Boosting Search with Variable Elimination in Constraint Optimization and Constraint Satisfaction Problems. *

A. Javier Larrosa (larrosa@lsi.upc.es)
Universitat Politècnica de Catalunya, Barcelona, Spain.

B. Rina Dechter (dechter@ics.uci.edu)
University of California at Irvine, USA

Abstract. There are two main solving schemas for *constraint satisfaction* and *optimization* problems: *i*) *search*, whose basic step is branching over the values of a variables, and *ii*) *dynamic programming*, whose basic step is variable elimination. Variable elimination is time and space exponential in a graph parameter called induced width, which renders the approach infeasible for many problem classes. However, by restricting variable elimination so that only low arity constraints are processed and recorded, it can be effectively combined with search, because the elimination of variables may reduce drastically the search tree size.

In this paper we introduce BE-BB(k), a hybrid general algorithm that combines search and variable elimination. The parameter k controls the tradeoff between the two strategies. The algorithm is space exponential in k. Regarding time, we show that its complexity is bounded by k and a structural parameter from the constraint graph. We provide experimental evidence that the hybrid algorithm can outperform state-of-the-art algorithms in constraint satisfaction, Max-CSP and Weighted CSP. Especially in optimization tasks, the advantage of our approach over plain search can be overwhelming.

Keywords: constraint satisfaction, constraint optimization, soft constraints, bucket elimination, branch and bound.

1. Introduction.

Many problems arising in domains such as scheduling, design, diagnosis, temporal reasoning and default reasoning, can be naturally modeled as *constraint problems*. A constraint problem consists of a finite set of *variables*, each associated with a finite *domain* of values, and a set of *constraints*. In a constraint *satisfaction* problem (CSP), constraints are relations indicating permitted tuples. A *solution* is an assignment of a value to every variable such that all constraints are satisfied. In a constraint *optimization* problem (COP), some constraints (called *soft*) are cost functions indicating preferences. The task of interest is to find a complete assignment satisfying hard constraints and minimizing the

* Research funded by the Spanish CICYT project TAP1999-1086-C03-03, by the NSF grant IIS-0086529 and by MURI ONR award N00014-00-1-0617.

global cost. Solving constraint problems is NP-*hard*. Therefore, general algorithms are likely to require exponential time in the worst-case.

Most complete algorithms for solving constraint problems belong to one of the two following schemas: *search* and *dynamic programming*. *Search* algorithms transform a problem into a set of subproblems. This is normally done by selecting a variable and considering the assignment of each of its domain values. The subproblems are solved applying recursively the same transformation rule. These algorithms generate a tree that is normally traversed in a depth-first manner, which has the benefit of being space linear. In constraint optimization, search follows a *branch and bound* (BB) schema [21]; in constraint satisfaction, search can be specialized to handle and propagate relations [15, 23]. In the worst-case, search algorithms need to explore the whole search tree. Nevertheless, in practice they typically do much better.

Dynamic programming algorithms solve a problem by a sequence of transformations that reduce the problem size, while preserving the value of the best cost attainable in the problem [2]. *Bucket Elimination* (BE) [8] is a complete algorithm that relies on the basic step of *variable elimination*. The algorithm proceeds by selecting one variable at a time and replacing it by a new constraint which summarizes the effect of the chosen variable. Once all variables have been eliminated, the best cost is computed and the corresponding assignment is obtained in a backtrack-free manner. The main drawback of BE is that new constraints may have large arities which are exponentially hard to process and store. The exponential space complexity limits severely the algorithm's usefulness. However, a nice property of BE is that its worst-case time and space complexities can be tightly bounded by a structural parameter of the problem called induced width.

In this paper we propose a general solving schema BE-BB(k) which combines branch and bound search and variable elimination in an attempt to exploit the best of each. The algorithm selects a variable and attempts its elimination, but this is only done when the elimination generates a small arity constraint. Otherwise, it switches to search. Namely, it branches on the variable and transforms the problem into a set of smaller subproblems where the process is recursively repeated. Parameter k controls the trade-off between variable elimination and search. The space complexity of BE-BB(k) is exponential in k. The time complexity is exponential in k and a refined structural parameter of the problem.

Our approach is applicable to many search strategies and a variety of tasks. In this paper, we focus the presentation on constraint optimization, although we report experimental results on both satisfaction and optimization problems. In all cases, we show that a bounded form of variable elimination may boost search, while never having a worsening effect. For optimization tasks, the advantage of our approach over plain search can be overwhelming.

This paper is organized as follows: The next Section introduces notation and necessary background. In Section 3 we describe BE-BB(k). In Section 4 we discuss the algorithm time and space complexity. In Section 5 we provide experimental results demonstrating the practical usefulness of our approach. Section 6 discusses related work and Section 7 concludes.

2. Preliminaries

2.1. Constraint Satisfaction and Optimization

A *constraint satisfaction problem* (CSP) consists of a set of variables $X = \{x_1, \ldots, x_n\}$, a set of domains $D = \{D_1, \ldots, D_n\}$ and a set of constraints $C = \{R_1, \ldots, R_m\}$. Domain D_i is a finite set of values that can be assigned to variable x_i. A constraint R is a *relation* over a subset of the variables $var(R) \subseteq X$ called the *scope* of R. Tuples in R denote the legal combinations of assignments to variables in its scope. A *solution* is an assignment of a value to every variable in X such that every constraint is satisfied. Finding a solution to a CSP is an NP-*complete* problem. The *arity* of a constraint is the size of its scope. The arity of a problem is the maximum arity over its constraints. In the sequel, n, d, m and r will denote the number of variables, the largest domain size, the number of constraints and the problem arity, respectively.

In a *constraint optimization problem* (COP), some constraints (called *soft* constraints) are *cost functions* denoting *preferences* among tuples. A cost function f is defined over its scope $var(f)$ and returns for each tuple a non-negative cost. Cost functions can be expressed intensionally as mathematical functions or computable procedures, or extensionally as tables of costs. Without loss of generality, we assume that all constraints (*i.e.*, soft and hard) are represented as cost functions. Hard constraints are bi-valued functions assigning cost zero to allowed tuples and cost infinity to forbiden tuples. We consider the *weighted* CSP (WCSP) model [26], where the objective function is the sum of all constraints $C = \{f_1, \ldots, f_m\}$,

$$f^*(X) = \sum_{j=1}^{m} f_j(X)$$

Thus, the cost of a complete assignment is the sum of costs given by the set of constraints. Observe that we use the simplified notation $f_j(X)$, where we mean f_j evaluated on X projected over $var(f_j)$.

The goal is to find a *complete assignment with minimum cost* (the maximization task is analogous). Solving a WCSP is NP-*hard*. We assume that evaluating f_j is time $O(1)$.

More general frameworks for COP have been presented in [5, 26] where different semantics can be given to costs (e.g, *probabilities*, *possibilities*, *fuzzy logic* values, *etc*). The WCSP model captures the algorithmic difficulty of the general case and our approach can be easily extended.

Given a constraint problem, its *constraint graph* G associates each variable with a node and connects any two nodes whose variables appear in the scope of the same (hard or soft) constraint. The *induced graph* of G relative to an ordering o of its variables, denoted $G^*(o)$, is obtained by processing the nodes in reverse order of o. For each node all its earlier neighbors are connected, including neighbors connected by previously added edges. 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*, denoted $w^*(o)$, is the maximum width of nodes in the induced graph. Finding the ordering o with minimum $w^*(o)$ is an NP-*complete* problem [1].

2.2. Branch and Bound

Branch and Bound (BB) is a *search* schema for COP solving [21, 13]. It traverses the search tree defined by the problem, where internal nodes represent incomplete assignments and leaf nodes stand for complete ones, which may or may not be optimal. During the traversal, which is typically depth-first, BB 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 t. This lower bound function is called the *heuristic function* (h') in heuristic search terminology.

