On the Practical use of Variable Elimination in Constraint Optimization Problems: 'Still-life' as a Case Study

Javier Larrosa (larrosa@lsi.upc.es)
Enric Morancho (enricm@ac.upc.es)
David Niso (f3068211@est.fib.upc.es)
Universitat Politecnica de Catalunya
Jordi Girona 1-3, 08034 Barcelona, Spain

January 31, 2005

Abstract

Variable elimination is a general technique for *constraint processing*. Although its high space complexity limits its use as a stand-alone algorithm, it can be combined with other techniques leading to efficient algorithms. In this paper, we study the applicability of variable elimination to the challenging problem of finding *maximum density still-lifes*. We show that this technique is best-known option both theoretically and empirically. In our experiments we have been able to solve the $n = 20$ instance, which is far beyond reach with alternative approaches.

1 Introduction

Many problems of practical interest are NP-hard. Therefore, all solving algorithms are likely to require exponential time in the worst case. However, this negative result does not prevent the existence of algorithms that can solve some real instances of interest in reasonable time. This possibility has motivated the work of many researchers in the last decades, their goal being the development of algorithms with good average efficiency. Many works in the *constraint satisfaction* framework fall within this category.

Assessing progress of this kind of research is a difficult task. Often, the only possible evaluation of new techniques is empirical, which relies on the existence of benchmarks. We distinguish three categories of benchmarks,

depending on the randomness of their instances: *i*) *Random models* allow the generation of random instances where some parameters can be specified. Such problems are totally unstructured. Some examples are the *four parameters random CSP model* (Smith & Dyer, 1996) and the *random k-SAT model* (Mitchell, Selman, & Levesque, 1992). *ii*) *Random instances of structured problems* are structured problems where some details are left unspecified. Thus, ensembles of instances can be generated by randomly determining the unspecified parts. Some examples are *random combinatorial auctions* (Sandholm, 1999), *random probabilistic decoding problems* (Dechter & Rish, 2003) and *random quasi-group completion problems* (Gomes & Shmoys, 2002). *iii*) *Fully structured problems* are totally specified problems. We distinguish two types: instances taken from real applications (e.g. *frequency assignment problems* (Cabon, de Givry, Lobjois, Schiex, & Warners, 1999), *earth observation satellite management* (Bensana, Lemaitre, & Verfaillie, 1999)) and academic problems (e.g. the *social golfer* problem (?), the *progressive party* problem (Smith, Brailsford, Hubbard, & Williams, 1996), *rectangle packing* (Korf, 2003)). Unlike most industrial problems, academic problems have the advantage of being easy to state, which allows the researcher to fully understand the problem with limited effort. Some academic problems are simplifications of problems with practical interest.

Since testing algorithms in each category has advantages and disadvantages, a good evaluation should include problems across the three categories. Nevertheless, experimenting with fully structured problems is more delicate and much more time consuming. The main reason is that general techniques cannot always be directly applied. Often times they need to be adapted to the problem before being competitive. As a consequence, when a new algorithm is proposed, most researchers only evaluate it on fully random problems and/or random instances of structured problems. This is the case of algorithms based on *variable elimination*.

In this paper we consider the problem of finding *maximum density still-life patterns* in the game of *life* which is a challenging fully structured problem, and we study the applicability of variable elimination. First we apply *bucket elimination* (BE) (Dechter, 1999; Bertele & Brioschi, 1972), which is the algorithm that just eliminates variables one after another, until the problem can be trivially solved. Against what could be expected, we observe that BE is competitive with state-of-the-art alternatives. Next, we introduce a more sophisticated algorithm that combines search and variable elimination (following the ideas of (Larrosa & Dechter, 2003)). Further, it uses a lower bound based on mini-buckets (following the ideas of (Kask & Dechter, 2001)). Thus, our work explores the synergy of these techniques

and complements previous experimental evaluation of them. With our algorithm, we solve in one minute the $n = 15$ instance, which was the largest instance ever solved (it required nearly 7 days of CPU with a sophisticated algorithm). We have been able to solve up to the $n = 20$ instance, which was far beyond reach with previous techniques.

Our work also complements previous algorithmic work on this particular problem. It had been tackled with *integer programming* and *constraint programming* techniques, both of them using *search*-based algorithms. Since variable elimination is based on *dynamic programming*, it provides a radically new approach to the problem.

It is often believed that variable elimination has very little practical interest due to its high space complexity. In this paper we provide some counter-evidence. For the still-life problem BE is competitive with sophisticated search-based techniques. Besides, when the space complexity of BE becomes a problem, we can still use some space-efficient variations, which prove very effective.

The structure of the paper is the following: In the next Subsection we define the game of life. In Section 2 we give some preliminary definitions. In Section 3 we formulate the problem of the maximal stable life as a *weighted constraint satisfaction problem*. In Section 4 we study the applicability of BE to the problem. In Section 5 we introduce the hybrid algorithm with which we obtained the results reported in Section 6. Finally, Section 7 gives some conclusions and lines of future work.

1.1 Life and Still-Life

The game of *life* was invented in the late 60s by John Horton Conway and was later popularized by Martin Gardner (Gardner, 1970). The game is played over an infinite checkerboard, where each square is called a *cell*. Each cell has eight *neighbors*: the eight cells that share one or two corners with it. The only player places checkers on some cells. If there is a checker on it, the cell is *alive*, else it is *dead*. The state of the board evolves iteratively according to the following 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.

As a simple example, consider the leftmost configuration in Figure 1 at a time instant t (assume dead cells outside the drawn region). In the next time iteration (i.e, $t+1$) the pattern evolves to the configuration on its right.

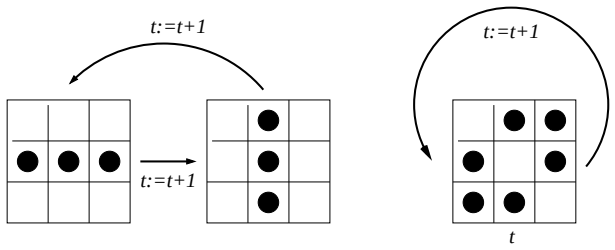

Figure 1: *Left:* A 2-oscillator. *Right:* A 3×3 still-life.

In the next time iteration the board returns to the initial configuration. Therefore, this pattern is a 2-period cycle, also called a *2-oscillator*.

Although defined in terms of extremely simple rules, the game of life has proven mathematically rich and it has attracted the interest of both mathematicians and computer scientists. Recently, it has been used as a source of inspiration for combinatorial optimization problems that have been used as a test bed for different algorithmic approaches (Bosch & Trick, 2002). In this paper, we will consider the problem of finding *maximum density stable patterns* (also called *maximum density still-lifes*, or just *still-lifes*) which are board configurations with a maximum number of living cells which do not change over time (Figure 1 (right) shows a 3x3 still-life). This problem has been recently included in the *CSPlib* repository (?) and a dedicated web page[1] has been set to maintain up-to-date results.

In (Bosch & Trick, 2002), Bosch and Trick solved the still-life problem using two different approaches: *integer programming* and *constraint programming*, both of them being based on search. They observed that none of them could solve up to the $n = 8$ problem within reasonable time. Their best results were obtained with a hybrid approach which combines the two techniques and exploits the problem symmetries in order to reduce the search space. With their algorithm, they solved the $n = 15$ case in about 8 days of *cpu*. Another interesting work can be found in (Smith, 2002) where pure constraint programming techniques are used, and the problem is solved in its dual form. In her work, Smith could not improve the $n = 15$ limit. Although not explicitly mentioned, these two works use algorithms with worst-case time complexity $O(2^{(n^2)})$.

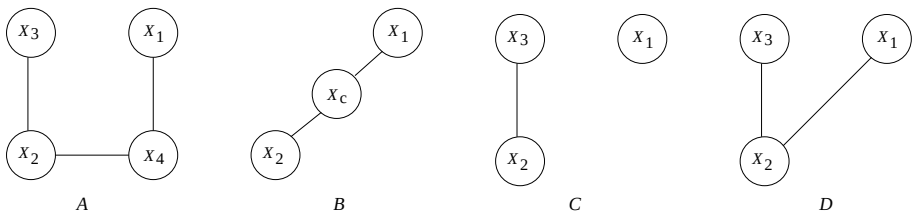

Figure 2: Graphical effect of three problem transformations. A: constraint graph of a simple WCSP instance with four variables and three cost functions. B: the constraint graph after *clustering* variables x_3 and x_4. C: the constraint graph after *assigning* variable x_4. D: the constraint graph after *eliminating* variable x_4.

