Annals of Operations Research 130, 41–56, 2004

Constraint Programming and Hybrid Formulations for Three Life Designs

ROBERT BOSCH bobb@cs.oberlin.edu
Department of Mathematics, Oberlin College, Oberlin, OH, USA

MICHAEL TRICK trick@cmu.edu
Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract. Conway's game of Life provides an interesting testbed for exploring issues in formulation, symmetry, and optimization with constraint programming and hybrid constraint programming/integer programming methods. We consider three Life pattern-creation problems: finding maximum density still-Lifes, finding smallest immediate predecessor patterns, and finding period-2 oscillators. For the first two problems, integrating integer programming and constraint programming approaches provides a much better solution procedure than either individually. For the final problem, the constraint programming formulation provides the better approach.

Keywords: integer programming, constraint programming, hybrid formulation, cellular automata, game of Life

Introduction

In the late 1960s, John Horton Conway invented the one-player game called Life (Berlekamp, Conway, and Guy, 1982). This game, particularly once popularized by Martin Gardner in his *Scientific American* columns (Gardner, 1970, 1971, 1983), attracted a huge following. Many amateur and professional mathematicians and computer scientists studied the game, and in the course of their work, they developed (or at least illustrated) many aspects of discrete dynamical systems.

In Life, the player places checkers on some of the squares of an infinite checkerboard. The player then follows some simple rules to create a new pattern on the board, replacing the current checkers. The resulting sequences of patterns are both aesthetically pleasing and mathematically interesting. Perhaps the most surprising result is that by clever placement of the checkers and appropriate interpretation of the patterns, it is possible to create a Turing-equivalent computing machine (see (Berlekamp, Conway, and Guy, 1982) for a summary of this work).

We begin with the rules of Life. Life is a single-player board game, where the board is a checkerboard that extends to infinity in each direction. Each square of the board is a *cell* and each cell has eight *neighbors*: the eight cells that share one or two corners with it, as shown in figure 1.

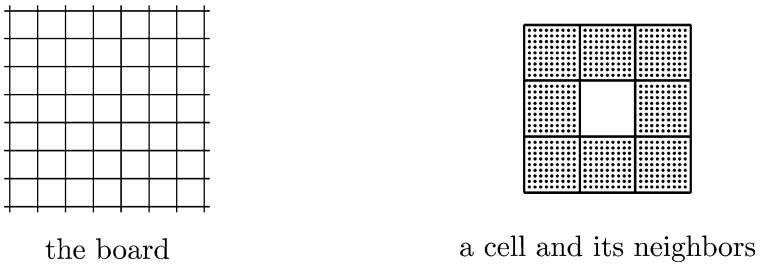


Figure 1.

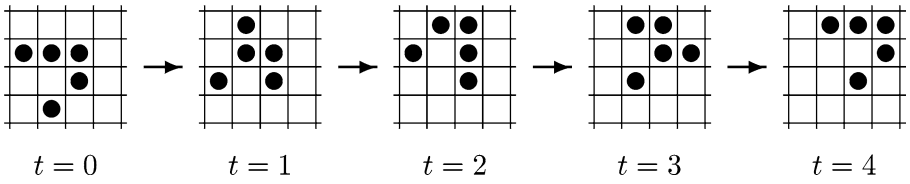


Figure 2.

To begin the game, the player places checkers on some of the cells of the board, creating an initial pattern. A cell with a checker in it is *living* and those without are *dead*.

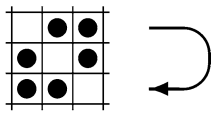
The pattern is then modified by applying the following rules over and over again:

- If a cell has exactly two living neighbors then its state remains the same in the next pattern (if living, it remains living; if dead, it remains dead).
- If a cell has exactly three living neighbors, then it is living in the next pattern. This is the "birth" condition.
- If a cell has fewer than two or more than three living neighbors then it is dead in the next pattern. These are the "death-by-isolation" and "death-by-overcrowding" conditions, respectively.

These simple rules are sufficient to generate incredibly complicated patterns and dynamics. As a simple example, consider the sequence in figure 2. Note that each configuration of checkers "moves" one row up and one column to the right every four time iterations.