The basic step of BB is *branching*. Namely, the algorithm selects a variable x_i and transforms the current problem into a set of subproblems, one for each value in D_i, extending the current assignment t (*i.e.*, branches on the variable alternatives). In depth-first search,

```
function BB(t, P, F)
1.  if F = ∅ then
2.      UB ← ComputeCost(t);
3.      BestTuple ← t;
4.  else
5.      x_i ← SelectVariable(F);
6.      while D_i ≠ ∅ do
7.          a ← PopValue(D_i);
8.          t ← Append(t, (x_i, a));
9.          LookAhead(t, P ∪ {x_i}, F − {x_i});
10.         if no empty domain then
11.             BB(t, P ∪ {x_i}, F − {x_i});
endfunction
function LookAhead(t, P, F)
12. for each x_j ∈ F, b ∈ D_j do
13.     t' ← Append(t, (x_j, b));
14.     if LB(t') ≥ UB then prune(j, b);
endfunction
```

Figure 1. Depth-first Branch and Bound with look-ahead for COP solving. t is the current assignment, P is the set of past variables, F is the set of future variables.

subproblems are considered sequentially and the same branching rule is recursively applied. The constraint graph of each child is obtained by removing node x_i and its outgoing edges from the parent graph.

Figure 1 shows a generic *depth-first* BB which is enhanced with a *look-ahead* process similar to that of *forward checking* [15] in constraint satisfaction. For the sake of simplicity, context restoration upon backtracking is omitted. At a given node, P and F denote the sets of *past* and *future* variables, respectively. If the set F is empty, the current assignment t improves over the current best solution, so the upper bound UB is updated (lines 2 and 3). Else, it selects a variable x_i (line 5) and iterates over its values. When considering a value a, the look-ahead procedure (called in line 9) iterates over every value b of every future variable x_j. It computes t', the extension of the current assignment t with (x_j, b) (line 13), and checks its feasibility (line 14). If the check fails, b is pruned from the current domain of x_j. If the look-ahead causes an empty domain, the next value of D_i is attempted. Else, a recursive call is made (line 11) in which the current assignment t is extended with (x_i, a).

2.2.1. *Lower Bound Computation*

Different lower bounds can be used within BB (in line 14). A simple approach is to consider the best cost that can be attained from each constraint, subject to the assignment.

$$LB(t) = \sum_{f \in C} \min_q \{f(t,q)\} \tag{1}$$

where C is the set of constraints and $\min_q\{f(t,q)\}$ denotes the minimum cost extension of t to variables in $var(f)$ not assigned in t.

The time complexity of computing expression (1) is $O(m \cdot d^{r-1})$, where m is the number of constraints, d is the domain size and r is the problem arity. This lower bound is impractical with high arity problems. It is possible to bound the complexity by restricting the set of considered constraints to $C^s = \{f \in C| \; |var(f) \cap F| \leq s\}$, the set of constraints having at most s uninstantiated variables in their scope given the current assignment. This idea, already suggested in [22], yields the following expression,

$$LB(t) = \sum_{f \in C^s} \min_q \{f(t,q)\} \tag{2}$$

The time complexity of computing lower bound (2) is $O(m \cdot d^s)$. More sophisticated lower bounds can be found in [28, 20, 25, 9, 16, 18].

The space complexity of depth-first BB is linear. The time complexity is bounded by the product of the search space size d^n and the complexity per node L, $O(d^n L)$.

2.3. Bucket Elimination

Bucket Elimination (*BE*) [8] is an algorithm for COP solving which falls into the category of dynamic programming methods [2]. It is based upon the following two operators over functions:

- 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,

$$(f+g)(X) = f(X) + g(X)$$

- The *elimination* of variable x_i from f, denoted $\text{elim}_i(f)$, is a new function with scope $var(f) - \{x_i\}$ which returns for each tuple the minimum cost extension to f,

$$(\text{elim}_i(f))(X) = \min_{a \in D_i} \{f(X,(x_i,a))\}$$

where $f(X, (x_i, a))$ means $f(X)$ with variable x_i set to value a. 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* $(f+g)(x_1, x_2, x_3) = x_1+x_2+x_1x_3$. *If domains are integers in the interval* $[1..10]$, *the elimination of* x_1 *from* f *is* $(\mathrm{elim}_1(f))(x_2) = 1 + x_2$. *The subsequent elimination of* x_2, *produces constant* 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 stores intermediate results extensionally in tables, which causes its high space complexity.

Given an arbitrary variable ordering o, BE partitions the set of constraints into *buckets*. There is a bucket B_i for each variable x_i and it contains all constraints having x_i in their scope as their highest variable (according to the ordering o). The algorithm processes variables one by one, from last to first. For each variable x_i, the algorithm *infers* a new constraint f_i that summarizes the effect of the variable on the rest of the problem. Variable x_i is eliminated and f_i is added to the the bucket of the highest variable in its scope. The addition of f_i *compensates* the deletion of x_i preserving the value of the problem optimal cost. The constraint f_i is computed by summing all constraints in B_i and subsequently eliminating x_i.

$$f_i \leftarrow \mathrm{elim}_i(\sum_{f \in B_i} f) \tag{3}$$

The elimination of the last variable produces an empty-scope constraint (*i.e.*, a constant function) which is the optimal cost of the problem. An assignment of variables producing the optimal cost is generated in a backtrack-free manner as follows: Variables are assigned from first to last according to o. The value for x_i is the best extension of the assignment to $(x_1, \ldots, x_{i-1})$ with respect to the set of constraints in B_i.

Figure 2 shows a recursive description of BE. The algorithm is typically described iteratively, but the recursive version facilitates the integration with BB that will be introduced in the following section. At a given point of the recursive execution, there is a set F of unprocessed or *future* variables, and a set CC of *current constraints* defined over F which define the current subproblem (the initial call has $F = X$ and $CC = C$). If F is empty, CC contains a constant function with the best cost, which is recorded (line 2). Else, BE selects a variable x_i from F

function BE(F, CC)
1. **if** $F = \emptyset$ **then** $\{CC$ contains a constant function $f\}$
2. $\quad BestCost \leftarrow f$
3. **else**
4. $\quad x_i \leftarrow$ `SelectVariable`(F);
5. $\quad B_i \leftarrow \{f \in C | x_i \in var(f)\}$
6. $\quad f_i \leftarrow \mathrm{elim}_i(\sum_{f \in B_i} f)$;
7. $\quad CC \leftarrow CC \cup \{f_i\} - B_i$;
8. $\quad$ BE$(F - \{x_i\}, CC)$;
9. $\quad x_i \leftarrow$ best extension of the assignment to $(x_1, \ldots, x_{i-1})$ relative to $\sum_{f \in B_i} f$

Figure 2. Recursive description of Bucket Elimination. F is the set of future (unprocessed) variables and CC is the set of constraints in the current subproblem.

(line 4), which must be the last variable in F according to the ordering o. Next, it generates its bucket B_i as the set of constraints in CC mentioning x_i (line 5). After that, the new constraint f_i is computed (line 6). Then, constraints in B_i are replaced in CC by f_i (line 7) forming the new subproblem with which BE is recursively called (line 8). The optimal assignment is constructed as recursive calls terminate. When the current call is over (*i.e:* line 8 is executed), the algorithm knows the optimal assignment to variables in $F - \{x_i\}$. The algorithm then computes the assignment to the current variable x_i (line 9) and returns to the previous call.

Observe that the elimination of a variable x_i generates a new constraint f_i whose scope is the set of neighbors of x_i in the current problem's graph. Thus, the arity of f_i is the degree of x_i in the current constraint graph, dg_i. Computing f_i is time $O(d^{dg_i+1})$, storing f_i is space $O(d^{dg_i})$. The addition of f_i to the current problem changes the current constraint graph by connecting all neighbors of x_i. It can be shown that for all $i, dg_i \leq w^*(o)$. Therefore, the complexity of BE along ordering o is time $O(n \cdot d^{w^*(o)+1})$ and space $O(n \cdot d^{w^*(o)})$, both exponential in the induced width of the ordering [8]. The space complexity of BE is its main drawback, since it can only be applied in practice to problems with an identified low induced width ordering.

3. Combining Variable Branching and Variable Elimination.

In this section we introduce BE-BB(k), a hybrid schema that combines variable elimination and branching. Let us suppose that we have a problem that we cannot solve with BE due to its high induced width.