2 Preliminaries

2.1 WCSP

A *Constraint satisfaction problem* (CSP) (Dechter, 2003) is defined by a tuple (X, D, C), where $X = \{x_1, \ldots, x_n\}$ 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 variables in $var(c)$, $t \in c$ iff t is allowed by the constraint. A *solution* to the CSP is a complete assignment that satisfies every constraint. Constraints can be given explicitly as tables of permitted tuples, or implicitly as logical predicates or computing procedures. The assignment of value a to variable x_i is noted $x_i \leftarrow a$. A set of assignments is represented by a tuple t. The concatenation of tuples t and t' is noted $t \cdot t'$. Thus, $t \cdot (x_i \leftarrow a)$ is the extension of t to $x_i \leftarrow a$.

Weighted constraint satisfaction problems (WCSP) (Bistarelli, Montanari, & Rossi, 1997; Schiex, Fargier, & Verfaillie, 1995) 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 is the sum of costs of all functions whose scope is assigned. 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.*

[1] www.icparc.ic.ac.uk/nys/life

Example 1 *Consider a WCSP with four variables* $X = \{x_i\}_{i=1}^4$ *with domains* $D_i = \{0, 1\}$ *and three cost functions:* $f_1(x_1, x_4) = x_1 + x_4$, $f_2(x_2, x_3) = x_2 x_3$ *and* $f_3(x_2, x_4) = x_2 + x_4$. *Clearly, an optimal solution is* $\forall x_i, x_i \leftarrow 0$ *which has cost* 0.

CSPs and WCSPs instances are graphically depicted by means of their *interaction* or *constraint graph*, which has one node per variable and one edge connecting any two nodes that appear in the same scope of some cost function. For instance, Figure 2.*A* shows the constraint graph of problem in the previous example.

2.2 Clustering

A well-known technique for constraint processing is *clustering* (Dechter & Pearl, 1989). It merges several variables into one *meta-variable*, while preserving the problem semantics. Clustering variables x_i and x_j produces meta-variable x_k, whose domain is $D_i \times D_j$. Cost functions must be accordingly clustered. For instance, in the problem of Example 1, clustering variables x_3 and x_4 produces variable x_c with domain $D_c = \{(0,0), (0,1), (1,0), (1,1)\}$. Cost functions f_2 and f_3 are clustered into $f_c(x_2, x_c) = f_2 + f_3$. With the new variable notation it is $f_c = x_2 x_c(1) + x_2 + x_c(2)$, where $x_c(i)$ denotes the i-th component of x_c. Function f_1 needs to be reformulated as $f_1(x_1, x_c) = x_1 + x_c(2)$. The constraint graph of the resulting problem is obtained by merging the clustered variables and connecting the *meta-node* with all nodes that were adjacent to some of the clustered variables. Figure 2.*B* shows the constraint graph after the clustering of x_3 and x_4. The typical use of clustering is to transform cyclic constraint graphs into a tree, which can be solved efficiently thereafter (Dechter & Pearl, 1989).

2.3 Search

WCSPs are typically solved with a depth-first search procedure. Search algorithms can be defined in terms of *conditioning*,

Definition 1 *Let* f *be an arbitrary function,* x_i *a variable in its scope, and* a *a value in* D_i. Conditioning f *to the assignment* $x_i \leftarrow a$ *is a new function with scope* $var(f) - \{x_i\}$ *which returns for each tuple* t, $f(t \cdot (x_i \leftarrow a))$

For instance, if we condition $f_1(x_1, x_4) = x_1 + x_4$ to $x_4 \leftarrow 1$, we obtain a new function $g(x_1) = x_1 + 1$.

Search algorithms traverse the search tree defined by the problem, where internal nodes represent incomplete assignments and leaf nodes stand for

complete ones. Assigned and unassigned variables are called *past* and *future* variables, respectively. At each internal node, the algorithm selects a future variable and sequentially assigns its domain values. Each assignment forms a successor node, which is the parent problem conditioned to the new assignment. During the traversal, the algorithms keeps the cost of the best solution found so far. Its cost is an *upper bound* UB on the problem's optimum cost. At the end of the execution, when the whole search space has been explored, UB is the problem best solution. At each internal node, defined by its current partial assignment t, the algorithm computes a *lower bound* $LB(t)$ which underestimates the best solution that can be found by extending t. When $UB \leq LB(t)$, the current best cost cannot be improved by extending t. Consequently, the algorithm *backtracks* pruning the subtree below the current node. The performance of search algorithms depends on their capability to backtrack as soon as possible. Thus, it is of great importance to provide the algorithm with good (i.e, high) lower bounds and good (i.e, low) upper bounds.

Assigning a variable transforms the current problem. For instance, assigning $x_4 \leftarrow 1$ in the problem of Example 1, produces a new problem with three variables $\{x_i\}_{i=1}^{3}$ and three cost functions: $f_1(x_1) = x_1 + 1$, $f_2(x_2, x_3) = x_2 x_3$ and $f_3(x_2) = x_2 + 1$. Figure 2.C shows the resulting constraint graph. Observe that it only depends on the variable that is assigned, not on the actual value that is assigned to it.

2.4 Variable Elimination

Variable elimination (Dechter, 1999; Bertele & Brioschi, 1972) is an alternative to search for WCSP solving. It is based on the following two operations,

Definition 2 *The* sum *of two functions f and g, noted $(f + g)$, is a new function with scope $var(f) \cup var(g)$ which returns for each tuple the sum of costs of f and g,*

$$(f + g)(t) = f(t) + g(t)$$

Definition 3 *The* elimination *of variable x_i from f, noted $f \Downarrow x_i$, is a new function with scope $var(f) - \{x_i\}$ which returns for each tuple t the best extension of t to x_i,*

$$(f \Downarrow x_i)(t) = \min_{a \in D_i} \{f(t \cdot (x_i \leftarrow a))\}$$

Observe that when f is a unary function (*i.e.*, arity one), eliminating the only variable in its scope produces a constant.

Definition 4 *Let $P = \{X, D, C\}$ be a WCSP instance. Let $x_i \in X$ be an arbitrary variable and let B_i be the set of all cost functions having x_i in their scope. We define g_i as*

$$g_i = (\sum_{f \in B_i} f) \Downarrow x_i$$

The elimination of x_i transforms P into a new problem $P' = \{X - \{x_i\}, D - \{D_i\}, (C - B_i) \cup g_i\}$. In words, P' is obtained by replacing x_i and all the functions of B_i by g_i.

P and P' have the same optimal cost because, by construction, g_i *compensates* the absence of x_i. The constraint graph of the resulting problem is obtained by forming a clique with all the nodes adjacent to node x_i and then removing x_i and all its adjacent edges. For example, eliminating x_4 in the problem of Example 1 produces a new problem with three variables $\{x_i\}_{i=1}^{3}$ and two cost functions: f_2 and g_4. The scope of g_4 is $\{x_1, x_2\}$ and it is defined as, $g_4 = (f_1 + f_3) \Downarrow x_4 = (x_1 + x_4 + x_2 + x_4) \Downarrow x_4 = x_1 + x_2$. Figure 2.$D$ shows the constraint graph after the elimination.

In the previous example, the new function g_4 could be expressed as a mathematical expression. Unfortunately, in general, the result of summing functions or eliminating variables cannot be expressed intensionally. Therefore, new cost functions must be stored extensionally in tables, which causes the high space complexity of variable elimination.

Bucket elimination (BE) uses an arbitrary variable ordering o that we assume, without loss of generality, lexicographical (i.e, $o = (x_1, x_2, \ldots, x_n)$). The set of cost functions is partitioned into so-called *buckets* B_i. There are n buckets, one per variable. Bucket B_i contains all cost functions having x_i as the highest variable in their scope. BE works in two phases. In the first phase, variables are eliminated one by one, from last to first. In the elimination of x_i, the new g_i function is computed and added to the corresponding bucket. The elimination of x_1 produces an empty-scope function (*i.e.*, a constant) which is the optimal cost of the problem. In the second phase, BE 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, variables are recovered from first to last according to o. The optimal value for x_i is the best extension of t with respect to the sum of functions in B_i. Without any additional overhead BE can also compute the number of optimal solutions (see (Dechter, 1999) for details).

In some cases, it may be convenient to eliminate a set of variables simultaneously (Dechter & Fatah, 2001). The elimination of the set of variables Y is performed by collecting in B_Y the set of functions mentioning at least

one variable of Y. Variables in Y and functions in B_Y are replaced by a new function g_Y defined as,

$$g_Y = (\sum_{f \in B_Y} f) \Downarrow Y$$

The set B_Y is called a *super-bucket*. Note that the elimination of Y can be seen as the clustering of its variables into a meta-variable x_Y and its elimination.