We look at three problems in creating interesting Life patterns:

1. In a given, finite region of the board, construct a maximium density still-Life. A still-Life is a pattern that does not change from one time period to the next.
2. Find a smallest immediate predecessor of a given pattern, where "smallest" means "least number of living cells."
3. In a given, finite region of the board, construct a 2-oscillator that has the maximum number of changing cells. A 2-oscillator consists of patterns A and B such that A is the successor of B and B is the successor of A.



$t = 0, 1, 2 \ldots$

Figure 3.

These problems illustrate standard issues in any discrete dynamical system, so insights gained here may go beyond the Life game. Algorithmically, we feel that the main interesting aspects of these problems are:

1. Basic formulations are very straightforward in constraint programming and quite arcane in integer programming.
2. The interplay of feasibility and optimization makes them a good setting for exploring the unique role objective functions take in CP formulations.
3. The symmetry embedded in these problems is very strong and leads both to algorithmic insights and algorithmic difficulties.
4. The problems are well-known, simple to state, and difficult even for relatively small instances.

The remaining sections address the three problems in turn.

1. Finding still-Lifes

A still-Life is a pattern that doesn't change, so that any cell that is alive in one period is alive in the next, and every dead cell remains dead. It is easy to see that still-Lifes exist. For instance, the empty pattern is a still-Life. A slightly more interesting still-Life is shown in figure 3.

Given a fixed region of the board, what is the most checkers we can have in a still-Life? The example in figure 3 shows that for a 3 by 3 region, it is possible to have 6 checkers. It is easy to show that it is not possible to have 7 or more, so the maximum density of a 3×3 still-Life is $6/9 = 2/3$ (the number of checkers over the area of the region). Note that the region is still embedded in the infinite board, so that no cell outside the region can either be alive or become alive according to the rules.

There has been work done on computational and theoretical aspects of still-Lifes. Bosch (1999, 2000) looked at a number of uses of integer programming in Life and its variants. Much of his work will be discussed in detail later in this section. Elkies (1998) has shown that for the infinite board the maximum density is $1/2$. (For finite size boards, no exact formula is known for the maximum density.) Callahan provides software (Callahan, 2001) that uses local search to find still-Life patterns. Cook (2001) has explored the computational complexity issues of identifying when a still-Life (or stable pattern in his terminology) is made up of two or more independent pieces.

Niemiec has put together a Web page (Niemiec, 2001) with listings of all small still-Lifes.

1.1. Constraint and integer programming formulations

We will consider finding still-Lifes in a square n by n region. (Rectangular or other shaped regions lead to similar results.)

An initial formulation for this problem is an almost trivial exercise in constraint programming. The rules define constraints of the form: "If a cell is living then 2 or 3 of its neighbors are living. If a cell is dead then it does not have 3 living neighbors." This creates constraints of the following form (extracted from the OPL formulation; all full formulations are available at `mat.gsia.cmu.edu/LIFE`):

```
(life[i,j]=1) =>
  2 <= sum(n in neighbor) life[i+n.row,j+n.col] <= 3;
(life[i,j]=0) =>
  sum(n in neighbor) life[i+n.row,j+n.col] <> 3.
```

In this formulation, `life` is a 0–1 matrix which is 1 if the cell is living and 0 otherwise; `neighbor` is a set giving the offsets to find the neighbors of a cell. Additional constraints are needed to prohibit three adjacent live cells along the border of the region (such a construct would create a live cell outside the region). Conceptually, this is an extremely easy formulation.

For integer programming, in contrast, the formulation of this problem takes a good amount of creativity. Bosch (1999) examines two formulations and determines that the following is the better approach:

Let x_{ij} be 1 if cell (i, j) is alive and 0 otherwise. Let $N(i, j)$ represent the cells that are neighbors of (i, j). Then, the constraint

$$3x_{ij} + \sum_{(i',j')\in N(i,j)} x_{i'j'} \leqslant 6$$

enforces the death-by-overcrowding condition: if a cell is alive, then at most three of its neighbors are alive. It is straightforward to see that if the cell is dead, then at most six of its neighbors are alive (otherwise one of the living cells would be overcrowded).

To handle the death-by-isolation condition, the following constraint suffices:

$$2x_{ij} - \sum_{(i',j')\in N(i,j)} x_{i'j'} \leqslant 0.$$

If a cell is alive then at least two of its neighbors are also.