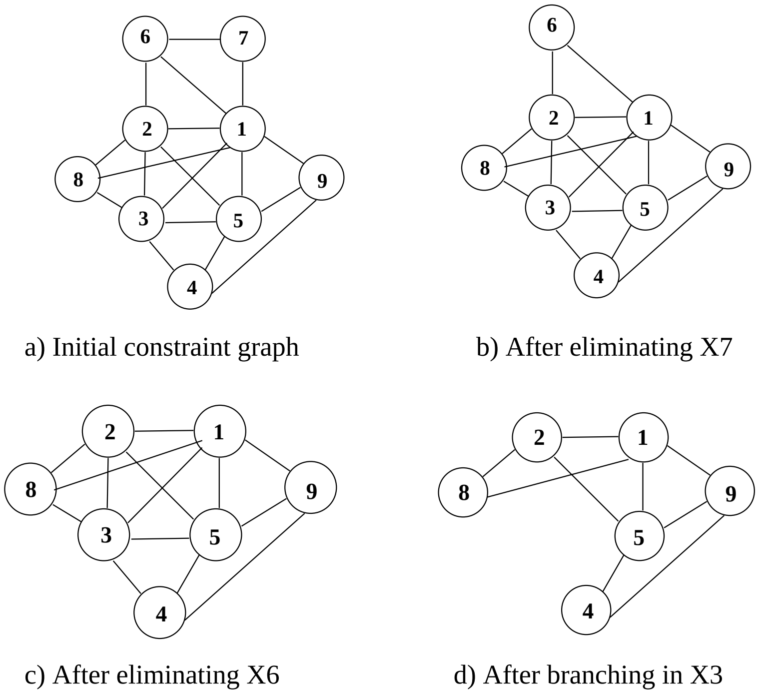

Figure 3. A constraint graph and its evolution over a sequence of variable eliminations and branchings.

We can still use BE as a pre-process and eliminate those variables whose elimination is not too costly. This will transform the problem into an equivalent one having a smaller set of variables X'. Subsequently, we can solve the reduced problem with plain BB. Once the search is over, we have the best cost of the reduced problem (which is also the best cost of the original problem) and the corresponding assignment to variables in X' which can be extended to X in a backtrack-free manner. The recursive application of this idea is the basis of BE-BB(k).

EXAMPLE 2. *Consider a COP whose constraint graph is depicted in Figure 3.a. Suppose that we want to eliminate a variable but we do not want to compute and store constraints with arity higher than two. Then we can only take into consideration variables connected to at most two variables. In our example, variable x_7 is the only one that can be selected. Its elimination transforms the problem into another one whose constraint graph is depicted in Figure 3.b. Now x_6 has its degree decreased to two, so it can also be eliminated. The new constraint graph is depicted in Figure 3.c. At this point, every variable has degree greater than two, so we switch to a search schema which selects a variable, say x_3, branches over its values and produces a set of d subproblems, one for* *each value in its domain. All of them have the same constraint graph, depicted in Figure 3.d. For each subproblem, it is possible to eliminate variable x_8 and x_4. After their elimination it is possible to eliminate x_2 and x_9, and subsequently x_5 and x_1. Eliminations after branching have to be done at every subproblem since the new constraints with which the eliminated variables are replaced differ from one subproblem to another. After processing all the variables (x_3 assigned, the rest eliminated), it is possible to compute at each subproblem its best cost subject to its assignment to x_3. If this cost improves the current upper bound, it is updated. The complete assignment responsible for this currently optimal cost can be computed as follows: variables are considered reversing the order in which they were either assigned or eliminated, that is, $x_1, x_5, x_9, x_2, x_4, x_8, x_3, x_6, x_7$. Eliminated variables are assigned as in BE and assigned variables take the value assigned in the subproblem path. In the example, only one branching has been made. Therefore, the elimination of variables has reduced the search space size from d^9 to d, where d is the size of the domains.*

In the example, we bounded the arity of the new constraints to two. However, in general our algorithm has a parameter k that bounds the arity of the new constraints. This parameter ranges from -1 to $n-1$ and controls the tradeoff between variable elimination and branching. Low values of k only allow the recording of small arity constraints which are efficiently computed and stored. However, they may allow substantial search. On the other hand, high values of k allow recording high arity constraints. It leads to substantial reduction of the search space, at the cost of processing and recording high arity constraints.

In the extreme case that k is set to -1, the algorithm never eliminates any variable and therefore performs plain BB. When k is set to 0, only disconnected variables are eliminated (they are replaced by constant functions). If k is set to a sufficiently high value, every variable elimination is permitted, so the algorithm becomes BE.

Figure 4 describes BE-BB(k). At an arbitrary call, the set of variables X is partitioned into three sets: assigned P, future F and eliminated E variables. Each recursive call receives the current subproblem defined by the current assignment t, the current sets P, F and E, and the current set of constraints CC (as in BE, the set of constraints changes during the execution). Constraints in CC are defined over $P \cup F$. In the initial call $t = \emptyset$, $F = X$, $E = P = \emptyset$ and $CC = C$.

If BE-BB(k) is called with an empty set of future variables, the current branch of search improves over the current best solution. Therefore, UB is updated with the cost of t and the current best assignment is generated by processing variables in the opposite order in which they

function BE-BB(t, P, E, F, CC)
1. **if** $F = \emptyset$ **then**
2. $UB \leftarrow$ `ComputeCost`(t);
3. $BestTuple \leftarrow$ extend t to variables in E;
4. **else**
5. $x_i \leftarrow$ `SelectVariable`(F);
6. $B_i \leftarrow \{f \in CC | \ x_i \in var(f)\}$;
7. $dg_i \leftarrow | \cup_{f \in B_i} (var(f) \cap F)|$;
8. **if** $dg_i \leq k$ **then** `VarElim`(t, P, E, F, CC, x_i)
9. **else** `VarBranch`(t, P, E, F, CC, x_i)

endfunction

function `VarElim`(t, P, E, F, CC, x_i)

10. $f_i \leftarrow \text{elim}_i(\sum_{f \in B_i} f)$;
11. $CC \leftarrow CC \cup \{f_i\} - B_i$;
12. **if** $LB(t, CC) \leq UB$ **then** BE-BB$(t, P, E \cup \{x_i\}, F - \{x_i\}, CC)$;

endfunction

function `VarBranch`(t, P, E, F, CC, x_i)

13. **while** $D_i \neq \emptyset$ **do**
14. $a \leftarrow$ `PopValue`(D_i);
15. $t \leftarrow$ `Append`$(t, (x_i, a))$;
16. `LookAhead`$(t, P \cup \{x_i\}, F - \{x_i\}, CC)$;
17. **if no** `empty domain` **then**
18. BE-BB$(t, P \cup \{x_i\}, E, F - \{x_i\}, CC)$;

endfunction

Figure 4. Algorithm BE-BB. t is the current assignment, P is the set of past (assigned) variables, E is the set of eliminated variables, F is the set of future variables, CC is the set of constraints relevant to the current subproblem.

were selected (lines 2 and 3). Values for eliminated variables E have to be computed, which requires the algorithm to access the buckets processed along the *current* branch of search. This process is omitted in Figure 4 for clarity reasons.

If there are future variables, BE-BB selects the current variable x_i from F (line 5). This selection can be done using any heuristic criterion, either static or dynamic. Then, the current bucket B_i is computed as the set of constraints in CC mentioning x_i in their scope (line 6). Note that these constraints may have past and future variables in their scope, due to the assignments previously made by BB. Next, the algorithm computes dg_i, the number of neighbors of x_i in the current problem, as the number of future variables in the scopes of constraints in B_i (line 7). If dg_i is not larger than the control parameter k, x_i is eliminated (`VarElim` is called). Procedure `VarElim` transforms the current sub problem into an equivalent problem (the value of the optimal cost is preserved) where the current variable is missing. Else, the algorithm branches on its values (`VarBranch` is called). Procedure `VarBranch` transforms the current subproblem into a sequence of problems and attempts to improve UB with each of them. Either way, x_i is removed from the set of future variables and BE-BB is recursively called.