2.5 Mini-Buckets

When the space complexity of BE is too high, an approximation, called Mini-Buckets (MB (Dechter & Rish, 2003)), can be used. Consider the elimination of x_i, with its associated bucket $B_i = \{f_{i_1}, \ldots, f_{i_k}\}$. BE would compute,

$$g_i = (\sum_{f \in B_i} f) \Downarrow x_i$$

The time and space complexity of this computation depends on the arity of g_i. If it is beyond our available resources, we can partition bucket B_i into so-called *mini-buckets* $B_{i_1}, \ldots, B_{i_k}$ where the number of variables in the scopes of each mini-bucket is bounded by a parameter. Then we can compute,

$$g_{i_j} = (\sum_{f \in B_{i_j}} f) \Downarrow x_i, \quad j = 1..k$$

where each g_{i_j} has a bounded arity. Since,

$$\overbrace{(\sum_{f \in B_i} f) \Downarrow x_i}^{g_i} \geq \sum_{j=1}^{k} \overbrace{(\sum_{f \in B_{i_j}} f) \Downarrow x_i}^{g_{i_j}}$$

the elimination of variables using mini-buckets yields a lower bound of the actual optimal cost.

3 Still-life as a WCSP

The *still-life* problem $SL(n)$ consist on finding a $n \times n$ stable pattern of maximum density in the game of life (Bosch & Trick, 2002). All cells outside the pattern are assumed to be dead. Considering the rules of the game, it is easy to see that each cell (i, j) must satisfy the following three constraints: If the cell is alive, it must have exactly two or three living neighbors. If the

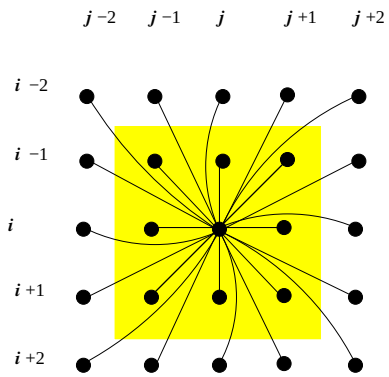

Figure 3: Structure of the constraint graph of $SL(n)$. The node in the center is associated to cell (i, j). It is linked to all cells it interacts with. The shadowed area illustrates the scope of S_{ij} (*i.e*, x_{ij} and its 8 neighbors).

cell is dead, it must not have three living neighbors. If the cell is at the grid boundary (i.e, $i = 1$ or $i = n$ or $j = 1$ or $j = n$), it cannot be part of a sequence of three consecutive living cells along the boundary. The last constraint is needed because three consecutive living cells at a boundary would produce living cells outside the grid.

Example 2 *Figure 1 (right) shows a solution to* $SL(3)$*. It is easy to verify that all its cells satisfy the previous conditions, hence it is stable. The pattern is optimal because it has* 6 *living cells and no* 3×3 *stable pattern with more that* 6 *living cells exists.*

$SL(n)$ can be easily formulated as a WCSP. There is a variable x_{ij} for each cell (i, j). Each variable has two domain values. If $x_{ij} = 0$ the cell is dead, if $x_{ij} = 1$ it is alive. There is a cost function S_{ij} for each variable x_{ij}. The scope of S_{ij} is x_{ij} and all its neighbors. It evaluates the stability of x_{ij}. If the value of x_{ij} and its neighbors does not satisfy some of the three constraints, S_{ij} returns ∞. Else, S_{ij} returns $1 - x_{ij}$.[2]

Thus, the goal is to find the complete assignment $X = \{x_{ij}\}_{i,j=1}^{n}$ with minimal sum of costs,

$$F(X) = \sum_{i=1}^{n} \sum_{j=1}^{n} S_{ij}$$

[2]Recall that, as a WCSP, the task is to *minimize* the number of dead cells. Therefore, we give cost 1 to dead cells and cost 0 to living cells.

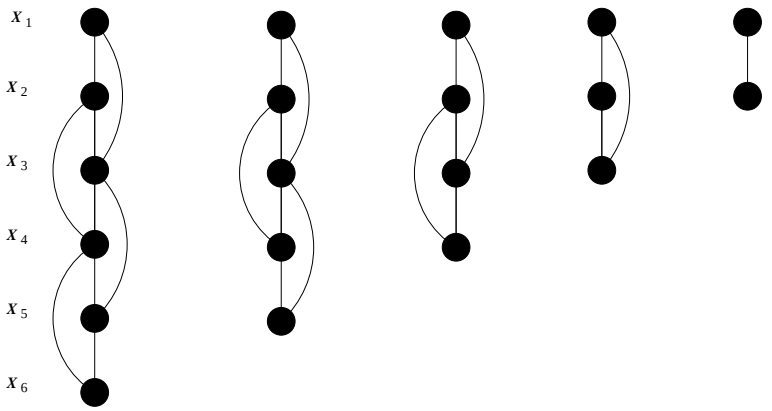

Figure 4: *Left:* Constraint graph of $SL(6)$ after clustering cells into row variables. *From Left to Right:* Evolution of the constraint graph during the execution of BE.

Figure 3 illustrates the structure of the constraint graph of $SL(n)$. The picture shows an arbitrary node x_{ij} (in the center) linked to all the nodes it interacts with. For instance, There is an edge between x_{ij} and $x_{i,j+1}$ because $x_{i,j+1}$ is a neighbor of x_{ij} in the grid and, consequently, both variables are in the scope of S_{ij}. There is an edge between x_{ij} and $x_{i-1,j-2}$ because both cells are neighbors of $x_{i-1,j-1}$ in the grid and, therefore, both appear in the scope of $S_{i-1,j-1}$. The shadowed area represents the scope of S_{ij} (namely, x_{ij} and all its neighbors). The complete graph (not depicted for the sake of clarity) is obtained by extending this connectivity pattern to all nodes in the graph.

4 Solving SL(n) with Bucket Elimination

In order to apply BE to $SL(n)$, we use a more compact formulation that makes the algorithm much simpler to describe and implement. We cluster all variables of each row into a single variable. Thus, x_i denotes the state of cells in the i-th row (namely, $x_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ with $x_{ij} \in \{0, 1\}$). Accordingly, it takes values over the sequences of n bits or, equivalently, over the natural numbers in the interval $[0..2^n - 1]$. Cost functions are accordingly clustered: there is a cost function f_i associated to each row i, defined as,

$$f_i = \sum_{j=1}^{n} S_{ij}$$

function $\texttt{BE}(i, g_{i+1})$
1. **if** $i = 2$ **then**
2. $\quad (x_1, x_2) := \texttt{argmin}_{a,b \in [0..2^n-1]}\{f_1(a,b) + g_3(a,b)\}$;
3. $\quad opt := g_3(x_1, x_2) + f_1(x_1, x_2)$;
4. **else**
5. $\quad$ **for** $a, b \in [0..2^n - 1]$ **do**
6. $\quad\quad g_i(a,b) := \texttt{min}_{c \in [0..2^n-1]}\{f_{i-1}(a,b,c) + g_{i+1}(b,c)\}$;
7. $\quad$ **endfor**
8. $\quad (opt, (x_1, x_2, \ldots, x_{i-1})) :=\texttt{BE}(i-1, g_i)$;
9. $\quad x_i := \texttt{argmin}_{c \in [0..2^n-1]}\{f_{i-1}(x_{i-2}, x_{i-1}, c) + g_{i+1}(x_{i-1}, c)\}$;
10. **endif**
11. **return**$(opt, (x_1, x_2, \ldots, x_{i-1}, x_i))$;

endfunction

Figure 5: Bucket Elimination for the still-life problem. The algorithm receives the index i of the next variable that must be eliminated and the function g_{i+1} created in the previous call. It returns the optimal value *opt* and the optimal assignment in $(x_1, x_2, \ldots, x_i)$ of the current problem. The initial call is $\texttt{BE}(n, f_n)$.

With the new notation the goal is to find the complete assignment $X = \{x_i\}_{i=1}^n$ with minimal sum of costs

$$F(X) = \sum_{i=1}^{n} f_i$$

For internal rows, the scope of f_i is $\{x_{i-1}, x_i, x_{i+1}\}$. The cost function of the top row, f_1, has scope $\{x_1, x_2\}$. The cost function of the bottom row, f_n, has scope $\{x_{n-1}, x_n\}$. If there is some unstable cell in x_i, $f_i = \infty$. Else, it returns the number of dead cells in x_i. Computing f_i is $\Theta(n)$ because all the bits of the arguments need to be checked. Figure 4 (left) shows the constraint graph of $SL(6)$ with this formulation. An arbitrary variable x_i is connected with the two variables above and the two variables below.

