Solving 'Still life' with Soft Constraints and Bucket Elimination ⋆

Javier Larrosa and Enric Morancho

Universitat Politecnica de Catalunya, Barcelona, Spain,
larrosa@lsi.upc.es, enricm@ac.upc.es

Abstract. In this paper we study the applicability of *bucket elimination* (BE) to the problem of finding *still-life patterns*. Very recently, it has been tackled using *integer programming* and *constraint programming*, both of them being search-based methods. We show that BE, which is based on *dynamic programming*, provides an exponentially lower worst-case time complexity than search methods. Unfortunately, BE requires exponential space, which is a disadvantage over the polynomial space requirement of depth-first search.
With our experiments, we show that BE is quite competitive with search-based approaches. It clearly outperforms simple encodings and it is comparable with dedicated methods. While the best current search approach solves the $n = 14$ instance in about 6 cpu days, BE solves it in about 1 day. BE cannot solve the $n = 15$ instance due to space exhaustion (this instance is solved by search in 8 days). Finally, we show how BE can be adapted to exploit the problem symmetries, with which in several cases we outperform previous results in a relaxation of the problem which restrict solutions to symmetric patterns, only.

1 Introduction

The game of *life* was invented in the late 60s by John Horton Conway and was later popularized by Martin Gardner [6]. Given an infinite checkerboard, the only player places checkers on some of its squares. Each square is a *cell*. If there is a checker on it, the cell is *alive*, else it is *dead*. Each cell has eight *neighbors*: the eight cells that share one or two corners with it. The state of the board evolves iteratively according to three rules: (i) if a cell has exactly two living neighbors then its state remains the same in the next iteration, (ii) if a cell has exactly three living neighbors then it is alive in the next iteration and (iii) if a cell has fewer than two or more than three living neighbors, then it is dead in the next iteration.

While conceptually simple, the game has proven mathematically interesting and has attracted a lot of curiosity, as can be seen in,

home.interserv.com/~mniemiec/lifepage.htm

⋆ The first author is supported by the REPLI project TIC-2002-04470-C03.

Maximum density stable patterns (also called *still lifes*) are board configurations with a maximal number of living cells which do not change along time. They can be seen as an academic simplification of a standard issue in discrete dynamic systems. [5] has shown that for the infinite board the maximum density is $1/2$. In this paper we are concerned with finite patterns. In particular, we consider $n \times n$ still lifes, for which no polynomial method is known. This problem has been recently included in the *CSPlib*[1] repository of challenging constraint satisfaction problems.

In [3] still life is solved using *integer programming* and *constraint programming*, both of them being search-based methods. Their best results were obtained with a hybrid approach which combines the two techniques and exploits the problem symmetries to reduce the search space. With their algorithm, they solved the $n = 15$ case in about 8 days of *cpu* with a modern computer. Another interesting work can be found in [11] where pure constraint programming techniques are used, and the problem is solved in its dual form. Although not explicitly mentioned, these two works use algorithms with worst-case time complexity $O(2^{(n^2)})$ and polynomial space.

In this paper we find still lifes using *dynamic programming*. We model the problem as a *weighted constraint satisfaction problem* (WCSP) [10, 2] and solve it with *bucket elimination* (BE) [4]. BE is a generic algorithm suitable for many automated reasoning and optimization problems. It is often overlooked due to its exponential space complexity. Here we show that for the still life problem it is highly competitive. In the theoretical side, we show that its time complexity is $\Theta(n^2 \times 2^{3n})$, which means an exponential improvement over search-based methods. Regarding space, the complexity is $\Theta(n \times 2^{2n})$. In the practical side we show that plain BE is much faster than basic search algorithms and comparable to sophisticated search methods. Our implementation of BE solves the $n = 14$ case in less than 30 hours. The $n = 15$ case cannot be solved with our computer due to space exhaustion. A nice feature of BE is that it can compute, with no extra cost, the number of optimal solutions. Thus, we report, for the first time, the number of still lifes up to $n = 14$.

An additional contribution of this paper is that we have adapted BE to exploit some of the problem symmetries, with which the speed is nearly doubled and the space requirement is halved (the $n = 14$ case is solved in about 15 hours, but we still could not solve the $n = 15$ case).

When n is too large to solve optimally with current methods, some authors [3, 11] find symmetric optimal solutions. We have also adapted BE to solve the problem subject to a vertical reflection symmetry and have solved the $n = 28$ case for the first time.

Although the space complexity seems to be a critical limitation of our method, it is not necessarily so. There are ways to trade space by time within the BE algorithm (see [7–9]), which give room to our approach to scale up and make it very promising. We discuss this in detail in Section 6.