The elimination of x_i in `VarElim` is slightly different from the one in BE (Figure 2), because constraints in B_i may have past assigned variables in their scope. The difference is that past variables are replaced by their assigned value when computing the new constraint.

EXAMPLE 3. *Let* $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$ *and* $g(x_1, x_3) = x_1 x_3$ *be the current set of constraints. Suppose all domains are integers in the interval* $[1..10]$ *and* x_1 *is a past variable with value* 5*. The elimination of* x_3 *introduces the following constraint:*

$$\text{elim}_3(f+g)(x_2) = \min_{x_3}\{5+x_2+x_3+5x_3\} = 5+x_2+1+(5\cdot 1) = 11+x_2$$

Observe that the elimination of a variable modifies the current set of constraints (line 11). As a consequence, the current lower bound used by the branch and bound algorithm may be increased. That is the reason for the feasibility test previous to the recursive call (line 12).

4. Complexity Analysis.

BE-BB(k) stores constraints of arity at most k which require $O(d^k)$ space. It only keeps constraints added in the current search path, so there are at most n simultaneously stored constraints. Therefore, BE-BB(k) has $O(n \cdot d^k)$ space complexity. Regarding time, the algorithm visits at most d^n nodes, because in the worst-case it performs plain search on a tree of depth n and branching factor d. Clearly, this worst-case bound for the search space size is loose since it ignores the search space reduction caused by variable eliminations. It is possible to obtain a more refined upper bound for the number of visited nodes, if we assumed that BE-BB(k) is executed with a static variable ordering o. The bound is based on the following definition:

DEFINITION 1. *Given a constraint graph* G *and an ordering* o *of its nodes, the* k-restricted induced graph *of* G *relative to* o*, denoted* $G^*(o, k)$*, is obtained by processing the nodes in reverse order from last to first. For each node, if it has* k *or fewer earlier neighbors (taking into account old and new edges), they are all connected, else the node is ignored. The number of nodes in* $G^*(o, k)$ *with width greater than* k

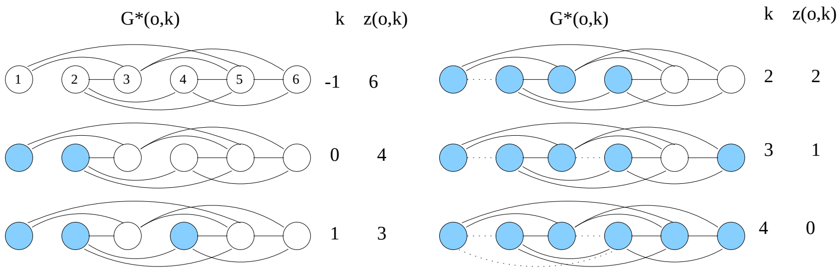

Figure 5. An ordered constraint graph G and its k-*restricted induced graphs* $G^*(o,k)$ for $-1 \leq k \leq 4$. Note that $G = G^*(o,-1)$. Ordering o is lexicographic.

is denoted $z(o,k)$*. The number of nodes in* $G^*(o,k)$ *with width less than or equal to* k *is denoted* $e(o,k)$*.*

In what follows we assume that BE-BB(k) is executed with a static variable ordering o and that it selects the variables *from last to first.* The k*-restricted induced graph* $G^*(o,k)$ can be used to synthesize the search space traversed by the algorithm. The nodes in $G^*(o,k)$ having width less than or equal to k are exactly the variables that BE-BB(k) eliminates. The edges added during the computation of $G^*(o,k)$ reflect the scopes of constraints that the algorithm adds when it performs variable elimination. The nodes that are ignored during the computation of $G^*(o,k)$ have width greater than k and thus correspond to the branching variables.

PROPOSITION 1. *BE-BB(k) with a static variable ordering o, runs in time* $O(d^{z(o,k)} \times (L + e(o,k)d^{k+1}))$ *where* L *is the cost per node of the branch and bound algorithm used.*

Proof. The algorithm branches in $z(o,k)$ variables with a branching factor of d, so the search space size is bounded by $d^{z(o,k)}$. Processing each visited node takes cost L, and at most $e(o,k)$ variable eliminations. Eliminating a variable with up to k neighbors takes $O(d^{k+1})$. Therefore, the total cost per node is $O(L + e(o,k)\ d^{k+1})$. Multiplying this by the total number of nodes, $d^{z(o,k)}$, we obtain the total time complexity. □

Proposition 1 shows that BE-BB(k) is exponential in $k + z(o,k)$. Increasing k is likely to decrease $z(o,k)$, which means that less search takes place at the cost of having more expensive (in time and space) variable elimination. Figure 5 illustrates this fact. Given an ordered constraint graph G (Figure 5 top left), we can see how $G^*(o,k)$ changes as k varies from -1 to 4 (note that $G = G^*(o,-1)$ because no edge can be possibly be added). The value of k appears at the right-hand side of each graph, along with the corresponding $z(o,k)$. Grey nodes are those

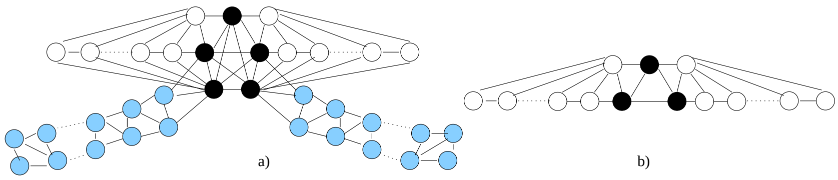

Figure 6. A constraint graph and a subgraph after the elimination of grey variables and branching on two black variables.

with width less than or equal to k and correspond to the variables that BE-BB(k) eliminates. Dotted edges are those added during the computation of $G^*(o, k)$. White nodes are those with width greater than k and correspond to branching variables. For example, when $k = 1$, variable 6 is branched first, next variable 5 is branched, then variable 4 is eliminated, and so on. The space requirements, search space size and effort per node as k varies is depicted in the following table.

k	$z(o, k)$	space	search space size	effort per node
-1	6	0	d^6	L
0	4	6 d^0	d^4	$L + 2\ d^1$
1	3	6 d^1	d^3	$L + 3\ d^2$
2	2	6 d^2	d^2	$L + 4\ d^3$
3	1	6 d^3	d^1	$L + 5\ d^4$
4	0	6 d^4	d^0	6 d^5

The time complexity of the algorithm suggests a class of problems for which it is likely to be effective. Namely, problems having a subset of the variables highly connected while the rest have low connectivity. The highly connected part renders BE infeasible. Similarly, a search procedure will branch on the low connectivity variables generating a huge search space. BE-BB(k) with a low k will eliminate the low-connectivity variables and only search on the dense subproblems.

EXAMPLE 4. *Consider a problem having the constraint graphs of Figure 6.a. There is a clique of size 5 (black nodes), which means that the complexity of BE is at least $O(nd^4)$ and $O(nd^5)$ space and time, respectively. On the other hand, BB has time complexity $O(d^n L)$. Algorithm BE-BB(k) with $k = 2$, if provided with the appropriate variable ordering eliminates all grey nodes before any branching. Subsequently, it may branch on the two bottom nodes of the clique producing d^2 subproblems. The subgraph of the subproblems is depicted in Figure 6.b. At this point, the problems can be completely solved with variable elimination. Thus, the space complexity of the process is $O(n\ d^2)$, the search space size is*

$O(d^2)$ and the time complexity is $O(d^2 \times (L + (n-2)d^3))$. So we are able to achieve time bounds close to BE with a lower $O(n\ d^2)$ space.

5. Empirical Results.

5.1. Random Problems

In this section we show the potential of our approach on randomly generated problems. We consider three different tasks: CSP, Max-CSP (*i.e.*, WCSP with 0/1 costs) and non-binary WCSP.

We use a random problem model which extends the well-known four-parameters model [27, 14] to non-binary soft constraints. A random WCSP class is characterized by 6 parameters: $\langle n, d, r, v, m, t\rangle$ where n is the number of variables, d the number of values per variable, r is the arity of all constraints, v is the number of different non-zero costs, m the number of constraints (problem *density* or *connectivity*) and t is the number of tuples in the constraint with non-zero cost (problem *tightness*). The scope of every constraint is randomly selected. For each constraint, t tuples with non-zero cost are randomly selected and their cost is randomly assigned within the interval $[1..v]$. Problems with different constraints having the same scope, as well as problems with a disconnected constraint graph, are discarded. When $r = 2$ and $v = 1$, our model is equivalent to the well-known random CSP model [27, 14]. In binary problems (*i.e.*, $r = 2$), we can characterize the *average degree* of variables in a class as $\overline{dg} = 2m/n$.