`BE` (Figure 5) is the algorithmic description of BE for the $SL(n)$ problem. Although BE is usually described iteratively, we propose a recursive description which will facilitate the development of the hybrid algorithm introduced in the next Section. The sequential structure of the constraint graph makes the encoding very simple. The algorithm receives the index i of the next vari-

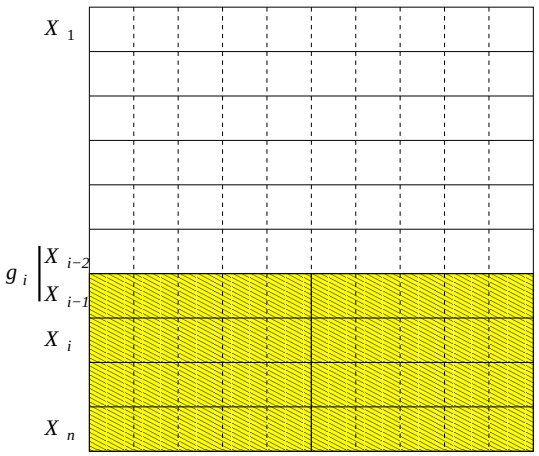

Figure 6: The elimination of x_i produces a new function $g_i(x_{i-2}, x_{i-1})$. The shadowed area indicates the part of the grid where the information of g_i comes from.

able that must be eliminated and the function g_{i+1} (with scope $\{x_{i-1}, x_i\}$) created in the previous call, due to the elimination of x_{i+1}. `BE` solves the current problem by eliminating x_i and recursively solving the resulting problem. The bucket of x_i is $B_i = \{g_{i+1}, f_{i-1}\}$ (these are the only cost function having x_i in their scope). The new function $g_i = (f_{i-1} + g_{i+1}) \Downarrow x_i$ with scope $\{x_{i-2}, x_{i-1}\}$ is computed in lines 5-7. The recursive call is made in line 8, which returns the optimal value and the optimal assignment to variables $(x_{i-1}, \ldots, x_1)$. The extension of the optimal assignment to x_i is computed as the best extension according to bucket B_i (line 9). In our implementation $i = 2$ is the non-recursive case which we solve with a brute-force exhaustive search (lines 2,3). Figure 4 shows the evolution of the constraint graph through the elimination of its variables. At an arbitrary call $\texttt{BE}(i, g_{i+1})$, the algorithm satisfies the following property,

Property 1 *$g_i(a, b)$ is the cost of the best extension of $(x_{i-2} \leftarrow a,\ x_{i-1} \leftarrow b)$ to the eliminated variables $(x_i, \ldots, x_n)$. Formally,*

$$\begin{aligned} g_i(a,b) \quad = \quad & \min_{v_i \in D_i, \ldots, v_n \in D_n} \{f_{i-1}(a, b, v_i) + f_i(b, v_i, v_{i+1}) + \\ & + f_{i+1}(v_i, v_{i+1}, v_{i+2}) + \ldots \\ & + f_{n-1}(v_{n-2}, v_{n-1}, v_n) + f_n(v_{n-1}, v_n)\} \end{aligned}$$

Figure 6 shows graphically the meaning of $g_i(a, b)$. If $g_i(a, b) = \infty$, the pattern a, b cannot be extended to the inferior rows forming a stable pattern.

n	opt	n. sol.	BE	CP	IP	CP/IP
5	9	1	0	0	0	0
6	18	48	0	0	1	0
7	21	2	0	4	3	0
8	28	1	0	76	26	2
9	38	76	4	> 600	> 600	20
10	46	3590	27	*	*	60
11	57	73	210	*	*	153
12	68	129126	1638	*	*	11536
13	79	1682	13788	*	*	12050
14	92	11	10^5	*	*	5×10^5
15	106	?	*	*	*	7×10^5

Figure 7: Experimental results of four different algorithms on the still-life problem. Times are in seconds.

If $g_i(a, b) = k$ ($k \neq \infty$), a, b can be extended and the optimal extension has k dead cells from x_{i-1} to x_n (shadowed area in Figure 6).

The space complexity of the algorithm is $\Theta(n \times 2^{2n})$, due to the space required to store functions g_i extensionally (2^{2n} entries). Regarding time, the critical part of the algorithm is the execution of lines 5-7. Line 6 has complexity $\Theta(n \times 2^n)$ (finding the minimum of 2^n alternatives, the computation of each one is $\Theta(n)$). It has to be executed 2^{2n} times at each recursive call, which makes a global complexity of $\Theta(n^2 \times 2^{3n})$. Observe that the time complexity of BE applied to $SL(n)$ is an exponential improvement over search algorithms, which have complexity $O(2^{n^2})$.

Table 7 reports some empirical results. They were obtained with a 2 Ghz Pentium IV machine with 2 Gb of memory. From left to right, the first three columns report: problem size, solution cost as the number of dead 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 BE in seconds. The fifth, sixth and seventh columns show the results obtained with the three approaches tried by (Bosch & Trick, 2002):[3] constraint programming (CP), integer programming (IP), and a more sophisticated algorithm (CP/IP) which combines CP and IP, and exploits the problem symmetries.

[3]The OPL code is available at http://mat.gsia.cmu.edu/LIFE.

It can be observed that BE clearly outperforms CP and IP by orders of magnitude. The $n = 14$ case is the largest instance that we could solve due to space exhaustion. Comparing BE with CP/IP, we observe that there is no clear winner. 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.

5 Combining Search and Variable Elimination

In (Larrosa & Dechter, 2003) it was suggested to overcome the high space complexity of BE by combining search and variable elimination in a hybrid approach. The idea is to use search (i.e, conditioning) in order to *break* the problem into independent smaller parts where variable elimination can be efficiently performed.

Let us reformulate the problem in a more convenient way for the hybrid algorithm. For the sake of simplicity and without loss of generality consider that n is even. We cluster row variables into three meta-variables: x_i^C denotes the two central cells of row i, x_i^R and x_i^L denote the $\frac{n}{2} - 1$ remaining cells on the right and left, respectively (see Figure 8.A). Consequently, x_i^C takes values in the range [0..3], x_i^L and x_i^R take values in the range $[0..2^{\frac{n}{2}-1} - 1]$. Cost functions are accordingly clustered,

$$f_i^L = \sum_{j=1}^{\frac{n}{2}} S_{ij}$$

$$f_i^R = \sum_{j=\frac{n}{2}+1}^{n} S_{ij}$$

With the new notation the goal is to find the complete assignment $X = \{x_i^L, x_i^C, x_i^R\}_{i=1}^n$ with minimal sum of costs

$$F(X) = \sum_{i=1}^{n} (f_i^L + f_i^R)$$

The scopes of internal row functions, f_i^L and f_i^R, are $\{x_{i-1}^L, x_{i-1}^C, x_i^L, x_i^C, x_{i+1}^L, x_{i+1}^C\}$ and $\{x_{i-1}^C, x_{i-1}^R, x_i^C, x_i^R, x_{i+1}^C, x_{i+1}^R\}$. Top functions f_1^L and f_1^R have scopes $\{x_1^L, x_1^C, x_2^L, x_2^C\}$ and $\{x_1^C, x_1^R, x_2^C, x_2^R\}$. Bottom functions f_n^L and f_n^R have scopes $\{x_{n-1}^L, x_{n-1}^C, x_n^L, x_n^C\}$ and $\{x_{n-1}^C, x_{n-1}^R, x_n^C, x_n^R\}$. Figure 8.$B$ shows the