[1] `www.csplib.org`

The structure of this paper is as follows: In Section 2 we give preliminary definitions. In Section 3 we show how the still life problem is modelled as a WCSP and solved with BE. In Section 4 we adapt BE to exploit problem symmetries. In Section 5 we modify BE to find symmetrical solutions. In Section 6 we highlight our ongoing work. Finally, Section 7 summarizes the conclusions of our work.

2 Preliminaries

A *Constraint satisfaction problem* (CSP) [12] is defined by a tuple (X, D, C), where $X = \{x_1, \dots, x_n\}$ is a set of *variables* taking values from their finite *domains* ($D_i \in D$ is the domain of x_i). C is a set of *constraints*, which prohibit the assignment of some combinations of values. A constraint $c \in C$ is a *relation* over a subset of variables $var(c)$, called its *scope*. For each assignment t of all variables in $var(c)$, $t \in c$ iff t is allowed by the constraint. A *solution* to the CSP is an complete assignment that satisfies every constraint. Constraints can be given explicitly as tables of permitted tuples, or implicitly as mathematical expressions or computing procedures.

Weighted constraint satisfaction problems (WCSP) [2] and [10] augment the CSP model by letting the user express preferences among solutions. In WCSP, constraints are replaced by cost functions (also called *soft constraints*). Forbidden assignments receive cost ∞. Permitted assignments receive finite costs that express their degree of preference. The *valuation* of an assignment t is the sum of costs of all functions whose scope is assigned by t. A *solution* to the WCSP is a complete assignment with a finite valuation. The task of interest is to *find the solution with the lowest valuation.*

A WCSP instance is graphically depicted by means of its *interaction* or *constraint graph*, which has one node per variable and one edge connecting any two nodes whose variables appear in the same scope of some cost function.

Bucket elimination (BE) [4, 1] is a generic algorithm that can be used for WCSP solving. It is based upon two operators over functions. For the WCSP case they are:

- The *sum* of two functions f and g denoted $(f + g)$ is a new function with scope $var(f) \cup var(g)$ which returns for each tuple the sum of costs of f and g,
$$(f + g)(t) = f(t) + g(t)$$
- The *elimination* of variable x_i from f, denoted $f \Downarrow i$, is a new function with scope $var(f) - \{x_i\}$ which returns for each tuple t the minimum cost extension of t to x_i,
$$(f \Downarrow i)(t) = \min_{a \in D_i} \{f(t \cdot (x_i, a))\}$$
where $t \cdot (x_i, a)$ means the extension of t to the assignment of a to x_i. Observe that when f is a unary function (*i.e.*, arity one), eliminating the only variable in its scope produces a constant.

Example 1 *Let $f(x_1, x_2) = x_1 + x_2$ and $g(x_1, x_3) = x_1 x_3$. The sum of f and g is $(f + g)(x_1, x_2, x_3) = x_1 + x_2 + x_1 x_3$. If domains are integers in the interval $[1..10]$, the elimination of x_1 from f is $(f \Downarrow 1))(x_2) = 1 + x_2$. The subsequent elimination of x_2, produces constant 2 (i.e, $((f \Downarrow 1) \Downarrow 2) = 2$).*

In the previous example, resulting functions were expressed intensionally for clarity reasons. Unfortunatelly, in general, the result of summing functions or eliminating variables cannot be expressed intensionally by algebraic expressions. Therefore, BE collects intermediate results extensionally in tables, which causes its high space complexity.

BE (Figure 1) uses an arbitrary variable ordering o that we assume, without loss of generality, lexicographical (i.e, $o = (x_1, x_2, \ldots, x_n)$). BE works in two phases. In the first phase (lines 1-5), the algorithm eliminates variables one by one, from last to first, according to o. In the second phase, the optimal assignment is computed processing variables from first to last. The elimination of variable x_i is done as follows: C is the set of current constraints. The algorithm stores the so called *bucket* of x_i, noted B_i, which contains all cost functions in C having x_i in their scope (Line 2). Next, BE computes a new function g_i by summing all functions in B_i and subsequently eliminating x_i (line 3). Then, C is updated by removing the functions in B_i and adding g_i (line 4). The new C does not contain x_i (all functions mentioning x_i were removed) but preserves the value of the optimal cost. The elimination of the last variable produces an empty-scope function (*i.e.*, a constant) which is the optimal cost of the problem. The second phase (lines 6-10) generates an optimal assignment of variables. It uses the set of buckets that were computed in the first phase. Starting from an empty assignment t (line 6), variables are assigned from first to last according to o. The optimal value for x_i is the best value regarding the extension of t with respect to the sum of functions in B_i (lines 8,9). We use the non standard notation $\text{argmin}_a\{f(a)\}$ to denote the value a producing minimum $f(a)$.