When we write an interval $[k..k']$ instead of a fixed parameter, we denote a sequence of random problem classes. For instance, $\langle n, d, r, v, m, [t..t']\rangle$ denotes the sequence of classes from $\langle n, d, r, v, m, t\rangle$ to $\langle n, d, r, v, m, t'\rangle$, where the five first parameters are fixed and the tightness varies from t to t'.

In all the experiments, the following dynamic variable ordering heuristic is used: Find the variable x_i with the lowest degree. If its degree is less than or equal to k, variable x_i is selected (it will be eliminated). Else, compute for each variable the ratio domain size divided by degree. The variable with the lowest ratio is selected (it will be branched) [4]. In the first two sets of experiments (CSP and Max-CSP), samples have 50 instances. In the third set of experiments (WCSP), samples have 25 instances.

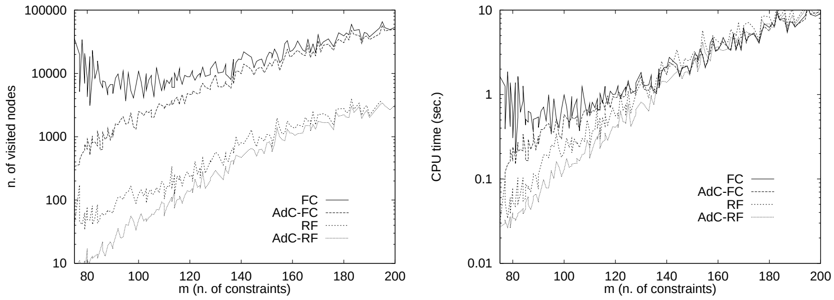

Figure 7. Average search effort of four algorithms on the classes $\langle 50, 10, 2, 1, [75..200], t^* \rangle$. Mean number of visited nodes and CPU time is reported. Note that plot curves come in the same top-to-bottom order than legend keys.

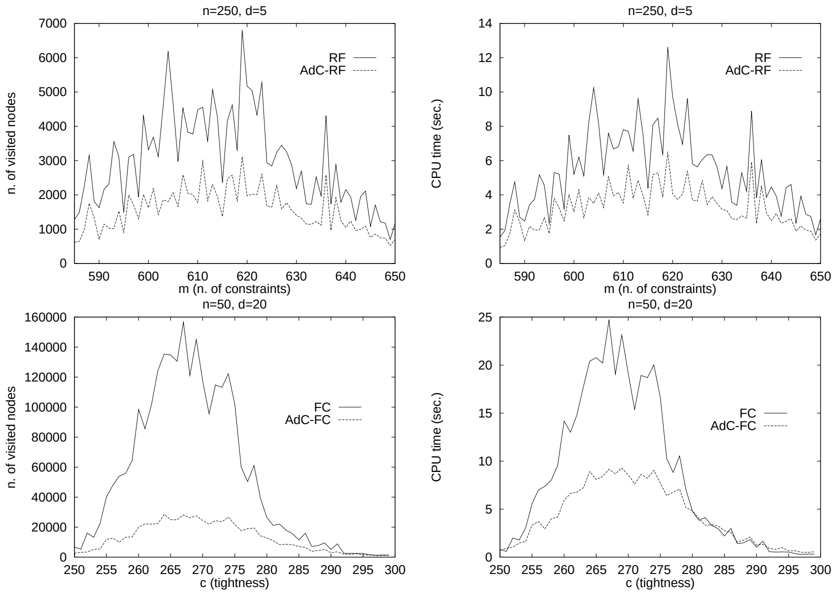

Figure 8. Experimental results on the classes $\langle 250, 5, 2, 1, [585..650], t^* \rangle$ and $\langle 50, 20, 2, 1, 125, [250..300] \rangle$. Mean number of visited nodes and CPU time is reported.

5.1.1. *Binary Constraint Satisfaction (CSP)*

Our first set of experiments considers binary CSP. In constraint satisfaction, it is possible to specialize BB and BE to handle hard constraints in a more efficient way. Therefore, we consider *forward checking* (FC) [15] and *really full look ahead* (RF) [23] as our reference search algorithms and *adaptive consistency* (AdC) [10] as the variable elimination algorithm. Thus, our approach leads to two parameterized algorithms: AdC-FC(k) and AdC-RF(k) (see [17] for details). In this experiment, we compare the performance of the algorithms with

$k = -1$ and $k = 2$. When $k = -1$ the algorithms are FC and RF, respectively. When $k = 2$ the algorithms eliminate variables with up to two neighbors (we refer to them as AdC-FC and AdC-RF).

Our implementation of RF is based on AC6 [3]. Since the overhead of AC6 cannot be fairly evaluated in terms of consistency checks, we consider the CPU time as the main computational effort measure. However, we also report the implementation independent measure of number of visited nodes.

In our first experiment we ran the four algorithms on the classes $\langle 50, 10, 2, 1, [75..200], t^* \rangle$, where t^* denotes the *cross-over* tightness (tightness that produces 50% satisfiable problems and 50% unsatisfiable) or the closest approximation. In this set of problems $\overline{dg}$ increases with the varying parameter, ranging from 3 to 8. Figure 7 reports the average number of visited nodes and average CPU time for each algorithm (note the *log* scale). We observe that AdC-FC (resp. AdC-RF) clearly outperforms FC (resp. RF) both in terms of nodes and CPU time. The maximum gain is observed in the most sparse problems where AdC-FC (resp. AdC-RF) can be 2 or 3 times faster than FC (resp. RF). As a matter of fact, in the most sparse problems variable elimination usually does most of the work and only a few variables are branched. As problems become more connected, the gain decreases. However, typical gains in problems with $\overline{dg}$ around 5 (*i.e.*, 125 constraints) are still significant (20% to 40%). As problems approach $\overline{dg} = 8$, it becomes less and less probable to find variables with degree less than or equal to 2. Therefore, the gain of AdC-FC and AdC-RF gracefully vanishes.

Next, we investigate how the algorithms scale up, while keeping the average degree around 5. The four algorithms were executed in the classes: $\langle 250, 5, 2, 1, [585..650], t^* \rangle$ and $\langle 50, 20, 2, 1, 125, [250..300] \rangle$. In the first set of problems, FC performed very poorly compared to RF, in the second it was RF which performed poorly compared to FC, so we report the results with the best algorithm for each class, only. Figure 8(top) reports the average results of executing RF and AdC-RF in the 250 variables problems. The superiority of AdC-RF is again apparent in this class of problems. AdC-RF is always faster than RF and the gain ratio varies from 1.2 to 3. Figure 8(bottom) reports the average results of executing FC and AdC-FC in the 50 variable problems. AdC-FC is also clearly faster than FC. At the complexity peak, AdC-FC is 2.5 times faster than FC.

5.1.2. *Binary Maximal Constraint Satisfaction (Max-CSP)*

In our second set of experiments we consider a simple optimization task: finding the assignment that violates the least number of constraints in over-constrained binary CSP. This task is known as Max-CSP [13].