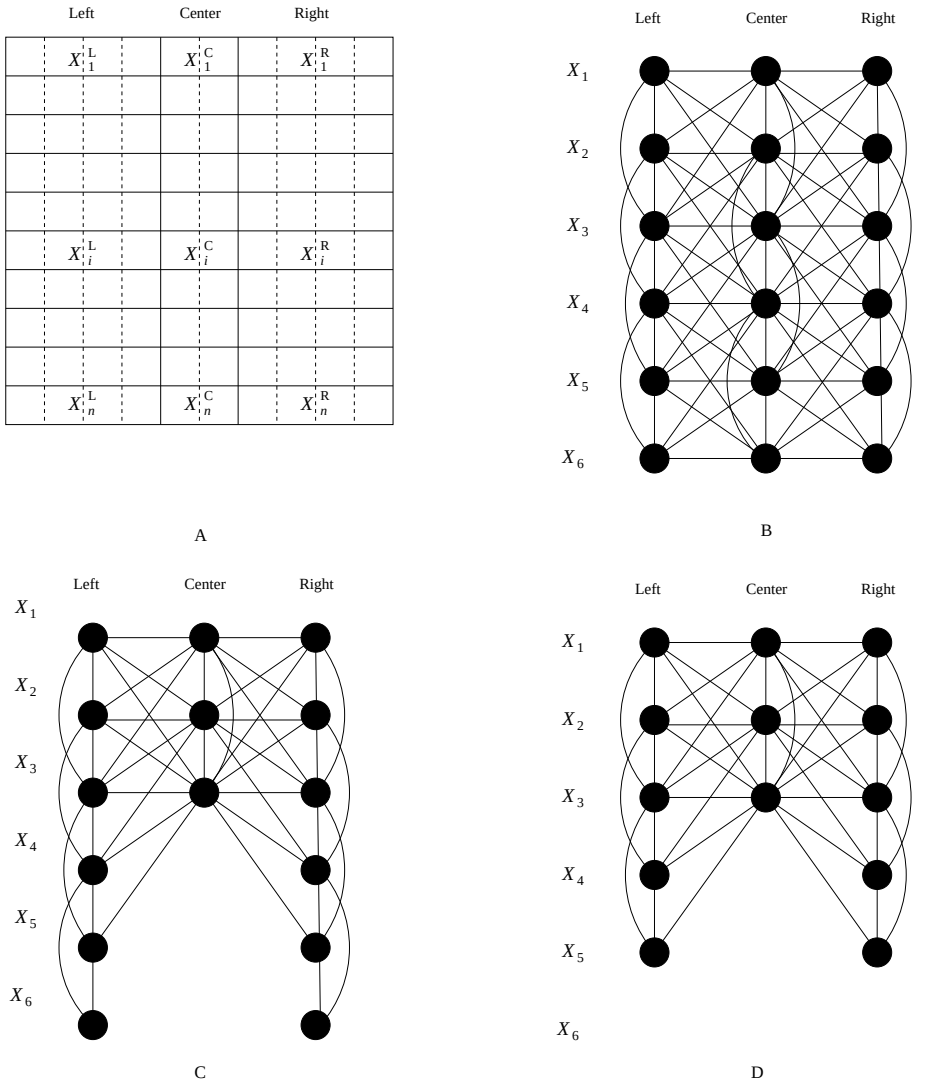

Figure 8: Formulation of $SL(n)$ used by the hybrid algorithm. A: Each row is clustered into three variables. B: Constraint graph of $SL(6)$. C: Constraint graph after the assignment of x_n^C, x_{n-1}^C and x_{n-2}^C. D: Constraint graph after the elimination of x_n^L and x_n^R.

corresponding constraint graph. The importance of this formulation is that x_i^L and x_i^R are independent (i.e, there is no edge in the constraint graph connecting left and right variables).

The hybrid algorithm searches over the central variables and eliminates the lateral variables. Variables are considered in decreasing order of their index. Thus, the algorithm starts instantiating x_n^C, x_{n-1}^C and x_{n-2}^C, which produces a subproblem with the constraint graph shown in Figure 8.*C*. Observe that variable x_n^L (respectively, x_n^R) is only connected with variables x_{n-1}^L and x_{n-2}^L (respectively, x_{n-1}^R and x_{n-2}^R). Therefore, its elimination produces a new function g_n^L with scope $\{x_{n-2}^L, x_{n-1}^L\}$ (respectively, g_n^R with scope $\{x_{n-2}^R, x_{n-1}^R\}$). Figure 8.*D* shows the resulting constraint graph. Lateral variables have domains of size $2^{\frac{n}{2}-1}$. Hence, the elimination is space $\Theta(2^n)$ and time $\Theta(2^{3\frac{n}{2}})$. It is important to note that these eliminations are *conditioned* to the current assignment of x_n^C and x_{n-1}^C. Therefore, they have to be recomputed when their value change. After the elimination of x_n^L and x_n^R, the algorithm would assign variable x_{n-3}^C which will make possible the elimination of x_{n-1}^L and x_{n-1}^R, and so on. At an arbitrary level of search, the algorithm assigns x_i^C, which makes x_{i+2}^L and x_{i+2}^R independent of the central columns and only related to their two variables above. Then, it eliminates them by replacing the variables by functions g_{i+2}^L and g_{i+2}^R with scopes $\{x_i^L, x_{i+1}^L\}$ and $\{x_i^R, x_{i+1}^R\}$, respectively.

Figure 9 shows `HYB`, the algorithm that implements this idea. The algorithm receives the current assignment of the central variables $X^C = (x_1^C, x_2^C, \ldots, x_n^C)$, the index i of the next variable to be assigned, and functions g_{i+3}^L and g_{i+3}^R which resulted from the elimination of x_{i+3}^L and x_{i+3}^R, performed at the parent node. In the general case $1 \leq i = n$, the algorithm sequentially instantiates variable x_i^C with its four domain values (line 9). At each iteration it eliminates variables x_{i+2}^L and x_{i+2}^R (lines 22-25). Next, it computes the lower bound of the current node (the details will be discussed later). Finally, if the current node can improve the best solution found so far, search continues recursively; else the algorithm backtracks from the current line of search. The cases $i = n$ and $i = n-1$ are recursive cases, but are treated separately because no lateral variable can be eliminated yet (they receive *dummy* g_{i+3}^L and g_{i+3}^R functions). If $i = n$, `HYB` computes dummy g_{n+2}^L and g_{n+2}^R and continues the search by making the recursive call. If $i = n-1$, it initializes g_{n+1}^L and g_{n+1}^R as f_n^L and f_n^R, conditioned to the current assignment of X^C and continues the search by making the recursive calls. The non-recursive case is $i = 0$ (line 1), where all central variables are assigned. The current problem contains four variables (x_2^L, x_1^L, x_2^R and

procedure HYB($X^C, i, g^L_{i+3}, g^R_{i+3}$)

1. **if** $i = 0$ **then**
2. $\quad LB^L := \mathtt{min}_{a,b\in[0..2^{\frac{n}{2}-1}-1]}\{f^L_1(a, x^C_1, b, x^C_2) + g^L_3(a, b)\}$
3. $\quad LB^R := \mathtt{min}_{a,b\in[0..2^{\frac{n}{2}-1}-1]}\{f^R_1(x^C_1, a, x^C_2, b) + g^R_3(a, b)\}$
4. $\quad$ **if** $LB^L + LB^R < UB$ **then**
5. $\quad\quad UB := LB^L + LB^R$
6. $\quad\quad BestSol :=$ RetrieveSol(X^C, g^L, g^R)
7. $\quad$ **endif**
8. **else**
9. $\quad$ **for** $v := 0$ **to** 3 **do**
10. $\quad\quad x^C_i := v$
11. $\quad\quad$ **if** $i = n$ **then**
12. $\quad\quad\quad$ **for** $a, b \in [0..2^{\frac{n}{2}-1} - 1]$ **do**
13. $\quad\quad\quad\quad g^L_{n+2}(a, b) := 0$
14. $\quad\quad\quad\quad g^R_{n+2}(a, b) := 0$
15. $\quad\quad\quad$ **endfor**
16. $\quad\quad$ **elseif** $i = n - 1$ **then**
17. $\quad\quad\quad$ **for** $a, b \in [0..2^{\frac{n}{2}-1} - 1]$ **do**
18. $\quad\quad\quad\quad g^L_{n+1}(a, b) := f^L_n(a, x^C_{n-1}, b, x^C_n)$
19. $\quad\quad\quad\quad g^R_{n+1}(a, b) := f^R_n(x^C_{n-1}, a, x^C_n, b)$
20. $\quad\quad\quad$ **endfor**
21. $\quad\quad$ **else**
22. $\quad\quad\quad$ **for** $a, b \in [0..2^{\frac{n}{2}-1} - 1]$ **do**
23. $\quad\quad\quad\quad g^L_{i+2}(a, b) := \mathtt{min}_{c\in[0..2^{\frac{n}{2}-1}-1]}\{f^L_{i+1}(a, x^C_i, b, x^C_{i+1}, c, x^C_{i+2}) + g^L_{i+3}(b, c)\}$
24. $\quad\quad\quad\quad g^R_{i+2}(a, b) := \mathtt{min}_{c\in[0..2^{\frac{n}{2}-1}-1]}\{f^R_{i+1}(x^C_i, a, x^C_{i+1}, b, x^C_{i+2}, c) + g^R_{i+3}(b, c)\}$
25. $\quad\quad\quad$ **endfor**
26. $\quad\quad$ **endif**
27. $\quad\quad\quad LB^L := \mathtt{min}_{a,b\in[0..2^{\frac{n}{2}-1}-1]}\{g^L_{i+2}(a, b)\}$
28. $\quad\quad\quad LB^R := \mathtt{min}_{a,b\in[0..2^{\frac{n}{2}-1}-1]}\{g^R_{i+2}(a, b)\}$
29. $\quad\quad$ **endif**
30. $\quad\quad$ **if** $LB^L + LB^R < UB$ **then** HYB$(X^C, i - 1, g^L_{i+2}, g^R_{i+2})$
31. $\quad$ **endfor**
32. **endif**

endprocedure

Figure 9: A hybrid algorithm for the still-life problem $SL(n)$.