BE can also compute the number of optimal solutions with not additional overhead. More than that, *all* optimal solutions can be easily retrieved from the buckets computed during the process (see [4] for details).

The complexity of BE depends on the problem structure, as captured by its constraint graph G, and the ordering o. The *induced graph* of G relative to o, noted $G^*(o)$, is obtained by processing the nodes in reverse order of o. When considering node i, new edges are added in order to form a clique with all its adjacent nodes, appearing before i in the ordering o. Given a graph and an ordering of its nodes, the *width* of a node is the number of edges connecting it to nodes lower in the ordering. The *induced width of a graph* along ordering o, denoted $w^*(o)$, is the maximum width of nodes in the induced graph.

Theorem 1 *[4] The complexity of* BE *along ordering o is time $O(Q \times n \times d^{w^*(o)+1})$ and space $O(n \times d^{w^*(o)})$, where d is the largest domain size and Q is the cost of evaluating cost functions (usually assumed $O(1)$).*

function BE(X, D, C)

for $i = n$ **downto** 1 **do**

$\quad B_i := \{f \in C |\ x_i \in var(f)\}$

$\quad g_i := (\sum_{f \in B_i} f) \Downarrow i;$

$\quad C := (C \cup \{g_i\}) - B_i;$

endfor

$t := \emptyset;$

for $i = 1$ **to** n **do**

$\quad v := \text{argmin}_{a \in D_i}\{(\sum_{f \in B_i} f)(t \cdot (x_i, a))\}$

$\quad t := t \cdot (x_i, v);$

endfor

return$(C, t);$

endfunction

Fig. 1. Bucket Elimination. (X, D, C) is the WCSP instance to be solved. The algorithm returns the optimal cost in C and one optimal assignment in t.

3 Finding still lifes with BE

3.1 Modelling still life as a WCSP

The *still life* problem consist of finding a $n \times n$ stable pattern of maximum density in the game of life, where all cells outside the pattern are assumed to be dead. Considering the rules of the game, it is clear that in stable patterns all *living cells must have exactly two or three living neighbors* in order to remain alive, and *dead cells must not have three living neighbors* in order to remain dead. Besides, *boundary rows and columns must not have more than two adjacent living cells*, since three consecutive cells would produce a new living cells outside the $n \times n$ region. Figure 2 (left) shows a 3×3 still life.

Still life can be easily modelled as a WCSP. We use a compact formulation with n variables, one for every row. Variable x_i is associated to the i-th row. Its domain D_i is the set of sequences of n bits. The j-th bit of value a, noted a_j, indicates the state of the j-th cell of the row. If a_j takes value 1 the corresponding cell is alive, else it is dead. Let a, b and c be domain values. We define $Z(a)$ as the number of zeroes in a. $S(a, b, c)$ is a boolean predicate satisfied iff all cells of b are stable cells being a the row above b and c the row below b ($S(a, b, c)$ is false if there is some unstable cell in b).

The problem has n cost functions f_i (with $i = 1, .., n$). For $i = 2, .., n-1$, f_i is ternary, with scope $var(f_i) = \{x_{i-1}, x_i, x_{i+1}\}$. If the arguments represent an unstable configuration it returns ∞, else it returns the number of zeroes in the middle row. Formally,

$$f_i(a, b, c) = \begin{cases} \infty & : \ \neg S(a, b, c) \\ \infty & : \ a_1 = b_1 = c_1 = 1 \\ \infty & : \ a_n = b_n = c_n = 1 \\ Z(b) & : \ \text{otherwise} \end{cases}$$

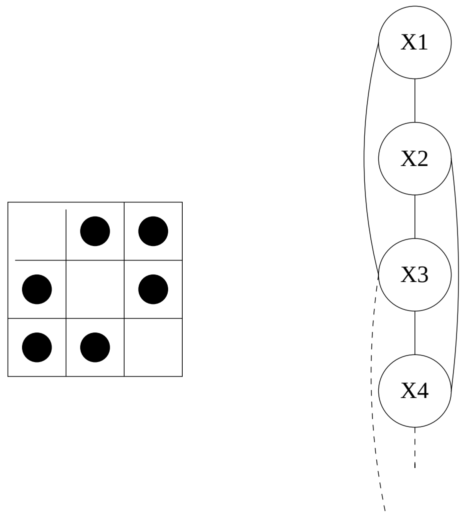

Fig. 2. *Left:* A 3×3 still life pattern. *Right:* Constraint graph of still life.