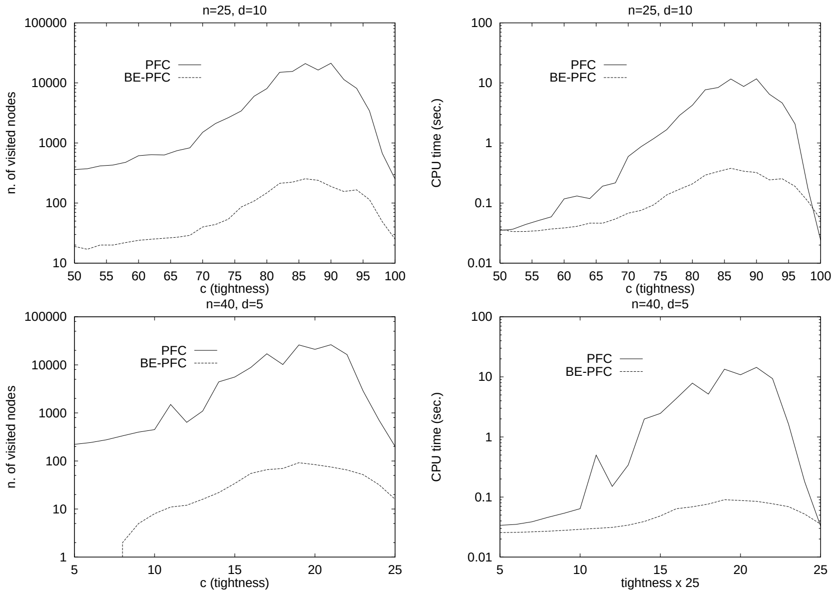

Figure 9. Average search effort of PFC and BE-PFC(2) on $\langle 25, 10, 2, 1, 37, [50..100]\rangle$ and $\langle 40, 5, 55, [5..25]\rangle$. Mean number of visited nodes and CPU time is reported.

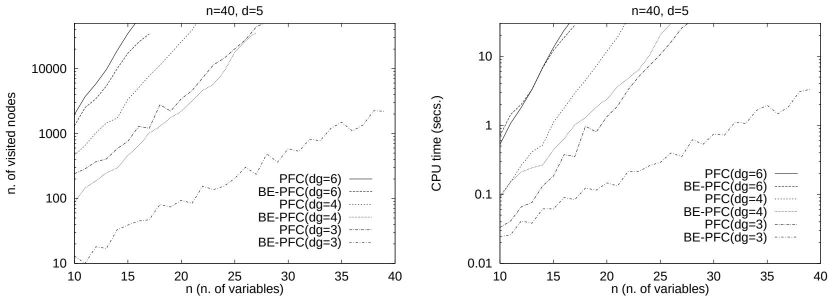

Figure 10. Average search effort of PFC and BE-PFC(2) on three classes $\langle [10..39], 10, 2, 1, 1.5n, 85\rangle$, $\langle [10..27], 10, 2, 1, 2n, 85\rangle$ and $\langle [10..17], 10, 2, 1, 3n, 85\rangle$. Mean number of visited nodes and CPU time is reported. Note that plot curves come in the same top-to-bottom order than legend keys.

There are a number of specialized BB algorithms for binary instances of this problem that perform orders of magnitude better than the BB presented in Subsection 2.2. We consider PFC-MRDAC [19], a branch and bound algorithm that uses arc-inconsistencies to improve the algorithm's pruning capabilities. We compare plain search PFC-MRDAC with the hybrid BE-PFC-MRDAC (PFC and BE-PFC, for short).

In our first experiment on Max-CSP we select the same class of sparse random problems that was used in [19] to prove the superiority of PFC over other competing algorithms. Namely, the classes

$\langle 25, 10, 2, 1, 37, c\rangle$ and $\langle 40, 5, 2, 1, 55, c\rangle$. Figure 9 reports the average results. As can be seen, the superiority of BE-PFC over PFC is impressive both in terms of nodes and CPU time (note the *log* scale). BE-PFC is sometimes 30 times faster than PFC and can visit 100 times fewer nodes in the 25 variable problems. In the 40 variable problems, the gain is even greater. BE-PFC is up to 130 times faster and visits nearly 300 times fewer nodes than PFC in the hardest instances. As a matter of fact, BE-PFC can solve the instances with lowest tightness without any search at all.

The problem classes of the previous experiment are very sparse (the average degree is below 3), which favors our approach. So the next experiment considers denser problems. The two algorithms are executed in the three following sequences of classes: $\langle [10..39], 10, 2, 1, 1.5n, 85\rangle$, $\langle [10..27], 10, 2, 1, 2n, 85\rangle$ and $\langle [10..17], 10, 2, 1, 3n, 85\rangle$. In these problem sets, we increase the number of variables and constraints proportionally, so the average degree remains fixed ($\overline{dg}$ is 3, 4 and 6, respectively). Tightness 85 guarantees that all classes are overconstrained. Figure 10 reports the search measures obtained in this experiment. Again, the superiority of BE-PFC is crystal clear. However, as could be expected, the gain decreases as $\overline{dg}$ increases. In the problems with $\overline{dg} = 3$, BE-PFC is up to 70 times faster and visits up to 300 fewer nodes. In the problems with $\overline{dg} = 4$, the gain ratio of the hybrid algorithm is 9 in terms of time and 18 in terms of visited nodes. In the problems with $\overline{dg} = 6$, both algorithms are very close. PFC is slightly faster in the smallest instances and BE-PFC is 30% faster in the largest instances. Regarding visited nodes, the gain of BE-PFC ranges from 40% in the smallest instances, to 250% in the largest. The plots also indicate that the advantage of BE-PFC over PFC seems to increase with the size of the problems if the average degree remains fixed.

5.1.3. *Non-binary WCSP*

In our third experiment, we consider non-binary WCSP. We take BE-BB(k) as described in Section 3 and analyze the effect of varying k. Regarding the lower bound, we use the parameterized LB (2) described in Subsection 2.2 which only takes into account constraints with at most s uninstantiated variables. Thus, we have a two parameters algorithm BE-BB(k, s). Note that BE-BB($-1, s$) yields BB(s) as described in subsection 2.2 and BE-BB($w^*(o), s$) is plain BE (where o is a min-degree ordering [16]) as described in 2.3 . We ran experiments on four problem classes:

1. $\langle 30, 5, 5, 100, 10, .995 \cdot 5^5\rangle$,
2. $\langle 35, 5, 5, 100, 12, .995 \cdot 5^5\rangle$,
3. $\langle 20, 5, 5, 100, 10, .995 \cdot 5^5\rangle$,
4. $\langle 40, 5, 2, 100, 80, 14\rangle$.

$\langle 30, 5, 5, 100, 10, .995 \cdot 5^5 \rangle$, $w^*(o) = 5$, $\overline{dg} = 6$										
			$s = 1$			$s = 2$			$s = 3$	
k	$z(o,k)$	solved	nodes	CPU	solved	nodes	CPU	solved	nodes	CPU
-1	30	11	530.9	89.7	21	323.2	49.0	21	228.2	54.7
0	21	19	175.8	57.4	25	7.1	6.1	25	3.0	17.7
1	14	19	162.6	54.6	25	2.6	2.5	25	.7	8.0
2	8	24	11.6	11.2	25	1.9	1.6	25	.4	4.9
3	4	25	.3	2.5	25	.2	.9	25	.1	1.3
4	1	25	.0	1.9	25	.0	**.5**	25	.0	.6
5	0	25	.0	2.7	25	.0	2.6	25	.0	1.0
6	0	25	.0	3.3	25	.0	3.2	25	.0	2.5

$\langle 35, 5, 5, 100, 12, .995 \cdot 5^5 \rangle$, $w^*(o) = 7$, $\overline{dg} = 6$										
k	$z(o,k)$		$s = 1$			$s = 2$			$s = 3$	
-1	35	3	646.5	113.2	4	684.6	107.5	4	493.3	108.1
0	24	9	259.5	96.1	24	24.6	27.5	19	9.5	68.4
1	16	9	253.5	95.6	25	8.3	11.2	23	3.2	39.2
2	10	25	12.6	12.7	25	4.1	4.3	25	1.5	22.1
3	5	25	1.6	5.7	25	.9	3.7	25	.7	5.9
4	2	25	.1	5.5	25	.1	**2.1**	25	.1	2.3
5	1	25	.0	8.3	25	.0	6.2	25	.0	2.7
6	1	25	.0	10.1	25	.0	9.6	25	.0	7.7