To prohibit violations of the birth condition, we need a large number of somewhat more complicated constraints. For each 3-element subset S of $N(i, j)$, we need

$$-x_{ij} + \sum_{(i',j')\in S} x_{i'j'} - \sum_{(i',j')\in N(i,j)-S} x_{i'j'} \leqslant 2.$$

In words, if all of S is alive, then either (i, j) is alive or some element of $N(i, j) - S$ is alive.

Together with the border constraints, the above constraints are sufficient to define an integer programming formulation for this problem (again, see the full OPL formulation at `mat.gsia.cmu.edu/LIFE`). This formulation, of course, is much less straightforward than the CP formulation.

1.2. Avoiding symmetry

Before we begin trying to solve these models, it is worthwhile to explore the symmetry issues in this problem. For any still-Life, it is possible to create an equivalent still-Life through a number of operations:

1. Rotate the square by 90, 180 or 270 degrees.

2. Reflect the pattern either horizontally, vertically, or along one long diagonal.

3. Combinations of rotations and reflections.

It would be very useful to find a set of constraints that would allow us to generate only one of the equivalent patterns, ignoring the others. To date, we have not been able to find a usable constraint that can be added to either our CP or IP formulations. Another possibility would be to adapt Smith's symmetry-breaking-during-search method (Smith, 2001).

Instead, we can add constraints that cut down on the redundant patterns. The simplest set of constraints recognizes that the top half and bottom half of a pattern may differ in density, so we can choose to look only at patterns where the top half is at least as dense as the bottom half. Similarly, and independently, we can restrict our search to those whose left half is at least as dense as its right half. These two simple linear constraints can be added to either the CP or the IP formulation.

At this point, we are ready to provide our first results. In table 1, ChPo refers to "Choice Points" (or nodes of the search tree for IP formulations) and T is time in seconds (Intel 650 MHz Pentium III with 196 MB memory running ILOG's OPL Studio 3.5).

Table 1
Results on base and symmetry.

Size	Value	CP w/o Sym (CP1)		CP with Sym (CP2)		IP w/o Sym (IP1)		IP with Sym (IP2)	
		T	ChPo	T	ChPo	T	ChPo	T	ChPo
4	8	0.02	275	0.02	259	0.41	0	0.37	0
5	16	0.04	636	0.05	636	0.76	1	0.50	0
6	18	1.51	24472	1.64	24083	22.68	1952	8.22	504
7	28	9.44	154151	10.07	154147	7.10	55	4.94	10
8	36	189.75	3084106	205.37	3082922	65.34	798	22.1	348

None of these four formulations were able to solve the size-9 instance in under half an hour (for instance, the basic CP formulation took 19,637 seconds or a bit over 5 hours).

While it is clear that the symmetry breaking helps the integer programming formulation, it does not aid the CP very much (while it slightly reduces the number of choice points, the overall time goes up slightly). We will explore this in the next section.

1.3. Linear constraints as a global constraint

Examining the details of the runs for the constraint programming model makes it clear why they are inefficient. We set the search routine to begin with the first row and try to set as many cells to one as possible; it then moves down to the next row.

Consider a partial solution, with the variables instantiated for the first few rows. At this point, the constraint propagation does not go any "deeper" into the board than one row beyond what is already instantiated. For all remaining cells, their domain remains {0, 1}. The resulting bound on the objective value is very weak: with all the uninstantiated cells being possibly alive, the possible objective is very high. A very large portion of the board must be instantiated before the objective function has any force. This leads to a huge number of choice points and a long computing time. This also explains why the "anti-symmetry" constraints do not work well with this CP model: it takes too long for the search to recognize that they are binding.

To solve this, we need to provide some global information to the search on how well a partial pattern can be completed. This information could be developed a number of ways. For instance, since we know that every 3 by 3 square can have no more than 6 live cells, we could cover the board in 3 by 3 squares linked to the original formulation. We could then have the objective be set equal to a covering of the board with the 3 by 3 squares. Such a method would give some "future" information to the search. In fact, we have experimented with such an approach, and it does improve on the results so far.

There is a simpler method, however, that involves linking the integer programming approach to the constraint programming approach. Starting with the standard CP formulation, we can add the integer programming constraints as linear inequalities. We can then use the linear relaxation of these inequalities to bound the possible values of the uninstantiated variables. In doing so, the "global constraints" of the linear inequalities work together with the logical constraints to greatly improve the search.