x_1^R) and four functions (g_3^L, f_1^L, g_3^R and f_1^R). The problem is formed by two independent parts, which are solved by a brute force exhaustive search (Lines 2,3). If its cost is lower than the current best solution, the upper bound is updated (line 5) and the best solution is retrieved (line 6, details omitted). The initial call is $\texttt{HYB}(X^C, n, g_{n+3})$, with $g_{n+3}(a,b) = 0, \forall a, b$.

g_i^L and g_i^R in `HYB` play the role of g_i in `BE`. Consequently, property 1 can be adapted to the hybrid algorithm as follows,

Property 2 *$g_i^L(a,b)$ is the cost of the best extension of $(x_{i-2}^L \leftarrow a,\ x_{i-1}^L \leftarrow b)$ to eliminated variables $(x_i^L, \ldots, x_n^L)$, conditioned to X^C. Formally,*

$$\begin{aligned} g_i^L(a,b) \quad = \quad & \min_{v_i \in D_i^L, \ldots, v_n \in D_n^L} \{ f_{i-1}^L(a, x_{i-2}^C, b, x_{i-1}^C, v_i, x_i^C) + \\ & + f_i^L(b, x_{i-1}^C, v_i, x_i^C, v_{i+1}, x_{i+1}^C) + \\ & + f_{i+1}^L(v_i, x_i^C, v_{i+1}, x_{i+1}^C, v_{i+2}, x_{i+2}^C) + \ldots + \\ & + f_{n-1}^L(v_{n-2}, x_{n-2}^C, v_{n-1}, x_{n-1}^C, v_n, x_n^C) + f_n(v_{n-1}, x_{n-1}^C, v_n, x_n^C) \} \end{aligned}$$

Similarly, for the right side, $g_i^R(a,b)$ is the cost of the best extension of $(x_{i-2}^R \leftarrow a,\ x_{i-1}^R \leftarrow b)$ to eliminated variables $(x_i^R, \ldots, x_n^R)$, conditioned to X^C,

$$\begin{aligned} g_i^R(a,b) \quad = \quad & \min_{v_i \in D_i^R, \ldots, v_n \in D_n^R} \{ f_{i-1}^R(x_{i-2}^C, a, x_{i-1}^C, b, x_i^C, v_i) + \\ & + f_i^L(x_{i-1}^C, b, x_i^C, v_i, x_{i+1}^C, v_{i+1}) + \\ & + f_{i+1}^L(x_i^C, v_i, x_{i+1}^C, v_{i+1}, x_{i+2}^C, v_{i+2}) + \ldots + \\ & + f_{n-1}^L(x_{n-2}^C, v_{n-2}, x_{n-1}^C, v_{n-1}, x_n^C, v_n) + f_n(x_{n-1}^C, v_{n-1}, x_n^C, v_n) \} \end{aligned}$$

Figure 10 gives a pictorial interpretation of the previous property. The black area represents the central variables that are instantiated. The shadowed area represents the source of the information contained in g_i^L and g_i^R.

A consequence of the previous Property is that the minimum $g_{i+2}^L(a,b)$ among all combinations of a and b is a lower bound of the best cost that can be obtained in the left part of the grid if we continue the current line of search. Therefore, $\min_{a,b}\{g_{i+2}^L(a,b)\} + \min_{a,b}\{g_{i+2}^R(a,b)\}$ is a valid lower bound of the current node.

The space complexity of the algorithm is $\Theta(n \times 2^n)$, due to the g_i^L and g_i^R functions which need to be explicitly stored. The time complexity is $O(n \times 2^{3.5n})$, because $O(4^n)$ nodes may be visited and the cost of each node is $\Theta(n \times 2^{3\frac{n}{2}})$ due to the variable eliminations.

Thus, comparing with BE, the time complexity increases from $\Theta(n^2 \times 2^{3n})$ to $O(n \times 2^{3.5n})$. This is the prize HYB pays for the space decrement from $\Theta(n \times 2^{2n})$ to $\Theta(n \times 2^n)$.

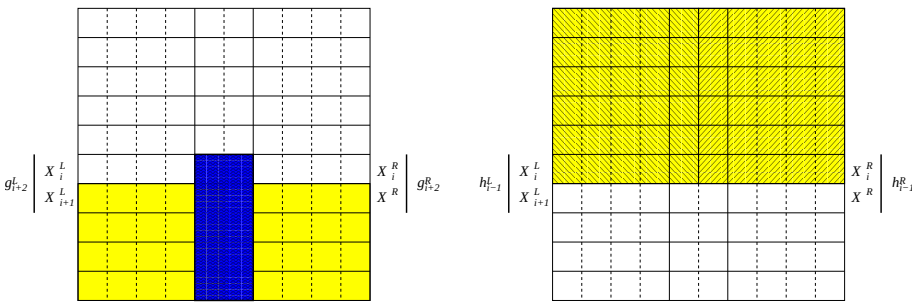

Figure 10: (*Left*) An arbitrary search state of HYB. The two central columns are instantiated from the n-th to the i-th row. Lateral parts are eliminated from the n-th to the $i+2$-th rows. The effect of the elimination is recorded in the g^L_{i+2} and g^R_{i+2} functions which contain information about the shadowed area. (*Right*) The shadowed area indicates the source of information of the h^L_{i-1} and h^R_{i-1} functions. Observe that it does not overlap with the g^L_{i+2} and g^R_{i+2} functions computed by HYB. Therefore, they can be safely combined in order to form a stronger lower bound.

5.1 Refining the Lower Bound

It is well-known that the average-case efficiency of search algorithms depends greatly on the lower bound that they use. Our algorithm is using a poor lower bound based on the g^L_i and g^R_i functions. These functions only contain information from the processed part of the grid (see Property 2 and Figure 10).

Kask and Dechter (Kask & Dechter, 2001) proposed a general method to incorporate information from yet-unprocessed variables into the lower bound. Roughly, the idea is to run Mini-Buckets (MB) prior search and save intermediate functions for future use. MB is executed using the reverse order in which search will instantiate the variables. When the execution of MB is completed, the search algorithm is executed. At each node, it uses mini-bucket functions as compiled look-ahead information. In this Subsection, we show how we have adapted this idea to $SL(n)$ and how we have integrated it into HYB.

Consider $SL(n)$ formulated in terms of *left, central* and *right* variables (x^L_i, x^C_i, x^R_i). The *exact* elimination of the first row variables (x^L_1, x^C_1, x^R_1) uses *super-bucket* $B_1 = \{f^L_1, f^R_1, f^L_2, f^R_2\}$ and computes the function,

$$h_1 := (f^L_1 + f^R_1 + f^L_2 + f^R_2) \Downarrow \{x^L_1, x^C_1, x^R_1\}$$

The scope of h_1 is $\{x^L_2, x^C_2, x^R_2, x^L_3, x^C_3, x^R_3\}$. Using the mini-buckets idea, we

procedure MB(i)
1. **if** $i < n-1$ **then**
2. **for** $a, b \in [0..2^{\frac{n}{2}-1}-1],\ a', b' \in [0..3]$ **do**
3. $h_i^L(a,a',b,b') := \mathtt{min}_{c\in[0..2^{\frac{n}{2}}-1],\ c'\in[0..3]}\{f_{i-1}^L(a,a',b,b',c,c')+$
$+h_{i-1}^L(b,b',c,c')\}$
4. $h_i^R(a',a,b',b) := \mathtt{min}_{c\in[0..2^{\frac{n}{2}}-1],\ c'\in[0..3]}\{f_{i-1}^R(a',a,b',b,c',c)+$
$+h_{i-1}^R(b',b,c',c')\}$
5. MB($i+1$);

endprocedure

Figure 11: Mini-Buckets pre-process for the still-life problem $SL(n)$. It computes h_i^L and h_i^R functions that are later used by HYB. The initial call is MB(1). h_0^L and h_0^L must be initialized to f_1^L and f_1^R.

partition the bucket into $B_1^L = \{f_1^L, f_2^L\}$ and $B_1^R = \{f_1^R, f_2^R\}$. Then, we approximate h_1 by two smaller functions h_1^L and h_1^R,

$$h_1^L := (f_1^L + f_2^L) \Downarrow \{x_1^L, x_1^C\}$$

$$h_1^R := (f_1^R + f_2^R) \Downarrow \{x_1^C, x_1^R\}$$

The scopes of h_1^L and h_1^R are $\{x_2^L, x_2^C, x_3^L, x_3^C\}$ and $\{x_2^C, x_2^R, x_3^C, x_3^R\}$, respectively. The same idea is repeated row by row. In general, processing row i, yields two functions,

$$h_i^L := (h_{i-1}^L + f_{i+1}^L) \Downarrow \{x_i^L, x_i^C\}$$

$$h_i^R := (h_{i-1}^R + f_{i+1}^R) \Downarrow \{x_i^C, x_i^R\}$$

The scopes of h_i^L and h_i^R are $\{x_{i+1}^L, x_{i+1}^C, x_{i+2}^L, x_{i+2}^C\}$ and $\{x_{i+1}^C, x_{i+1}^R, x_{i+2}^C, x_{i+2}^R\}$, respectively. MB (Figure 11) implements this idea. It is very similar to BE (Figure 5). The main difference is that the order of elimination is reversed, computations are *split* into two parts and no result is returned (we only care for the h_i functions which must be declared as global variables so that they can be used by HYB). Before the execution, h_0^L and h_0^L must be initialized to f_1^L and f_1^R, respectively.