Functions f_1 and f_n are binary. They are equivalent to the ternary cost functions, but assuming dead cells above the top row and below the bottom row, respectively. The scope of f_1 is $\{x_1, x_2\}$ and it is defined as,

$$f_1(b, c) = \begin{cases} \infty & : \quad \neg S(\mathbf{0}, b, c) \\ Z(b) & : \quad \text{otherwise} \end{cases}$$

where $\mathbf{0}$ denotes the *all zeroes* string of bits. Similarly, the scope of f_n is $\{x_{n-1}, x_n\}$ and it is defined as,

$$f_n(a, b) = \begin{cases} \infty & : \quad \neg S(a, b, \mathbf{0}) \\ Z(b) & : \quad \text{otherwise} \end{cases}$$

Note that computing $f_i(a, b, c)$, $f_1(b, c)$ and $f_n(a, b)$ is $\Theta(n)$.

3.2 BE for still life

The constraint graph of our still life formulation is a sequence of size 3 cliques (Figure 2, right). The induced graph $G^*(o)$ with $o = (x_1, x_2, \ldots, x_n)$ does not have new edges (i.e, $G^*(o) = G$). Consequently, the induced width is $w^*(o) = 2$. Since domains have size 2^n, by Theorem 1, the complexity of BE is time $O(n^2 \times 2^{3n})$ and space $O(n \times 2^{2n})$.

The sequential structure of the constriant graph makes the implementation of BE very simple (see Figure 3). Sequences of bits of size n are represented by integers in the interval $[0..2^n - 1]$. In the first phase, we process variables from last to first. Buckets are implicitly computed. The bucket of x_n is $B_n = \{f_n, f_{n-1}\}$ (these are the only cost function having x_n in their scope). B_n is used

function $\mathtt{BE}(n)$
1. **for** $a, b \in [0..2^n - 1]$ **do**
2. $\quad g_n(a, b) := \mathtt{min}_{c \in [0..2^n-1]}\{f_{n-1}(a, b, c) + f_n(b, c)\}$;
3. **endfor**
4. **for** $i = n - 1$ **downto** 3 **do**
5. $\quad$ **for** $a, b \in [0..2^n - 1]$ **do**
6. $\qquad g_i(a, b) := \mathtt{min}_{c \in [0..2^n-1]}\{f_{i-1}(a, b, c) + g_{i+1}(b, c)\}$;
7. $\quad$ **endfor**
8. **endfor**
9. $(x_1, x_2) := \mathtt{argmin}_{a,b \in [0..2^n-1]}\{g_3(a, b) + f_1(a, b)\}$;
10. $opt := g_3(x_1, x_2) + f_1(x_1, x_2)$;
11. **for** $i = 3$ **to** $n - 1$ **do**
12. $\quad x_i := \mathtt{argmin}_{c \in [0..2^n-1]}\{f_{i-1}(x_{i-2}, x_{i-1}, c) + g_{i+1}(x_{i-1}, c)\}$;
13. **endfor**
14. $x_n := \mathtt{argmin}_{c \in [0..2^n-1]}\{f_{n-1}(x_{n-2}, x_{n-1}, c) + f_n(x_{n-1}, c)\}$;
15. **return**$(opt, (x_1, x_2, \ldots, x_n))$;

endfunction

Fig. 3. Bucket Elimination for the still life problem. The algorithm returns the optimal value in *opt* and the optimal assignment in $(x_1, x_2, \ldots, x_n)$

to compute a new binary cost function g_n with scope $\{x_{n-2}, x_{n-1}\}$ (lines 1-3). By construction, $g_n(a, b)$ is the cost of the best extention of $(x_{n-2} = a,\ x_{n-1} = b)$ to the eliminated variable x_n. The bucket of x_{n-1} is $B_{n-1} = \{g_n, f_{n-2}\}$. It is used to compute g_{n-1} with scope $\{x_{n-3}, x_{n-2}\}$ (lines 5-7, first iteration). $g_{n-1}(a, b)$ is the cost of the best extension of $(x_{n-3} = a,\ x_{n-2} = b)$ to the eliminated variables x_{n-1} and x_n. Subsequent iterations of the loop eliminate subsequent variables. In the last iteration variable x_3 is eliminated. When the algorithm reaches line 9, the current problem contains two cost functions: g_3, which contains the optimal extensions of each potential assignment of x_1 and x_2 to the rest of variables, and f_1. Instead of continuing the elimination of variables, we found it to be more efficient to solve the current problem with a brute-force exhaustive search (line 9). Variables x_1 and x_2 are assigned with their optimal values (line 9) and the optimal cost is assigned to *opt* (line 10).

In the second phase (lines 11-14), we process variables from first to last. We assign to each variable the best value according to its bucket and previously assigned variables.

It is easy to verify the complexity of the algorithm. Regarding space it is $\Theta(n \times 2^{2n})$, due to the space required to store functions g_i extensionally, which have 2^{2n} entries each. Regarding time, the critical part of the algorithm is the execution of lines 4-8. Line 6 has complexity $\Theta(n \times 2^n)$ (finding the minimum of 2^n alternatives, the computation of each one being $\Theta(n)$). It has to be executed $\Theta(n \times 2^{2n})$ times, which makes a global complexity of $\Theta(n^2 \times 2^{3n})$. Observe that the complexity of BE in the still life problem is an exponential improvement over search algorithms.

