001`, `0011`, `0010`, `0110`, `0111`, `0101`, `0100`, `1100`, `1101`, `1111`, `1110`, `1010`, `1011`, `1001`, `1000`.

The source code of our QBF solver based on incremental SAT, called GrayQBF, is shown in Appendix A. It is a conversion script that translates a QBF formula in the standard DIMACS encoding into an iCNF file that can be used with any incremental SAT solver. Consecutive generations only differ in the assignment of one x' variable using Gray codes. Our solver consists of only 40 lines of C code of which half are used for parsing.

Table 2 compares the QBF solvers on relatively small grid sizes of 3×3, 4×4 and 5×5. The runtimes rapidly increase for larger grids. All the state-of-the-art QBF solvers struggle to solve the 5×5 grid. Our compact GrayQBF solver outperforms the others on these GoE benchmarks. Applying bloqqer as preprocessor is clearly beneficial for the other solvers though not as useful for GrayQBF.

5.3 Search for Gardens of Eden

Currently, the smallest GoE is bounded by an 11×11 grid with a lower bound of 6×5. Problem sizes between those bounds are of interest to find GoEs and improve existing records. Records as tracked by [11] are currently:

1. The smallest bounding box, 11×11;
2. Fewest number of defined cells, 107;
3. Fewest alive cells, 45;
4. Lowest density of alive cells, 0.388;
5. The lower bound for GoEs, 6×5.

Table 2: Comparing the runtimes in seconds for QBF solvers and the effect of preprocessing.

grid size	bloqqer	$\lvert\forall\rvert$	$\lvert\exists\rvert$	$\lvert F\rvert$	depQBF	QuBE	areqs2	GrayQBF
3×3	no	9	25	1710	0.03	0.08	0.48	**0.01**
3×3	yes	7	84	3767	**0.01**	0.05	0.14	**0.01**
4×4	no	16	36	3040	6.81	16.59	81.56	**0.56**
4×4	yes	15	33	2485	1.17	0.94	4.61	**0.54**
5×5	no	25	49	4750	$> 36,000$	$> 36,000$	$> 36,000$	**438.05**
5×5	yes	25	45	4195	24,712	30,597	11,578	**394.88**

Bounding box sizes larger than 11×11 will reveal information about how GoEs are spread over problem sizes. Therefore, we consider problem sizes of 6×6 to 14×14 as interesting. Only square grid sizes have been considered, to keep the number of tests limited.

With the current incremental SAT efficiency, 2^{30} generations can be checked in reasonable time (about a day). We used two Intel Xeon X5570 quadcore (@2,93 GHz) processors with 32 GB RAM. These processors contain 8 cores and thus the search space should not grow beyond 2^{33} generations. Enforcing internal symmetries restricts the search space to symmetric solutions and should be minimized. Symmetry breaking is used when applicable, as this does not restrict the search space.

Example 5. The 9×9 grid predecessor problem has 2^{81} states (generations). Applying σ_H, σ_D, or σ_{180} reduces the number of generations to approximately 2^{45} — larger than 2^{33}. Therefore a double internal symmetry needs to be used, for example (σ_H, σ_{180}). The number of distinctly defined cells is $\lceil 9/2\rceil\lceil 9/2\rceil$=25, resulting in $2^{25}/2^1 = 2^{24}$ generations.

Table 3: Overview of the experimental results. n indicates a grid of size $n{\times}n$, *IS* the internal symmetry enforced, *SB* symmetry breaking reductions, *size* the search space size, GoEs denoted the number of GoEs found, and *ET(h)* the summed execution time of all cores in hours.

n	*IS*	*SB*	*size*	GoEs	*ET(h)*
6	-	2^3	2^{33}	0	283.31
7	σ_{180}	2^2	2^{23}	0	0.30
7	σ_H	2^2	2^{26}	0	2.41
7	σ_D	2^2	2^{26}	0	2.34
8	σ_{180}	2^2	2^{30}	0	56.80
8	σ_H	2^2	2^{30}	0	54.70
9	σ_{90}	2^1	2^{20}	0	0.06
9	σ_H, σ_{180}	2^1	2^{24}	0	0.76
9	σ_D, σ_{180}	2^1	2^{24}	0	0.71
10	σ_{90}	2^1	2^{24}	8	1.21
10	σ_H, σ_{180}	2^1	2^{24}	0	1.05
10	σ_D, σ_{180}	2^1	2^{29}	62	38.75
11	σ_{90}	2^1	2^{30}	$42,044$	141.79
13	σ_{90}, σ_D	2^0	2^{28}	$1,501,648$	249.81
14	σ_{90}, σ_D	2^0	2^{28}	$673,925$	460.77