A blind application of this approach however, is not the best solution. In particular, a natural choice would be to simply combine the two formulations. This approach does not even improve on the approaches already examined. A more efficient formulation recognizes the role the linear constraints are playing. Our goal here is to give the search a bound on the best values for the uninstantiated variables: there is a tradeoff between quality of bound and time taken to find the bound. In our integer programming formulation, for each cell there is

1. one death-by-crowding constraint,

Table 2
Results on hybrid approaches.

Size	Value	IP with Sym all cons. (IP2)		CP/IP with Sym all cons. (HYB1)		CP/IP with Sym no birth (HYB2)		CP/IP with Sym no birth/iso (HYB3)	
		T	ChPo	T	ChPo	T	ChPo	T	ChPo
8	36	22	348	47	1810	3	2310	2	2474
9	43	–	–	–	–	85	46345	51	48975
10	54	–	–	–	–	291	98082	147	102865
11	64	–	–	–	–	655	268520	373	318942
12	76	–	–	–	–	49166	11772869	30360	12733484
13	90	–	–	–	–	50871	10996298	30729	11845960

2. one death-by-isolation constraint, and

3. 56 birth constraints.

We can decrease the number of linear constraints by approximately 95% simply by not including the birth constraints. The logical constraints in the CP side will enforce the birth condition.

In fact, we can go even farther. Death-by-isolation, while a necessary constraint, is not often binding in dense still-Lifes. Again, we can allow the logical constraints to enforce this requirement and only keep the key death-by-overcrowding linear constraints. This decreases the number of the linear constraints by an additional 50%.

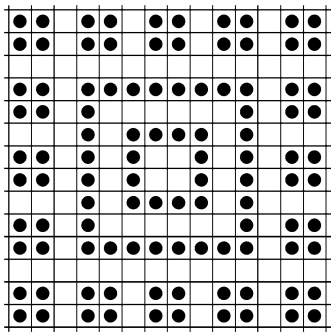
This gives three possible formulations: HYB1 has all of the linear constraints, HYB2 removes the birth constraints, and HYB3 removes both the birth constraints and the death-by-isolation constraints. Table 2 displays the results for larger instances. Figures 4 and 5 display optimal solutions to the largest instances we have solved. (HYB3 required approximately 6 days to solve the size-14 instance and just over 8 days to solve the size-15 instance.)

While we have developed methods based on integer programming that can also prove the optimality of the size-12 and size-13 instances, the method presented here seems to be much faster (roughly a factor of 5 on the larger problems).

1.4. Exploiting symmetry

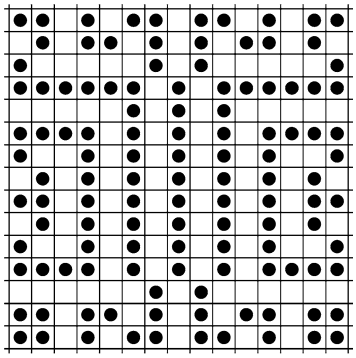
As a final improvement, we also have tried to exploit the symmetry in this game by finding feasible solutions that are either horizontally or vertically symmetrical, or are symmetric after rotation by 180 degrees. In all cases we have examined, there exists an optimal solution with such a symmetry. For any given symmetry, finding the maximum density pattern with that symmetry simply forces symmetric cells to take on identical values. The result is an instance with far fewer degrees of freedom.

Table 3 gives the optimal symmetric solutions found by this approach. We conjecture that there exist no better nonsymmetric solutions. We can use these values to help prove optimality by adding constraints to force the generation only of better solutions.



optimal size-14 still-Life
(value = 104)

Figure 4.



optimal size-15 still-Life
(value = 119)

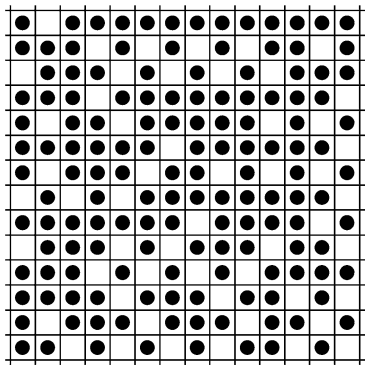
Figure 5.