There is a simple average-case time optimization that we found very effective. Observe that lines 2 and 6 require the evaluation of $f_i(a,b,c)$ with a and b fixed and varying c. All values of c such that $f_i(a,b,c) = \infty$ are irrelevant because they cannot provide the minimum valuation. Let u_{ab} be the smallest value such that $f_i(a,b,u_{ab}) \neq \infty$. Clearly line 6 (similarly line 2) can be replaced by:

$$g_i(a,b) := \mathtt{min}_{c\in[u_{ab}..2^n-1]}\{f_{i-1}(a,b,c) + g_{i+1}(b,c)\};$$

which in many cases reduces the interval size drastically. Since all f_i in the original problem are essentially equal (the only difference is their scope) value u_{ab} is common to all f_i (with $i = 2..n-1$). For each a, b, we compute u_{ab} during a pre-process and store it in a table that is used to speed up every variable elimination. Note that this table has 2^{2n}. Thus, it does not affect the space complexity of the algorithm.

3.3 Experimental Results

n	cost	n. sol.	BE	CP	IP	CP/IP-sym
5	16	1	0	0	1	0
6	18	48	0	1	23	0
7	28	2	0	10	7	0
8	36	1	1	189	65	2
9	43	76	4	> 1500	> 1500	51
10	54	3590	27	*	*	147
11	64	73	210	*	*	373
12	76	129126	1638	*	*	30360
13	90	1682	13788	*	*	30729
14	104	11	10^5	*	*	5×10^5
15	119	?	*	*	*	7×10^5

Fig. 4. Experimental results of four different algorithms on the still life problem. Times are in seconds.

Table 4 reports the results that we obtained with a 1 Ghz Pentium III machine with 1 Gb of memory. From left to right, the first three columns report: problem size, solution cost (as the number of living cells) and number of optimal solutions (most of them have never been reported before). We count as different two solutions even if one can be transformed to the other through a problem symmetry. The fourth column reports the CPU time of our executions (BE) in seconds. For comparison purposes, the fifth, sixth and seventh columns show times obtained in [3] with basic constraint programming (CP), integer programming (IP), and a sophisticated hybrid algorithm (CP/IP-sym) which exploits the

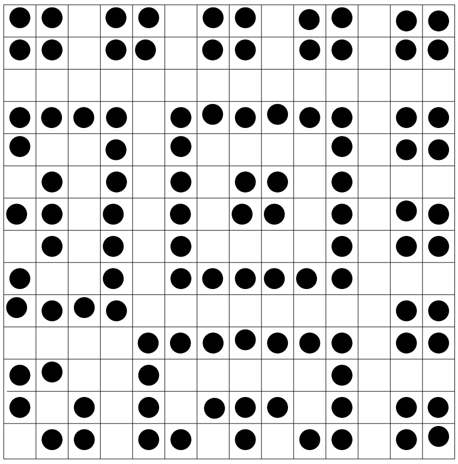

Fig. 5. A 14×14 still life pattern.

problem symmetries (see Section 4). In their experiments, they used a 650 Mhz Pentium III with 196 Mb of memory. Time comparison should be done with caution, because machines are different. Note as well that times in [3] were obtained using a commercial solver, while our times have been obtained with our *ad-hoc* implementation. On the one side, our implementation was made specifically for the still life problem, which has the advantage of optimizing the use of space and specializing some parts of the code. On the other side, our implementation is a prototype, inplemented in a few weeks, which is in disadvantage with respect to commercial solvers, developed during months or years. Having said that, it can be observed that BE clearly outperforms basic CP and IP by orders of magnitude. While CP and IP algorithms cannot solve the problem beyond $n = 8$ in less than half an hour, BE can solve the $n = 12$ case subject to the same time limit. The $n = 14$ case is the largest instance that we could solve due to space exhaustion (see Figure 5). As a matter of fact, the original code could not be executed for the $n = 14$ case. We solved it by disabling the *counting solutions* feature which deallocates some memory. We computed the number of solutions in a different execution with a slower machine with more memory space. Comparing BE with the CP/IP hybrid we observe that both algorithms give very similar times (BE is faster, but within the same order of magnitude). Given the simplicity of the BE algorithm we consider it a very satisfactory result. An additional observation is that BE scales up very regularly, each execution requiring roughly eight times more time and four times more space than the previous, which is in clear accordance with the algorithm complexity.