Experiments close to the limit of 2^{33} states have been solved in parallel by manually partitioning the search space. We fixed the innermost three cells to the 2^3 possible truth values to equally divide the load over eight cores. In terms of 2QBF, three universally quantified variables are expanded, and each of the eight resulting formulas is solved by a different core.

Table 3 summarizes the experiments. The number of GoEs found explodes with the problem size. This is expected by observing that every σ_{90}, σ_D GoE in an 11×11 grid is also included in 13×13 searches, where one extra ring of cells surrounds the orphan. This adds 7 free cells, yielding 128 GoEs of size 13×13 for every 11×11 GoE with equivalent internal symmetries.

That is also the reason why no experiment is performed with a 12×12 grid. All solutions found for n=12 (σ_{90}, σ_D) are included in the test with n=14 (σ_{90}, σ_D). The n=11 (σ_{90}) test is less restricted by internal symmetry than the n=13 (σ_{90}, σ_D) test; therefore both tests have been performed.

The number of GoEs has been plotted as function of the number of alive cells for n=14 in Figure 2. It shows a normal distribution with mean μ=115.753 and standard deviation σ=158.395 alive cells. Most GoEs have been found for an alive ratio of 0.592. For 13×13 grids this ratio is 0.574. It seems there is a relation between the density of Life and the occurrence of GoEs.

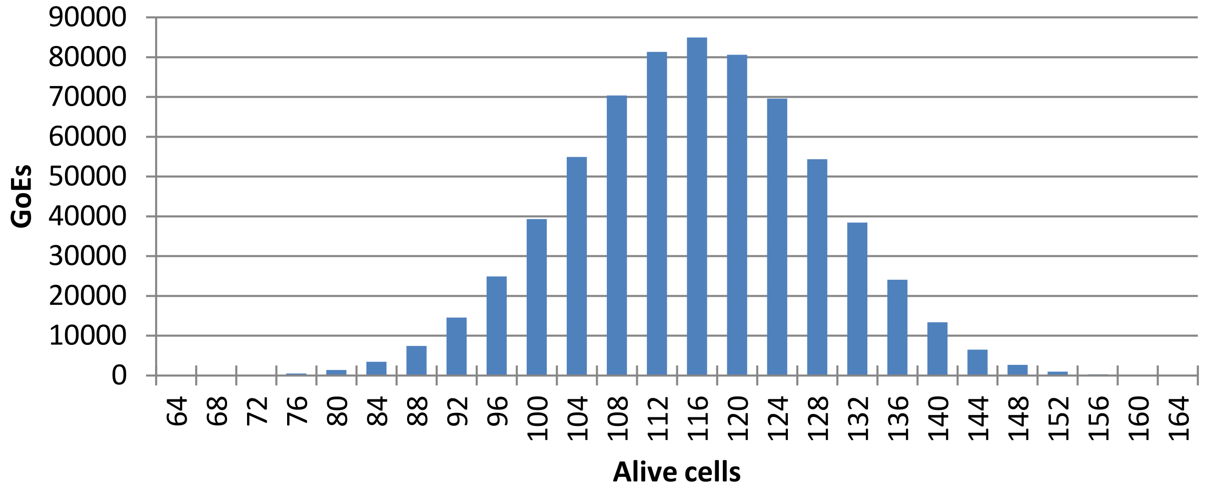


Figure 2: Number of GoEs vs. number of alive cells in 14x14

Many of the detected GoEs can be reduced to the same orphan. For instance, all 8 GoEs of n=10 (σ_{90}) can be reduced to two symmetric GoEs of which one is shown in Figure 3 (left). The 62 GoEs of the n=10 (σ_D, σ_{180}) test can be reduced to 37 orphans. The n=11 test resulted in 42044 GoEs which can be reduced to 24391 orphans. The smallest one defines 93 cells and is shown in Figure 4 (left). For the 13×13 and 14×14 tests, the number of GoEs was too large to determine all the orphans.