$\langle 20, 5, 5, 100, 10, .995 \cdot 5^5 \rangle$, $w^*(o) = 8$, $\overline{dg} = 7$										
k	$z(o,k)$		$s = 1$			$s = 2$			$s = 3$	
-1	20	25	109.6	30.4	21	40.8	45.3	17	28.0	72.2
0	14	25	69.4	21.6	22	23.2	38.8	17	14.8	66.9
1	10	25	51.0	16.9	24	15.6	31.1	20	10.1	59.3
2	7	25	17.5	10.8	25	8.3	15.9	21	6.5	46.5
3	5	25	4.9	11.9	25	2.7	**8.8**	25	2.5	18.7
4	4	25	1.7	43.2	25	1.2	11.5	25	1.0	15.3
5	3	25	.4	66.0	25	.5	46.3	25	.4	24.6
6	2	25	.1	105.8	25	.1	89.8	25	.1	77.2

$\langle 40, 5, 2, 100, 80, 14 \rangle$, $w^*(o) = 9$, $\overline{dg} = 4$										
k	$z(o,k)$		$s = 1$			$s = 2$			$s = 3$	
-1	40	9	162.9	84.9	9	162.1	84.9	9	162.1	84.9
0	22	14	44.2	63.2	14	44.2	63.2	14	44.2	63.2
1	9	24	13.9	26.5	24	13.9	26.5	24	13.9	26.5
2	7	25	3.2	6.8	25	3.2	6.8	25	3.2	6.8
3	6	25	1.7	**6.0**	25	1.3	**6.0**	25	1.3	**6.0**
4	5	25	1.0	14.8	25	.6	8.7	25	.5	9.0
5	4	23	.5	45.4	24	.4	29.6	25	.3	20.5
6	3	24	.1	149.4	24	.1	131.2	24	.1	87.9

Figure 11. Experimental results on four classes of WCSP and different k and s. For each combinations of k and s we report three measures. From left to right: number of solved instances (out of 25), mean number of visited nodes (in thousands) and mean CPU time (in seconds). We use bold face to indicate the best CPU times for each experiment.

For each class, we generated 25 instances. The mean induced width $w^*(o)$ of the samples along a min-degree ordering o was 5, 7, 8 and 9, respectively. The mean average degree $\overline{dg}$ of the constraint graphs was 6, 6, 7 and 4, respectively. Each instance was solved with all combinations of parameters ranging in the intervals $-1 \leq k \leq 6$ and $1 \leq s \leq 3$. Thus, each problem was solved 24 times. Some combinations of parameters were clearly inefficient, and could not solve many instances within a reasonable time. Therefore, the algorithm periodically checked the time spent so far and stopped searching when it surpassed 120 seconds. Figure 11 reports the results of this experiment (mean values). It consists of four tables, one per problem class. The first column of each table indicates the value of k. The second column indicates the mean value of $z(o,k)$ for a static min-degree variable ordering o computed as follows: Let x_i and x_j be the variables with the lowest and highest degree in the constraint graph, respectively. If the degree of x_i is less than or equal to k, x_i is selected to be the last in the ordering, it is removed from the graph, and all its neighbors are connected. Else, x_j is selected to be last in the ordering and removed from the graph. The process is recursively repeated until all nodes have been selected. This ordering o, is a static version of the dynamic variable ordering heuristic used by our algorithms (static because it disregards domain size). Thus, $d^{z(o,k)}$ can be taken as an estimated bound of the search space size traversed by BE-BB(z,s). As it can be observed in the tables, $z(o,k)$ rapidly decreases as k is increased, which indicates the impressive decrement of the search space that variable elimination causes when combined with search.

The following columns report the results obtained for the three different values of s. For each value, we provide three measures. From left to right: the number of instances solved by the algorithm within the time limit (out of 25), the mean number of visited nodes in thousands and the mean CPU time in seconds.

Regarding the number of visited nodes (middle column for each s value), our results confirm an expected behaviour. Namely, the algorithm visits fewer nodes as we increase either k (the search space size decreases) or s (more pruning is performed). Obviously, this decrement in the number of nodes requires a more expensive variable elimination and a more expensive lower bound computation. In the following, we discuss under which conditions the overhead pays off.

In all the problem classes and for all values of s, we observe a similar performance pattern as we vary k. With low values of k the algorithm performs badly because a large number of nodes have to be visited. Often times, many instances cannot be solved within the time limit. In addition, solved instances require large amounts of CPU

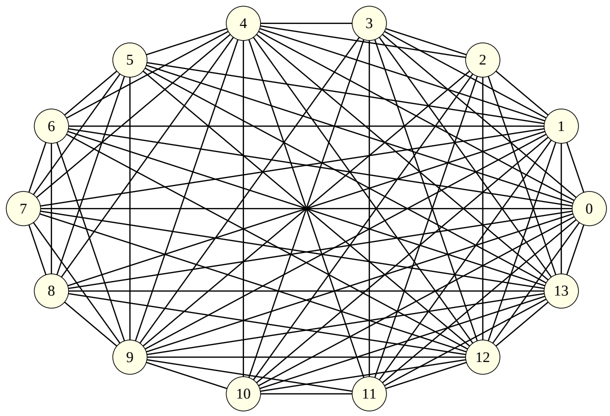

Figure 12. Subinstance 1 of CELAR6.

time on the average. As we increase k, the performance improves. Increasing k from -1 to 0 produces significant gains, which means that BB generates subproblems with variables disconnected from the rest. The variable selection heuristic delays their selection to the last moment (their heuristic selection value is infinity) and the algorithm is unable to extract their contribution to the lower bound. When k is in the range 2..4, all instances can be solved. Minimum CPU times are obtained with $k = 2, 3, 4$. Gains with respect to $k = -1$ typically range from 1 to 2 orders of magnitude in terms of CPU time (observe that when not all instances are solved, the CPU time reported is an underestimation of the real time). Higher values of k ($k > 4$) allow to solve the problems with very little search (often times visiting less than one hundred nodes) but with expensive variable elimination, which causes higher CPU times.

Regarding parameter s, we observe that it is not clear what is its best value, since it depends on the value of k (additional experiments not reported in the paper showed that $s = 0$ and $s > 3$ produce in general very bad performances). Regarding parameters k and s simultaneously, the best results are obtained with $s = 2$ and $k = 3, 4$.

5.2. Frequency Assignment Problems

In this subsection we analyze the behavior of our approach on the *radio link frequency assignment problem* (RLFAP) domain. This is a communication problem where the goal is to assign frequencies to a set of radio links in such a way that all the links may operate together without noticeable interference.

Some RLFAP instances can be naturally cast as WCSP. In particular, we experiment on subinstance 1 of the CELAR6 problem [6]. It

		$s=0$		$s=1$		$s=2$		$s=3$	
k	$z(o,k)$	nodes	CPU	nodes	CPU	nodes	CPU	nodes	CPU
-1	14	*	*	*	*	*	*	*	*
0	12	*	*	*	*	*	*	*	*
1	10	*	*	*	*	*	*	*	*
2	8	8554	30	7430	34	7430	34	7430	34
3	6	448	33	348	**27**	348	**27**	348	**27**
4	5	46	49	49	40	49	40	49	40
5	4	6	86	5	89	5	89	5	89
6	3	*	*	*	*	*	*	*	*

Figure 13. Experimental results on a frequency assignment problem and different k and s. For each combinations of k and s we report the mean number of visited nodes (in thousands) and the mean CPU time (in thousands of seconds).

has 14 variables, 74 binary constraints and all domains have 44 values. Figure 12 shows its constraint graph. This problem can be solved in about one hour with branch and bound with a sofisticated lower bound [20]. However, it is too difficult for our basic non-binary solver BE-BB(k,s) which uses a very simple lower bound. For that reason, in this experiment we reduce the problem size by reducing each domain to its ten first values. The optimal cost of the resulting problem is 24749. We considered the task of proving optimality.

Figure 13 shows the cost of solving this problem with BE-BB(k,s) as described in the previous subsection under different combinations of parameters. For each execution, we report the number of visited nodes (in thousands) and the CPU time (in thousands of seconds). An asterisc indicates that the algorithm did not finish in 24 hours. For each value of k, we also report in the second column the $z(o,k)$ value along the min-degree variable ordering described in the previous section. As can be observed, a plain search strategy ($k=-1$) cannot solve the problem. More than that, a very limited amount of variable elimination ($k=0,1$) is not enough to boost search in this problem. On the other hand, the allowance of expensive variable elimination ($k \geq 5$) causes a prohibitive overhead. Best results are obtained with $2 \leq k \leq 4$, which is in accordance with the results obtained for random WCSP.