4 Exploiting problem symmetries

Still life is a highly symmetric problem. For any stable pattern, it is possible to create an equivalent pattern by: (i) rotating the board by 90, 180 or 270 degrees, (ii) reflecting the board horizontally, vertically or along one diagonal or (iii) doing any combination of rotations and reflections. Search methods proposed in [3] and [11] exploit that fact by cutting off some search paths that only contain solutions that are symmetric of previously processed ones.

In the following we show how BE can also be adapted to take advantage of some of the symmetries.

Let's assume that n is an even number (the odd case is similar). Consider the algorithm of Figure 3 and assume that we stop the execution after the elimination of variable $x_{\frac{n}{2}+2}$. The elimination of $x_{\frac{n}{2}+2}$ produces $g_{\frac{n}{2}+2}$, with scope $\{x_{\frac{n}{2}}, x_{\frac{n}{2}+1}\}$. At this point supose that we change the order of elimination of the remaining variables to $x_1, x_2, \ldots, x_{\frac{n}{2}-1}$. The elimination x_1 produces a new function g_1 with scope $\{x_2, x_3\}$. Due to the 180 rotation symmetry it is the same to eliminate x_1 or rotate the board by 180 degrees and eliminate x_n. Therefore, for all a and b it holds that

$$g_1(a,b) = g_n(\bar{b}, \bar{a})$$

Where $\bar{a}$ (respectively, $\bar{b}$) is the reflection of value a (respectively, b). In addition, due to the vertical reflection symmetry we have that,

$$g_n(\bar{b}, \bar{a}) = g_n(b, a)$$

Therefore, it follows that,

$$g_1(a,b) = g_n(b,a)$$

In general, the elimination of variable x_i (with $1 \le i \le \frac{n}{2} - 1$) produces a new function g_i with scope $\{x_{i+1}, x_{i+2}\}$. Due to the problem symmetries, we have that,

$$g_i(a,b) = g_{n-i+1}(b,a)$$

Therefore, variables $x_1, x_2, \ldots, x_{\frac{n}{2}-1}$ do not have to be eliminated, because the effect of the elimination can be *inferred.* At this point, the current problem contains only two variables ($x_{\frac{n}{2}}$ and $x_{\frac{n}{2}+1}$) and one cost function between them ($g_{\frac{n}{2}+1}(x_{\frac{n}{2}}, x_{\frac{n}{2}+1}) + g_{\frac{n}{2}+1}(x_{\frac{n}{2}+1}, x_{\frac{n}{2}})$). This problem can be solved by exhaustive exploration. It is clear that the savings from avoiding the elimination of half of the variables reduces the time and space requirements to one half.

The previous idea is illustrated by Algorithm BE-sym (Figure 6). In lines 1-6 the elimination of $x_n, x_{n-1}, \ldots, x_{\frac{n}{2}+2}$ is performed as in BE. In line 7, the optimal cost is computed where $g_{\frac{n}{2}+1}(x_{\frac{n}{2}}, x_{\frac{n}{2}+1})$ provides the effect of the performed elimination of $x_n, x_{n-1}, \ldots, x_{\frac{n}{2}+2}$ and the inferred elimination of $x_1, x_2, \ldots, x_{\frac{n}{2}-1}$. In line 8 the optimal assignment of $x_{\frac{n}{2}}$ and $x_{\frac{n}{2}+1}$ is computed. Lines 9-12 compute the optimal assignment of $x_n, x_{n-1}, \ldots, x_{\frac{n}{2}+2}$ as in the BE algorithm. Lines 13-16 compute the optimal assignment of $x_1, x_2, \ldots, x_{\frac{n}{2}-1}$. The optimal assignment of x_i without exploiting the simmetries would be,

$$x_i := \mathtt{argmin}_{c\in[0..2^n-1]}\{f_{i+1}(c, x_{i+1}, x_{i+2}) + g_{i-1}(c, x_{i+1})\}$$

however, since $g_{i-1}(a, b) = g_{n-i}(b, a)$, it can be computed as,

$$x_i := \mathtt{argmin}_{c\in[0..2^n-1]}\{f_{i+1}(c, x_{i+1}, x_{i+2}) + g_{n-i}(x_{i+1}, c)\}$$