5.4 Improving existing records

During the experiments, we broke most existing records. We found an orphan with a bounding box of 10×10 with 56 alive and 36 dead cells, the smallest at the time of writing, see Figure 3 (left). With 92 defined cells, we also broke the record for the fewest defined cells. We nearly matched this record with an 11×11 bounded orphan of 93 defined cells, see Figure 4 (left).

The third record we broke is for the lowest density of alive cells, see Figure 3 (right). This orphan has 49 alive and 104 dead cells, yielding a density of 0.320. We did not improve the record of the fewest alive cells, but we matched the record of 45 alive cells, see Figure 4 (right). Notice that there are only two grey corners in this figure. By assigning two diagonal corners to grey, this generation has predecessors.

Finally, we improved the lower bound of 6×5 by proving that no orphans exist within a bounding box of 6×6 or smaller. Improving this lower bound to 6×7 or larger grid sizes falls outside currently available resources. A 6×7 experiment would last about 2^6 times as long as 6×6.

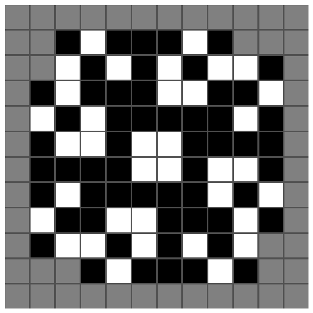
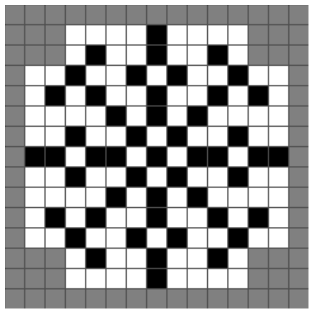



Figure 3: Our records. On the left is an orphan with smallest bounding box (10×10) and fewest defined cells (92). On the right is an orphan with the lowest density 0.320.

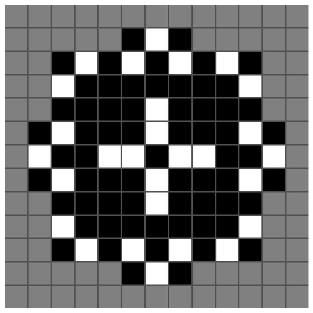
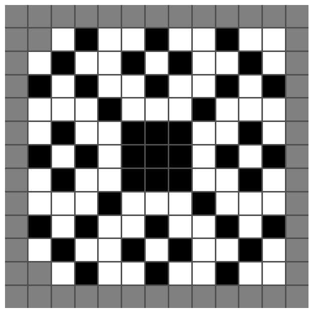



Figure 4: Almost records. On the left is an 11×11 with only 93 defined cells. On the right is an 11×11 orphan with a density of only 0.378 and 45 alive cells.

6 Conclusions and Future work

In this paper, we demonstrated how QBF and SAT techniques can be used to find Gardens of Eden in Conway's Game of Life. We used incremental SAT solving to perform predecessor checks for instances of Life. Reuse of heuristics in combination with systematic search strongly improved the performance of our checks; this may be the most effective feature of incremental SAT for applications consisting of many instances. The key to the performance boost is ensuring that similar instances are succeeding each other. This is best realized using Gray codes. We integrated this into our GrayQBF solver which appears to be very effective on small, hard 2QBF instances such as finding GoEs.

To further reduce the computational costs, we enforced observed internal symmetries. Additionally, we exploited the fact that our approach is trivial to parallelize by running the experiments on multiple cores. We broke most records regarding GoEs and increased the lower bound.

Still the challenge remains: can someone prove the existence of a smaller orphan? Or find orphans with fewer living cells, a lower density of living cells, or fewer defined cells?

Acknowledgements

We thank Donald Knuth and Nathan Wetzler for detailed comments and suggestions on a draft version of this paper.

References

[1] Abdelwaheb Ayari and David A. Basin. Qubos: Deciding quantified boolean logic using propositional satisfiability solvers. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, FMCAD '02, pages 187–201, London, UK, UK, 2002. Springer-Verlag.

[2] Stuart Bain. Time-reversal in Conway's Life as SAT. In *Proceedings of the 20th Australian joint conference on Advances in artificial intelligence*, AI'07, pages 614–618, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] Nicolay Beluchenko. Garden of Eden search. http://oeis.org/A196447, 2011.