Table 3
Symmetric, feasible solutions.

Size	16	17	18
Objective	136	152	170

2. The smallest immediate predecessor problem

The smallest immediate predecessor problem (SIPP) involves going backwards in Life. Suppose we encounter some people playing Life, and they tell us that they have applied Conway's rules just once (to every cell of the board). They then ask us if we can determine what their initial pattern was. Such questions have important algorithmic consequences. In many discrete dynamical systems, "going backwards" is much more difficult than going forwards, so discrete dynamical systems provide a sort of "one-way



GoE

Figure 6.

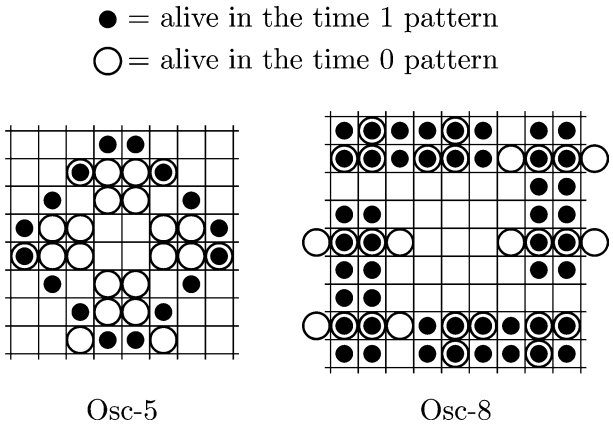


Figure 7.

function." Determining the computational difficulty of deducing the predecessor of a position addresses the strength of this one-way function. Furthermore, in the Life community, there has been a significant amount of work in finding "Garden of Eden" (GoE) positions for which there is no immediate predecessor. (Figure 6 displays a GoE found by Achim Flammenkamp.) Most of the effort in this work has been to prove a position has no predecessor. This section provides the first computational method to do that.

Now clearly there is no way to uniquely determine what the "time 0" pattern was, given the "time 1" pattern. After all, a pattern can have an infinite number of immediate predecessors. (Consider, for example, an infinite board and the pattern that has no living cells. Every pattern that has exactly one living cell is an immediate predecessor of this pattern.) The SIPP is concerned with finding, amongst all of the patterns that could have been the time 0 pattern, a pattern with the fewest number of living cells. Figure 7 displays two instances of the SIPP, along with optimal solutions (which are not unique).

2.1. *Formulations for the SIPP*

Let A stand for the set of cells that are alive in the time 1 pattern, D for the set of cells that are dead in the time 1 pattern, and $N(i, j)$ for the set of cells that are neighbors of cell (i, j).

We begin with an integer programming formulation for this problem.

Let x_{ij} equal 1 if cell (i, j) is alive in the time 0 pattern and 0 otherwise. Consider the following integer program for solving instances of the SIPP:

$$\min \sum_{(i,j)} x_{ij}$$

s.t.

$$\sum_{(i',j')\in N(i,j)} x_{i'j'} \leqslant 3 \qquad \forall (i, j) \in A,$$

$$x_{ij} + \sum_{(i',j')\in N(i,j)} x_{i'j'} \geqslant 3 \qquad \forall (i, j) \in A,$$

$$\sum_{(i',j')\in S} x_{i'j'} - \sum_{(i',j')\in N(i,j)-S} x_{i'j'} \leqslant 2 \qquad \forall (i, j) \in D\colon\ \forall S \subseteq N(i, j)\colon\ |S| = 3,$$

$$2x_{ij} + 2\sum_{(i',j')\in S} x_{i'j'} - \sum_{(i',j')\in N(i,j)-S} x_{i'j'} \leqslant 4 \qquad \forall (i, j) \in D\colon\ \forall S \subseteq N(i, j)\colon\ |S| = 2,$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in A \cup D.$$

The first two sets of inequalities, taken together, guarantee that all of the cells that are supposed to be alive in the time 1 pattern are in fact alive then. The inequality

$$\sum_{(i',j')\in N(i,j)} x_{i'j'} \leqslant 3$$

makes sure that (i, j) has at most three living neighbors in the time 0 pattern. The inequality

$$x_{ij} + \sum_{(i',j')\in N(i,j)} x_{i'j'} \geqslant 3$$

makes sure that if (i, j) is alive at time 0, then (i, j) has at least two living neighbors at time 0, and *also* that if (i, j) is dead at time 0, then (i, j) has at least three living neighbors then.