function BE-sym(n)
1. **for** $a, b \in [0..2^n - 1]$ **do**
2. $\quad g_n(a, b) := \mathtt{min}_{c\in[0..2^n-1]}\{f_{n-1}(a, b, c) + f_n(b, c)\}$;
3. **for** $i = n - 1$ **downto** $n/2 + 2$ **do**
4. $\quad$ **for** $a, b \in [0..2^n - 1]$ **do**
5. $\quad\quad g_i(a, b) := \mathtt{min}_{c\in[0..2^n-1]}\{f_{i-1}(a, b, c) + g_{i+1}(b, c)\}$;
6. **endfor**
7. $opt := \mathtt{min}_{a,b\in[0..2^n-1]}\{g_{\frac{n}{2}+2}(a, b) + g_{\frac{n}{2}+2}(b, a)\}$;
8. $(x_{\frac{n}{2}}, x_{\frac{n}{2}+1}) := \mathtt{argmin}_{a,b\in[0..2^n-1]}\{g_{\frac{n}{2}+2}(a, b) + g_{\frac{n}{2}+2}(b, a)\}$;
9. **for** $i = \frac{n}{2} + 2$ **to** $n - 1$ **do**
10. $\quad x_i := \mathtt{argmin}_{c\in[0..2^n-1]}\{f_{i-1}(x_{i-2}, x_{i-1}, c) + g_{i+1}(x_{i-1}, c)\}$;
11. **endfor**
12. $x_n := \mathtt{argmin}_{c\in[0..2^n-1]}\{f_{n-1}(x_{n-2}, x_{n-1}, c) + f_n(x_{n-1}, c)\}$;
13. **for** $i = \frac{n}{2} - 1$ **to** 2 **do**
14. $\quad x_i := \mathtt{argmin}_{c\in[0..2^n-1]}\{f_{i+1}(c, x_{i+1}, x_{i+2}) + g_{n-i}(x_{i+1}, c)\}$;
15. **endfor**
16. $x_1 := \mathtt{argmin}_{c\in[0..2^n-1]}\{f_2(c, x_2, x_3) + f_1(c, x_2)\}$;
17. **return**(opt, $(x_1, x_2, \ldots, x_n)$);

endfunction

Fig. 6. Bucket Elimination exploiting symmetries (assume n even).

Table 7 reports the results obtained with BE-sym. The first column tells the size of the problem. The second column indicates times obtained with BE-sym. To facilitate comparison, the third column reports results obtained with BE and the fourth column reports the best times obtained by [3] with their hybrid CP/IP algorithm which also exploits symmetries (again, be aware of the different machines). Comparing BE *vs.* BE-sym, the experiments confirm that BE-sym is twice as fast as BE. Although BE-sym requires less memory than BE, we still could not execute the $n = 15$ case. Comparing it with the CP/IP hybrid, it can be observed that BE-sym seems to be systematically faster.

5 Restricting to symmetric still life

When n is too large to solve optimally with current methods, previous authors proposed finding symmetric optimal solutions. In [3] optimal horizontally sym-

n	BE-sym	BE	CP/IP-sym
9	2	4	51
10	14	27	147
11	120	210	373
12	813	1638	30360
13	7223	13788	30729
14	6×10^4	10^5	5×10^5
15	*	*	7×10^5

Fig. 7. Experimetal results of three algorithms on the still life problem.

metric solutions for $n = 18$ are found, and in [11] optimal 90 degrees rotational symmetric solutions for $n = 18$ are also found.

We followed the same approach and adapted BE to consider vertically symmetric patterns. With our formulation, changes are straightforward: we only need to reduce domains to symmetrical values. Lets assume that n is an even number (the odd case is similar). We represent symmetric sequences of bits of length n by considering the left side of the sequence (clearly, the symmetrical right part can be obtained by reversing the left part), which can be implemented as integers in the interval $[0..2^{\frac{n}{2}} - 1]$. It is easy to see that the complexity of BE is now time $\Theta(n^2 \times 2^{3n/2})$ and space $\Theta(n \times 2^n)$, which means that the size of problems that we can solve should be doubled. Observe that this problem has exactly the same symmetries as the original problem. Consequently, we can still use the BE-sym algorithm.

Figure 9 reports the results that we obtained with BE-sym. The first column contains the problem size (we only solved even values of n), the second column reports the optimal value as number of living cells, the third column reports the number of solutions and the fourth column reports CPU time obtained with the BE-sym algorithm. As predicted, we solve up to the $n = 28$ case (Figure 8). The $n = 30$ case could not be execute due to space exhaustion. These results improve significatively over the previous works of [3, 11].

6 Future Work

We have shown that BE provides an efficient solver approach to the still life problem, although it has the fundamental limitation of its exponential space complexity which makes impossible with current computers to solve the problem beyond $n = 14$. Fortunately, some authors [7–9] suggest ways to overcome space exhaustion when executing BE. These approaches propose parameterized algorithms, where the parameter indicates the amount of space the user is willing to use. The algorithms dynamically switch to search each time BE cannot carry out the solving process. BE is resumed as soon as the space-costly part of the problem has been solved. We are currently exploring these ideas. Hopefully we will be reporting new results in the near future.