By construction, $h_i^L(a,a',b,b')$ contains the cost of the best extension of a, a', b, b' to processed variables $x_i^L, x_i^C, \ldots, x_1^L, x_1^C$ considering left functions only. The same for $h_i^R(a',a,b',b)$. Formally,

Property 3 *The execution of* MB *generates* $h_i^L(x_{i+1}^L, x_{i+1}^C, x_{i+2}^L, x_{i+2}^C)$ *and* $h_i^R(x_{i+1}^C, x_{i+1}^R, x_{i+2}^C, x_{i+2}^R)$ *functions such that,*

$$\begin{aligned} h_i^L(a,a',b,b') \;=\; & \min_{v_1^L\in D_1^L, v_1^C\in D_1^C, \ldots, v_i^L\in D_i^L, v_i^C\in D_i^C} \{f_1^L(v_1^L, v_1^C, v_2^L, v_2^C) + \\ & + f_2^L(v_1^L, v_1^C, v_2^L, v_2^C, v_3^L, v_3^C) + \ldots + f_{i+1}^L(v_i^L, v_i^C, a, a', b, b')\} \end{aligned}$$

$$\begin{aligned} h_i^L(a',a,b',b) \;=\; & \min_{v_1^C\in D_1^C, v_1^R\in D_1^R, \ldots, v_i^C\in D_i^C, v_i^R\in D_i^R} \{f_1^R(v_1^C, v_1^R, v_2^C, v_2^R) + \\ & + f_2^R(v_1^C, v_1^R, v_2^C, v_2^R, v_3^C, v_3^R) + \ldots + f_{i+1}^R(v_i^C, v_i^R, a', a, b', b)\} \end{aligned}$$

The complexity of MB is space $\Theta(n \times 2^n)$ and time $\Theta(n^2 \times 2^{1.5n})$. Since these complexities are smaller than the complexity of HYB, running this preprocess does not affect its overall complexity.

After MB is executed, HYB can use the information recorded in the h_i^L and h_i^R functions. Consider an arbitrary node in which HYB assigns x_i^C and eliminates x_{i+2}^L and x_{i+2}^R. Let a and b be domain values of variables x_i^L and x_{i+1}^L. From Property 2 we have that $g_{i+2}^L(a,b)$ contains the best extension of a, b that can be attained in the left part of rows $i+1$ to n as long as the current assignment X^C is maintained. Additionally, from Property 3, we have that $h_{i-1}^L(a, x_i^C, b, x_{i+1}^C)$ contains a lower bound of the best extension of a, b that can be attained in the left part of rows i to 1. Therefore, $g_{i+2}^L(a,b) + h_{i-1}^L(a, x_i^C, b, x_{i+1}^C)$ is a lower bound for a, b and X^C of the left part of the grid. Consequently,

$$\texttt{min}_{a,b\in[0..2^{\frac{n}{2}-1}-1]}\{g_{i+2}^L(a,b) + h_{i-1}^L(a, x_i^C, b, x_{i+1}^C)\}$$

is a lower bound of the left part of the grid for the current assignment. With the same reasoning on the right part we have that,

$$\begin{aligned} & \texttt{min}_{a,b\in[0..2^{\frac{n}{2}-1}-1]}\{g_{i+2}^L(a,b) + h_{i-1}^L(a, x_i^C, b, x_{i+1}^C)\} + \\ & + \texttt{min}_{a,b\in[0..2^{\frac{n}{2}-1}-1]}\{g_{i+2}^R(a,b) + h_{i-1}^R(x_i^C, a, x_{i+1}^C, b)\} \end{aligned}$$

is a lower bound of the current assignment. The new lower bound can be easily incorporated to HYB. Lines 27 and 28 in the code of Figure 9 must be replaced by,

27. $LB^L := \texttt{min}_{a,b\in[0..2^{\frac{n}{2}-1}-1]}\{g_{i+2}^L(a,b) + h_{i-1}^L(a, x_i^C, b, x_{i+1}^C)\}$
28. $LB^R := \texttt{min}_{a,b\in[0..2^{\frac{n}{2}-1}-1]}\{g_{i+2}^R(a,b) + h_{i-1}^L(x_i^C, a, x_{i+1}^C, b)\}$

5.2 Refining the Upper Bound

The efficiency of the algorithm also depends on the initial value of the upper bound. A low upper bound facilitates pruning earlier in the search tree. Bosch (Bosch & Trick, 2002) suggests modifying $SL(n)$ by adding the additional constraint of considering symmetric patterns, only. Since the space of solutions becomes considerably smaller, the problem is presumably simpler. Clearly, the cost of an optimal symmetric stable pattern is an upper bound of the optimal cost of $SL(n)$ (as the number of dead cells). It has been observed that such upper bounds are very tight.

Since the motivation of our work is to use bucket elimination techniques, we have considered still-lifes which are symmetric over a vertical reflection, because they can be efficiently solved using BE. The symmetric still-life problem $SSL(n)$ consists on finding a $n \times n$ stable pattern of maximum density in the game of life subject to a vertical reflection symmetry (namely, the state of cells (i, j) and $(i, n - j + 1)$ must be the same.[4]

Adapting BE to solve $SSL(n)$ is extremely simple: we only need to reduce domains to symmetrical values. Let us assume that n is an even number (the odd case is similar). We represent a symmetric sequences of bits of length n by considering the left side of the sequence (i.e, the first $n/2$ bits). The right part is obtained by reversing the left part. Thus, we represent symmetrical sequences of n bits as integers in the interval $[0..2^{\frac{n}{2}} - 1]$. Reversing a sequence of bits a is noted $\bar{a}$. Hence, if a is a sequence of $n/2$ bits, $a \cdot \bar{a}$ is the corresponding symmetrical sequence of n bits.

Algorithm `BE-SSL` (Figure 12) gives the algorithmic details. It works like `BE`. The only difference is that domains are restricted to symmetrical values. Since we only care for the optimal value, the optimal assignment is not computed. Function $g_i(a, b)$ contains the cost of the optimal extension of $x_{i-2} \leftarrow (a \cdot \bar{a})$ and $x_{i-1} \leftarrow (b \cdot \bar{b})$ to the processed variables.

It is easy to see that the complexity of `BE-SSL` is time $\Theta(n^2 \times 2^{1.5n})$ and space $\Theta(n \times 2^n)$. Therefore, executing `BE-SSL` before `HYB` and setting the upper bound with its optimal cost does not affect the overall complexity of the hybrid.

[4] Unlike the work in (Smith, 2002) we cannot exploit a larger variety of symmetries such as rotations and diagonal reflections

function BE-SSL(i, g_{i+1})
1. **if** $i = 2$ **then**
2. $\quad opt := \min_{a,b \in [0..2^n-1]}\{f_1(a \cdot \bar{a}, b \cdot \bar{b}) + g_3(a, b)\}$;
4. **else**
5. $\quad$ **for** $a, b \in [0..2^n - 1]$ **do**
6. $\quad\quad g_i(a, b) := \mathtt{min}_{c \in [0..2^n-1]}\{f_{i-1}(a \cdot \bar{a}, b \cdot \bar{b}, c \cdot \bar{c}) + g_{i+1}(b, c)\}$;
7. $\quad$ **endfor**
8. $\quad opt :=$BE-SSL$(i - 1, g_i)$;
10. **endif**
11. **return**(opt);

endfunction

Figure 12: Bucket Elimination for the symmetric still-life problem $SSL(n)$. The algorithm receives the index i of the next variable that must be eliminated and the function g_{i+1} created in the previous call. It returns the optimal value *opt* which is used as initial upper bound by the hybrid algorithm. The initial call is BE-SSL(n, f_n).

5.3 Further Exploitation of Symmetries

$SL(n)$ 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.