The third and fourth sets of inequalities play a similar role, but for the cells that are supposed to be dead in the time 1 pattern.

This IP formulation works well on very small instances and on some larger instances that have a very small number of dead cells in the time 1 pattern. It does not work well on other instances, including those in the testbed we are considering. One explanation for this is that the formulation uses only two linear constraints for each living cell but $\binom{8}{3} + \binom{8}{2} = 84$ linear constraints for each dead cell in the time 1 pattern. The number of constraints make this approach uncompetitive for the instances in this test.

As might be expected, the constraint programming formulation is much more straightforward. Here is some OPL pseudocode:

```
forall ((i,j) in A) {
  sum((i1,j1) in N(i,j)) x[i1,j1] <= 3;
  x[i,j] + sum((i1,j1) in N(i,j)) x[i1,j1] >= 3;
}

forall ((i,j) in D) {
  sum((i1,j1) in N(i,j)) x[i1,j1] >= 4 \/
  x[i,j] + sum((i1,j1) in N(i,j)) x[i1,j1] <= 2;
}
```

The first two sets of inequalities are from the IP formulation. They ensure that all of the cells that are supposed to be alive in the time 1 pattern are in fact alive then. The remaining inequalities (the disjunctions) make sure that the dead cells are dead. They do this by stipulating that each cell that is dead in the time 1 pattern (i) has four or more living neighbors at time 0, (ii) is dead at time 0 and has at most two living neighbors then, or (iii) is alive at time 0 and has at most one living neighbor then. (The first part of the disjunction expresses condition (i); the second part expresses both (ii) and (iii).)

2.2. CP versus CP/IP

We compared the CP formulation described above to a hybrid CP/IP formulation. In the hybrid formulation, we again use the linear constraints as a global constraint (by minimizing "with linear relaxation" in OPL Studio). Just as in the previous section, including all of the linear constraints is computationally ineffective. Instead we include only the constraints for the live cells, but not the much more numerous constraints for the dead cells. For these, the more compact constraint programming formulation will enforce the appropriate predecessor cells.

For test instances of the SIPP, we used numerous "famous" Life patterns. For example, the figure 7 patterns are well-known oscillators of period 5 and 8, respectively. The figure 8 patterns are "spaceships" (constructed by Dean Hickerson and Tim Coe, respectively) that move to the right (one cell every three iterations and one cell every four iterations, respectively). The results for these four instances are given in the top portion of table 4. (The results for the other test instances were similar.) We also tried to find a smallest immediate predecessor of Flammenkamp's GoE; the results are displayed in the bottom of table 4. In each case, the computations were performed with ILOG's OPL Studio 3.0.2 on an Intel 800 MHz Pentium III with 384 MB memory.

On the "famous" Life patterns test instances, the hybrid CP/IP approach performed much better than the pure CP approach. On the GoE instance, the CP approach was vastly superior. (This is no surprise; on an infeasible instance, the objective function provides no help whatsoever.)

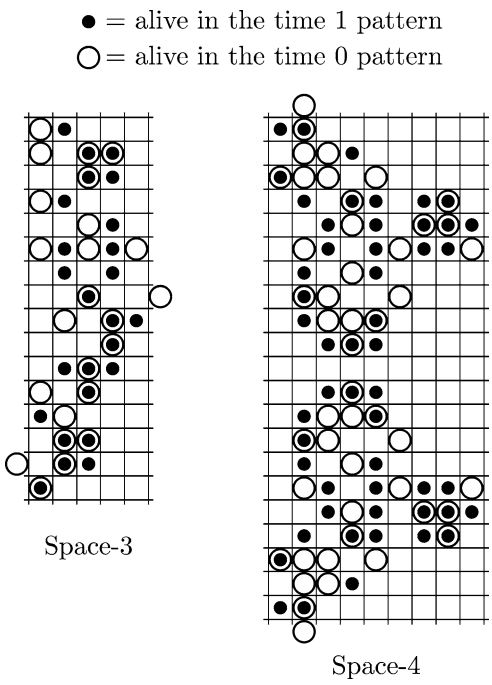


Figure 8.

Table 4
Results on CP versus hybrid.