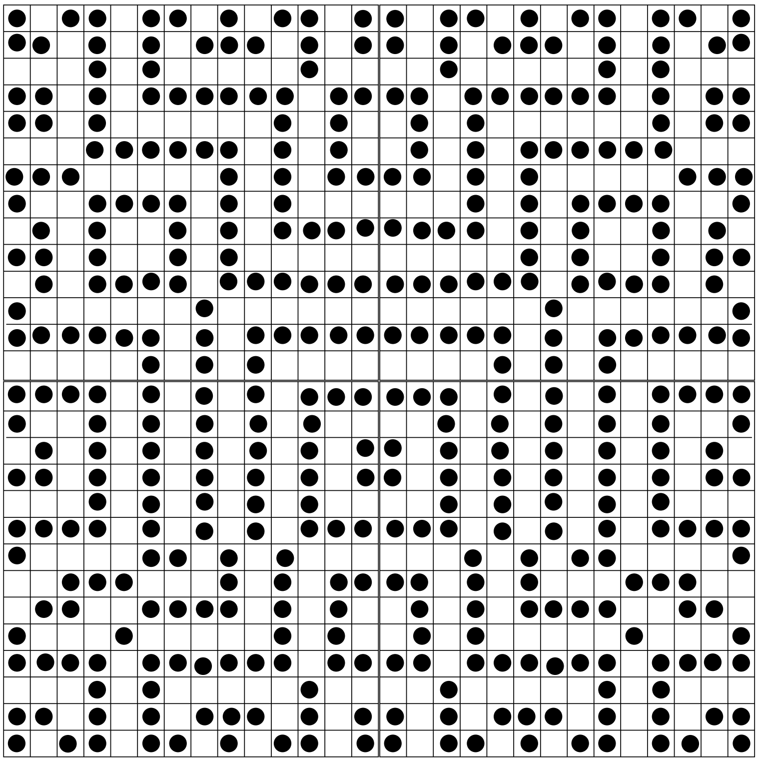

Fig. 8. A 28×28 symmetric still life. The optimal value is 406 living cells.

7 Conclusion

Bucket Elimination is often believed to be an algorithm of little practical interest due to its exponential space complexity. In this paper we showed that it is extremely competitive for the still life problem. We showed that it provides a much lower worst-case time complexity than search-based methods which makes it systematically faster in practice. The space complexity drawback comes to the fore where search methods fail due to their exponential time complexity. We reported some results, which we think are new: the number of optimal solutions up to $n = 14$ and the optimal cost and the number of solutions of vertically symmetric still lifes up to $n = 28$.

As far as we know, there is no previous work on how to adapt BE to exploit symmetries. We enhanced the performance of our BE implementation by considering some of the problem symmetries. We belive that it is a preliminary step towards a wider (although possibly limited) practical applicability of BE.

n	opt. cost	n. sol.	BE-sym
10	52	133	0
12	76	8	0
14	104	1	0
16	136	3	0
18	170	4	10
20	208	1813	81
22	252	635	633
24	300	5363	4620
26	350	55246	37600
28	406	12718	1.7×10^5

Fig. 9. Experimental results on for finding vertical reflection symmetric still lifes with BE.

References

1. Bertele, U., Brioschi, F.: Nonserial Dynamic Programming. Academic Press, London, 1972.
2. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-Based Constraint Satisfaction and Optimization. Journal of the ACM. **44(2)** (1997) 201–236.
3. Bosch, R., Trick, M.: Constraint programming and hybrid formulations for three life designs. Proceedings of the International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CP-AI-OR'02 (2002), 77–91.
4. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence. **113** (1999) 41–85.
5. Elkies, N.D.: The still-life density problem and its generalisations. Voronoi's impact on modern science, Book 1 (1998) 228–253. Institute of Math. Kyiv.
6. Gardner, M.: The fantastic combinations of John Conway's new solitary game. Scientific American. **223** (1970) 120–123.
7. Larrosa, J., Dechter, R.: Boosting Search with Variable Elimination. In Proceedings of Principles and Practice of Constraint Programming, CP-2000 (Singapore, Singapore, 2000), 291–305.
8. Larrosa, J.: Boosting Search with Variable Elimination in Constraint Optimization and Constraint Satisfaction Problems. Constraints: an International Journal. To appear.
9. Dechter, R., El Fattah, Y.: Topological Parameters for Time-Space Tradeoff. Artificial Intelligence. To appear.
10. Schiex, T., Fargier, H., Verfaillie, G.: Valued Constraint Satisfaction Problems: hard and easy problems. In Proceedings of the 14th. International Join Conference on Artificial Intelligence, IJCAI-1995 (Montreal, Canada, 1995), 631–637.
11. Smith, B.: A dual graph translation of a problem in life. In Proceedings of Principles and Practice of Constraint Programming, CP-2002 (Ithaca, USA, 2002).
12. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press, London, 1993.