Symmetries can be exploited at very different algorithmic levels. In general, we can save any computation whose outcome is equivalent to a previous computation due to a symmetry if we have kept its outcome. For instance, in MB it is not necessary to compute $h_i^R(a', a, b', b)$ because it is equal to $h_i^L(a, a', b, b')$ due to a reflection symmetry. Therefore, Line 5 of MB can be removed, which saves half the time and space required by the algorithm. Another example occurs in HYB. Let $x_n^C \leftarrow v_n, x_{n-1}^C \leftarrow v_{n-1}, \ldots, x_i^C \leftarrow v_i$ be the current assignment. The reversed assignment $x_n^C \leftarrow \overline{v_n}, x_{n-1}^C \leftarrow \overline{v_{n-1}}, \ldots, x_i^C \leftarrow \overline{v_i}$ is equivalent due to the reflection symmetry. Thus, if it has already been considered, the algorithm can backtrack. Our implementation uses these tricks and some others which we do not report because it would require a much lower level description of the algorithms.

n	opt	opt-SSL	CP/IP	BE	HYB	HYB no LB	HYB no UB
13	79	79	12050	13788	2	2750	2
14	92	92	5×10^5	10^5	2	7400	3
15	106	106	7×10^5	*	58	2×10^5	61
16	120	120	*	*	7	6×10^5	49
17	137	137	*	*	1091	*	2612
18	153	154	*	*	2029	*	2311
19	171	172	*	*	56027	*	56865
20	190	192	*	*	2×10^5	*	2×10^5
22	?	232	*	*	*	*	*
24	?	276	*	*	*	*	*
26	?	326	*	*	*	*	*
28	?	378	*	*	*	*	*

Figure 13: Experimental results of three different algorithms on the still-life problem. Times are in seconds.

6 Experimental Results

Figure 13 shows the empirical performance of our hybrid algorithm. Experiments were again performed with a Pentium IV at 2GHz with 2Gb of memory. The first column contains the problem size. The second column contains the optimal value as the number of dead cells. The third column contains the optimal value of the symmetrical problem $SSL(n)$, obtained by executing `BE-SSL`. It can be observed that $SSL(n)$ provides very tight upper bounds to $SL(n)$. The fourth column reports the time obtained with the CP/IP algorithm of (Bosch & Trick, 2002). The fifth column reports times obtained with BE. The sixth column contains times obtained with our hybrid algorithm HYB. As it can be seen, the performance of HYB is spectacular. The $n = 14$ and $n = 15$ instances, which required days of CPU, are solved by HYB in a few seconds. Instances up to $n = 18$ are solved in less than one hour. The largest instance that we can solve is $n = 20$, which requires about two days of CPU (Figure 14 shows the optimal $n = 19$ and $n = 20$ still-lifes). Regarding space, our computer can handle executions of HYB up to $n = 22$. However, neither the $n = 21$ nor the $n = 22$ instance could be solved within a week of CPU. It may seem that solving the $n = 20$ instance is a petty progress with respect previous results on the problem. This is clearly not the case. The search space of the $n = 15$ and $n = 20$ instances have size $2^{15^2} = 2^{225}$ and $2^{20^2} = 2^{400}$, respectively. Thus, we have been able to solve a problem with a search space 2^{175} times larger than

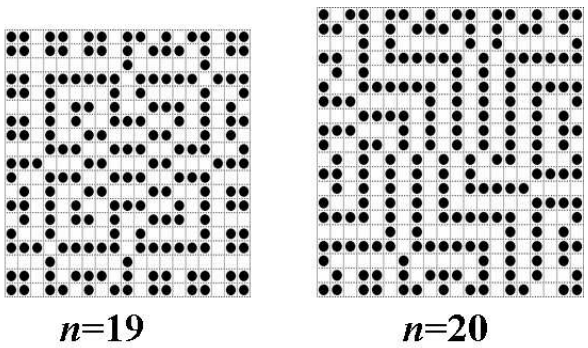

Figure 14: Maximum density still-lifes for $n = 19$ and $n = 20$.

before. Since BE scales up very regularly, we can accurately predict that it would require 4000 Gb of memory and about 7 centuries to solve the $n = 20$ instance.

Since HYB combines several techniques, it is interesting to assess the impact of each one. The seventh column reports times obtained with HYB without using mini-buckets information in the lower bound. As can be seen, the algorithm is still better than plain BE, but it performance is greatly affected. The information gathered during the preprocess improves the quality of the lower bound and anticipates pruning. Finally, the eighth column reports times obtained with HYB without having the upper bound set to $SSL(n)$. In this case we see that the importance of this technique is quite limited. The reason is that HYB, even with a bad initial upper bound, finds the optimum very rapidly and, after that moment, the quality of the initial upper bound becomes irrelevant. Figure 14 shows the optimal solutions that we obtained for $SL(19)$ and $SL(20)$.

7 Conclusions

In this paper we have studied the applicability of variable elimination to the problem of finding maximum density still-lifes. Finding still-lifes is a challenging problem and developing new solving techniques is an interesting task *per se*. Thus, the first contribution of this paper is the observation that plain variable elimination (i.e, BE) provides time complexity exponentially better than previous approaches. Empirically, we have been able to solve up to the $n = 20$ instance, with which we clearly improved previous results. The

second contribution of the paper has a deeper insight. Our algorithm uses recent techniques based on variable elimination. Since these techniques are little known and rarely applied, the results presented in this paper add new evidence on their practical importance. We have also shown that variable elimination can be used beyond the academic still-life problem by providing experimental results in some real scheduling instances.

References

Bensana, E., Lemaitre, M., & Verfaillie, G. (1999). Earth observation satellite management. *Constraints*, *4(3)*, 293–299.

Bertele, U., & Brioschi, F. (1972). *Nonserial Dynamic Programming*. Academic Press.

Bistarelli, S., Montanari, U., & Rossi, F. (1997). Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, *44*(2), 201–236.

Bosch, R., & Trick, M. (2002). Constraint programming and hybrid formulations for three life designs. In *Proceedings of the International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CP-AI-OR'02*, pp. 77–91.

Cabon, B., de Givry, S., Lobjois, L., Schiex, T., & Warners, J. (1999). Radio link frequency assignment. *Constraints*, *4*, 79–89.

Dechter, R. (1999). Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, *113*, 41–85.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann, San Francisco.

Dechter, R., & Pearl, J. (1989). Tree clustering for constraint networks. *Artificial Intelligence*, *38*, 353–366.

Dechter, R., & Fatah, Y. E. (2001). Topological parameters for time-space tradeoff. *Artificial Intelligence*, *125*(1–2), 93–118.

Dechter, R., & Rish, I. (2003). Mini-buckets: A general scheme for bounded inference. *Journal of the ACM*, *50*(2), 107–153.

Gardner, M. (1970). The fantastic combinations of john conway's new solitary game. *Scientific American*, *223*, 120–123.

Gomes, C., & Shmoys, D. (2002). Completing quasigroups or latin squares: a structured graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and Generalizations*.

Kask, K. (2000). New search heuristics for max-csp. In *Proc. of the 6^{th} CP*, pp. 262–277, Singapore. LNCS 1894. Springer Verlag.

Kask, K., & Dechter, R. (2001). A general scheme for automatic generation of search heuristics from specification dependencies. *Artificial Intelligence*, *129*, 91–131.

Korf, R. (2003). Optimal rectangle packing: initial results. In *Proc. of the 18^{th} IJCAI*, Acapulco, Mexico.

Larrosa, J., & Dechter, R. (2003). Boosting search with variable elimination in constraint optimization and constraint satisfaction problems. *Constraints*, *8*(3), 303–326.

Larrosa, J., & Schiex, T. (2003). In the quest of the best form of local consistency for weighted csp. In *Proc. of the 18^{th} IJCAI*, Acapulco, Mexico.

Mitchell, D., Selman, B., & Levesque, H. (1992). Hard and easy distributions of sat problems. In *Proc. of AAAI-92*, pp. 459–465, San Jose, CA.

Sandholm, T. (1999). An algorithm for optimal winner determination in combinatorial auctions. In *IJCAI-99*, pp. 542–547.

Schiex, T., Fargier, H., & Verfaillie, G. (1995). Valued constraint satisfaction problems: hard and easy problems. In *IJCAI-95*, pp. 631–637, Montréal, Canada.

Smith, B. (2002). A dual graph translation of a problem in life. In *Proc. of CP-2002*, pp. 0–1, Ithaca, USA. LNCS. Springer Verlag.

Smith, B., & Dyer, M. (1996). Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, *81*(1), 155–181.

Smith, B., Brailsford, S., Hubbard, P., & Williams, H. (1996). The progresive party problem: integer linear programming and constraint programming compared. *Constraints*, *1*, 119–138.