Time 1	CP		Hybrid	
pattern	T	ChPo	T	ChPo
Osc-5	8294	183982210	576	2280727
Osc-8	7364	181567598	1	4603
Space-3	589	13687792	31	131176
Space-4	–	–	2812	6859533
GoE	811	24881972	55781	23579748

3. Finding period-2 oscillators

For our final problem, we turn to some of the most beautiful Life patterns: oscillators. Oscillators are patterns which repeat themselves after some number of time periods (the period of the oscillator). Still-Lifes are period-1 oscillators; they repeat each period. A period-2 oscillator repeats every second period. Oscillators are known for *most* small periods and all periods greater than 57 (see (Buckingham and Callahan, 1998)). We address the problem of finding period-2 oscillators. Examples of such oscillators are shown in figure 9, where open circles represent one time period, and filled circles the other. (Note that these diagrams do not do justice to the beauty of these patterns.)

period-2 oscillators

Figure 9.

For a finite region of the plane, there are many definitions possible for the "best" oscillator that fills that region: one might choose to maximize the total number of living cells for both time periods, or to maximize the difference between the number of living cells in one period and the number in the other. We choose to maximize the number of cells that differ in the two periods. In some sense, this gives the maximum amount of "blinking" in the pattern.

If we let `life1[i,j]` and `life2[i,j]` represent whether cell (i, j) is alive in periods 1 and 2, respectively, the constraint program for the maximum-difference period-2 oscillator is again straightforward for the constraint program. Here is our OPL pseudocode:

```
Minimize
  sum((i,j)) (life1[i,j] <> life2[i,j]);
Subject to
  forall ((i,j)){
    sum((i1,j1) in N(i,j)) life1[i1,j1] = 3
      => life2[i,j] = 1;
    sum((i1,j1) in N(i,j)) life1[i1,j1] = 2
      => life2[i,j] = life1[i,j];
    sum((i1,j1) in N(i,j)) life1[i1,j1] < 2
      => life2[i,j] = 0;
    sum((i1,j1) in N(i,j)) life1[i1,j1] > 3
      => life2[i,j] = 0;
    sum((i1,j1) in N(i,j)) life2[i1,j1] = 3
      => life1[i,j] = 1;
    sum((i1,j1) in N(i,j)) life2[i1,j1] = 2
      => life1[i,j] = life2[i,j];
    sum((i1,j1) in N(i,j)) life2[i1,j1] < 2
      => life1[i,j] = 0;
    sum((i1,j1) in N(i,j)) life2[i1,j1] > 3
      => life1[i,j] = 0;
}
```

The first set of constraints ensure that period 2 follows from period 1, while the second set ensures that period 1 follows from period 2.

Table 5
Finding period-2 oscillators.

Size	Value	IP		CP		CP/IP	
		T	ChPo	T	ChPo	T	ChPo
3	4	6.24	1318	0.01	80	0.06	80
4	8	102.13	5129	0.11	1035	0.71	987
5	8	–	–	1.45	11742	9.43	11078
6	8	–	–	16.70	153725	125.79	136461
7	18	–	–	160.03	1489806	1316.28	1315814
8	28	–	–	1548.20	13441626	–	–

The integer programming formulation follows the same general structure, but each of the constraints is more difficult. In this case, we explicitly need a variable `differ[i,j]`, which is 1 if the cell differs between periods 1 and 2, and 0 if it does not. To relate the `differ` variables to the `life1` and `life2` variables, we require:

```
forall ((i,j)) {
  differ[i,j] <= life1[i,j] + life2[i,j];
  differ[i,j] <= 2 - life1[i,j] - life2[i,j];
}
```

(This seemingly arcane set of inequalities is a standard trick to formulate the absolute value function for 0–1 variables when maximizing.)