[4] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for qbf. In *Proceedings of the 23rd international conference on Automated deduction*, CADE'11, pages 101–115, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Robert Bosch and Michael Trick. Constraint programming and hybrid formulations for three life designs. *Annals OR*, 130(1-4):41–56, 2004.

[6] Aaron R. Bradley. SAT-based model checking without unrolling. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Edmund M. Clarke, Robert P. Kurshan, and Helmut Veith. The localization reduction and counterexample-guided abstraction refinement. In *Essays in Memory of Amir Pnueli*, pages 61–71, 2010.

[8] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. SAT*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.

[9] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[10] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[11] Achim Flammenkamp List of orphan patterns. http://wwwhomes.uni-bielefeld.de/achim/orphan.html.

[12] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, October 1970.

[13] Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. Qube7.0. *JSAT*, 7(2-3):83–88, 2010.

[14] Frank Gray. Pulse code communication, 03 1953.

[15] Youssef Hamadi, Said Jabbour, and Lakhdar Saïs. Control-based clause sharing in parallel sat solving. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 499–504, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[16] J. Hardouin-Duparc. Paradis terrestre dans l'automate cellulaire de Conway. *Rev. Française Automat. Informat. Recherche Operationnelle Ser. Rouge*, 8:64–71, 1974.

[17] Marijn J. H. Heule and Toby Walsh. Symmetry within solutions. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI '10)*, pages 77–82. AAAI Press, 2010.

[18] Mikoláš Janota and Joao Marques-Silva. Abstraction-based algorithm for 2qbf. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 230–244, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] Matti Järvisalo, Armin Biere, and Marijn J. H. Heule. Blocked clause elimination. In *TACAS'10*, volume 6015 of *LNCS*, pages 129–144. Springer, 2010.

[20] Florian Lonsing and Armin Biere. Depqbf: A dependency-aware qbf solver. *JSAT*, 7(2-3):71–76, 2010.

[21] Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. *Conflict-Driven Clause Learning SAT Solvers*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, pages 131–153. IOS Press, February 2009.

[22] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[23] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of the 10th international conference on Theory and applications of satisfiability testing*, SAT'07, pages 294–299, Berlin, Heidelberg, 2007. Springer-Verlag.

[24] Barbara Smith. A dual graph translation of a problem in 'life'. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, volume 2470 of *Lecture Notes in Computer Science*, pages 89–94. Springer Berlin / Heidelberg, 2006.

[25] Robert Wainwright. Lifeline volume 3, 1971.

A 2QBF solver GrayQBF

The figure below shows the C code of our 2QBF solver that actually transforms a 2QBF formula in DIMACS format into an iCNF file for incremental SAT solvers in DIMACS format.

```
#include <stdio.h>
#include <stdlib.h>

void main (int argc, char **argv) {
    long long i, j, Gray;
    int literal, nVars, nLines, tmp, *universal, size = 0;

    FILE *file = fopen (argv[1], "r");

    do {                                       // find the "p cnf" line
         tmp = fscanf (file, " p cnf %i %i \n", &nVars, &nLines);
         if (tmp > 0 && tmp != EOF) break;
         tmp = fscanf (file, "% *s\n");        // skip a comment line
    }
    while (tmp != 2 && tmp != EOF);

    universal = (int*) malloc (sizeof(int) * nVars);

    fscanf (file, " a ");                      // parse the universals
    for (;;) {
        fscanf (file, " %i ", &literal);
        if (literal == 0) break;               // found end of a-line
        else universal[ size++ ] = literal;    // add universal to array
    }
    fscanf (file, "% *s\n");                   // ignore the 2QBF e-line

    printf("p inccnf\n");                      // print the format line
    while (nLines) {                           // copy the clauses
        fscanf (file, " %i ", &literal);
        printf("%i ", literal);
        if (literal == 0) { printf("\n"); nLines--; }
    }

    for (i = (1LL << size)-1; i >= 0; i--) {   // loop over the universal
        Gray = (i>>1) ^ i;                     // variables using Gray codes
        printf("a ");                          // to maximally reuse the
        for (j = 0; j < size; j++)             // heuristics of the solver
           printf ("%i ", ((Gray >> j) & 1LL)? universal[j]:-universal[j]);
        printf ("0\n");
    }

    fclose (file);
    free (universal);
}
```