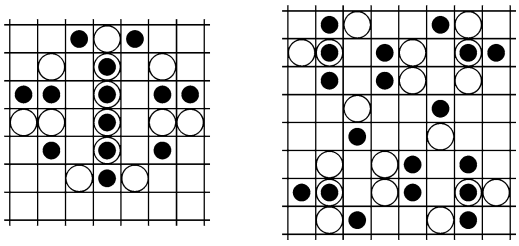
The rest of the formulation is similar to the formulation for still-Lifes except the live cells in one period affect the cells in the other. For instance, the overcrowding constraint becomes

```
forall ((i,j)) {
  5*life2[i,j] + sum((i1,j1) in N(i,j)) life1[i1,j1] <= 8;
  5*life1[i,j] + sum((i1,j1) in N(i,j)) life2[i1,j1] <= 8;
}
```

Note that in this case, we cannot strengthen the constraints by assuming that there are no more than six live cells next to any cell: that argument does not hold of oscillating patterns with period more than 1. Birth constraints, death-by-isolation constraints, and border constraints can be similarly added (see Web page for full formulation).

Symmetry again plays a role in this problem, with an additional symmetry between the pattern in time 1 and the pattern in time 2. We add constraints forcing the pattern in time 1 to have more live cells, along with more live cells in the top half and in the left half.

We again can combine the integer and linear programming formulations by taking just the overcrowding and isolation constraints from the integer program and using their linear relaxation as a global constraint for the constraint program (the birth constraints are again too numerous to be computationally effective).



optimal period-2 oscillators

Figure 10.

The results for this problem (see table 5) are striking. Not only is the constraint program much more effective than the integer program, but even the hybrid approach is inferior to the constraint program alone. In this case, the linear relaxation is giving insufficient information to be computationally useful.

See figure 10 for pictures of the optimal size-7 and size-8 period-2 oscillators.

4. Conclusions and future research

Finding Life patterns is an interesting challenge for constraint programming and integer programming. The constraints to enforce the transition rules are much more natural in constraint programming, but the global view necessary to quickly prove optimality is best handled by the linear constraints of integer programming. The combination is much better than either individual approach. Key to this success are a number of insights:

1. Symmetry breaking is important, but only when the symmetry possibilities can be recognized early in the search.
2. Symmetry can be exploited to decrease problem dimension to quickly get good feasible solutions.
3. When adding linear constraints as global constraints, it is important to add only what is necessary to find good bounds.
4. Sometimes the amount of information you get from the linear constraints is insufficient to be computationally effective.

While this method does appear to be the fastest at the moment, there is ample opportunity for improvement. In particular, the symmetry breaking being done seems ripe for improvement. Also, a more intelligent method of adding linear constraints seems a good opportunity.

Acknowledgments

A preliminary version of this section was presented at FORMUL'01, Cyprus, 2001, and the authors thank the participants of that workshop for their feedback.

References

Berlekamp, E.R., J.H. Conway, and R.K. Guy. (1982). *Winning Ways for Your Mathematical Plays*, Vol. 2: *Games in Particular*. London: Academic Press.

Bosch, R.A. (1999). "Integer Programming and Conway's Game of Life." *SIAM Review* 41(3), 594–604.

Bosch, R.A. (2000). "Maximum Density Stable Patterns in Variants of Conway's Game of Life." *Operations Research Letters* 27(1), 7–11.

Buckingham, D.J. and P.B. Callahan. (1998). "Tight Bounds of Periodic Cell Configurations in Life." *Experimental Mathematics* 7(3), 221–241.

Callahan, P. (2001). *Random Still Life Generator*, `http://www.radicaleye.com/lifepage/stilledit.html`.

Cook, M. (2001). *Still Life Theory*, `http://paradise.caltech.edu/~cook/Workshop/CAs/2DOutTot/Life/StillLife/StillLifeTheory.html`.

Elkies, N.D. (1998). "The Still-Life Density Problem and Its Generalizations." In P. Engel and H. Syta (eds.), *Voronoi's Impact on Modern Science*, Book 1. Kyiv: Institute of Mathematics.

Gardner, M. (1970). "The Fantastic Combinations of John Conway's New Solitaire Game "Life"." *Scientific American* 223, 120–123.

Gardner, M. (1971). "On Cellular Automata, Self-Reproduction, the Garden of Eden and the Game "Life"." *Scientific American* 224, 112–117.

Gardner, M. (1983). *Wheels, Life, and Other Mathematical Amusements*. New York: W.H. Freeman.

Niemiec, M.D. (2001). *Mark D. Niemiec's Life Page*, `http://home.interserv.com/~mniemiec/lifepage.htm`.

Smith, B.M. (2001). "Reducing Symmetry in a Combinatorial Design Problem." In *CP-AI-OR 2001*, Wye College, Kent.