Regarding parameter s, we observe that increasing it above 1 does not have any effect. A careful examination of the behavior of BE-BB(k,s) on this problem provides the explanation of this fact. We observed that the algorithm never interleaves search and variable elimination in the same search path: it branches until a point in which all

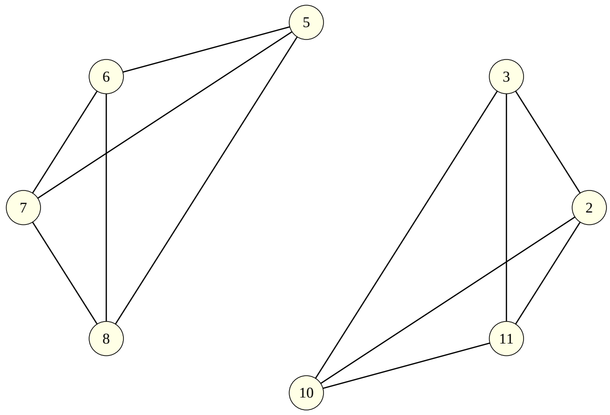

Figure 14. Subinstance 1 of CELAR6.

remaining variable can be eliminated. Consequently, since the problem is originally binary, all search is carried out with binary constraints, which makes $s > 1$ useless. We illustrate this fact for the $k = 3$ case. Observe that variables $0, 1, 4, 9, 12$ and 13 are totally connected in the original constraint graph (Figure 12). They are typically assigned first (their high degree favors their selection for branching). After this assignments, the current subproblem has the constraint graph depicted in Figure 14. All variables have degree 3, which allows the elimination of the remaining variables. We have observed a similar behavior for $k = 2, 4, 5$.

6. Related Work

The idea of using a restricted form of variable elimination to overcome its high space and time complexity is not new. Some forms of local consistency in constraint satisfaction are achieved with limited forms of variable elimination. The clearest example is relational consistency [12] in the CSP context, where the size of buckets and the arity of the new constraints is bounded (rendering the algorithm incomplete). Processed variables are not eliminated in order to preserve the soundness of subsequent search. Enforcing relational consistency makes explicit constraints that were implicit in the original problem making it presumably simpler.

Similar ideas have been explored in constraint optimization problems. Mini-bucket elimination [11] is another BE-based algorithm which also bounds the size of the buckets and the arity of the new constraints (which also renders the algorithm incomplete). It can be used as an approximation algorithm providing an upper and a lower bound on the best cost attainable. It was shown in [16] that lower bounds obtained by mini-buckets can be used during branch and bound, increasing its pruning capabilities [16].

The idea of combining variable elimination and search has also been explored in the past. The cycle-cutset method [7] was proposed for constraint satisfaction. It consists of assigning variables until a tree-structured subproblem is obtained. The subproblem is trivially solved with a dynamic programming procedure. This is essentially the same as a BE-BB(1) specialization to constraint satisfaction. Much closer to BE-BB(k) is the work of Rish and Dechter [24] in the *satisfiability* domain. They explored different hybrids of Directional Resolution, a variable elimination schema, and the Davis-Putnam search procedure. One of them, DCDR(k), is a specialization of BE-BB(k) to satisfiability. The contributions of this paper beyond this earlier work are: a) in generalizing the idea to constraint satisfaction and optimization, b) in providing a new worst-case time bound and c) in the empirical demonstration that this approach can speed-up state-of-the-art algorithms in a number of domains.

7. Conclusions

Variable elimination is the basic step of *Bucket Elimination*. It transforms the problem into an equivalent one, having one less variable. Unfortunately, there are many classes of problems for which it is infeasible, due to its exponential space and time complexity. However, by restricting variable elimination so that only low arity constraints are processed and recorded, it can be effectively combined with search to reduce the search tree size.

In this paper, we have extended a previous work of [24] in the satisfiability domain. We have introduced BE-BB(k), a new general algorithm for constraint problems that combines variable elimination and search. The tradeoff between the two solving strategies is controlled by parameter k.

We have introduced a worst-case bound for the algorithm's time complexity that depends on k and the constraint graph topology. The bound can be used to find the best balance between search and variable elimination for particular problem instances. So far, our analysis is restricted to static variable orderings. However it may effectively bound also dynamic orderings, since those tend to always be superior. Further analysis on dynamic ordering remains in our future research agenda.

We have provided empirical evaluation on three different domains. The results show that augmenting search with variable elimination is a very effective approach for both decision and optimization constraint problems. In fact, they demonstrate a general method for boosting the behavior of any search procedure. The gains on sparse random problems are impressive (sometimes up to two orders of magnitude). These results are substantially more encouraging than those reported in [24], where the hybrid approach was sometimes counter-productive.

Finally, we want to note that the idea behind our algorithm can be applied to a variety of automatic reasoning tasks (*i.e.*, probabilistic reasoning, fuzzy-logic reasoning,...), because dynamic programming and search are widely used in a variety of domains [8]. An ultimate goal of our research is to understand the synergy between these two schemas in general.

References

1. Arnborg, S.: 1985, 'Efficient algorithms for combinatorial problems on graphs with bounded decomposability - A survey'. *BIT* **25**, 2–23.
2. Bertele, U. and F. Brioschi: 1972, *Nonserial Dynamic Programming*. Academic Press.
3. Bessiére, C.: 1994, 'Arc-consistency and Arc-Consistency Again'. *Artificial Intelligence* **65**(1), 179–190.
4. Bessière, C. and J.-C. Regin: 1996, 'MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems'. In: *Proc. of the 2^{nd} CP*. Alexandria, USA, pp. 61–75.
5. Bistarelli, S., U. Montanari, and F. Rossi: 1997, 'Semiring-Based Constraint Satisfaction and Optimization'. *Journal of the ACM* **44**(2), 201–236.
6. Cabon, B., S. de Givry, L. Lobjois, T. Schiex, and J. Warners: 1999, 'Radio Link Frequency Assignment'. *Constraints* **4**, 79–89.
7. Dechter, R.: 1990, 'Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition'. *Artificial Intelligence* **41**, 273–312.
8. Dechter, R.: 1999, 'Bucket elimination: A unifying framework for reasoning'. *Artificial Intelligence* **113**, 41–85.
9. Dechter, R., K. Kask, and J. Larrosa: 2001, 'A General Scheme for Multiple Lower Bound Computation in Constraint Optimization'. In: *Proc. of the 7^{th} CP*. pp. 346–360.
10. Dechter, R. and J. Pearl: 1989, 'Tree Clustering for Constraint Networks'. *Artificial Intelligence* **38**, 353–366.
11. Dechter, R. and I. Rish: 1997, 'A Scheme for Approximating Probabilistic Inference'. In: *Proceedings of the 13th UAI-97*. San Francisco, pp. 132–141.
12. Dechter, R. and P. van Beek: 1997, 'Local and global relational consistency'. *Theoretical Computer Science* **173**(1), 283–308.
13. Freuder, E. and R. Wallace: 1992, 'Partial Constraint Satisfaction'. *Artificial Intelligence* **58**, 21–70.
14. Frost, D. and R. Dechter: 1994, 'In Search of the Best Constraint Satisfaction Search'. In: *Proceedings of the 12th AAAI*. pp. 301–306.

15. Haralick, R. M. and G. L. Elliott: 1980, 'Increasing tree seach efficiency for constraint satisfaction problems'. *Artificial Intelligence* **14**, 263–313.
16. Kask, K. and R. Dechter: 2001, 'A general scheme for automatic generation of search heuristics from specification dependencies'. *Artificial Intelligence* **129**, 91–131.
17. Larrosa, J.: 2000, 'Boosting Search with Variable Elimination'. In: *Proc. of the 6^{th} CP*. Singapore, pp. 291–305.
18. Larrosa, J.: 2002, 'Node and Arc Consistency in Weighted CSP'. In: *Proceedings of the 18th AAAI*.
19. Larrosa, J. and P. Meseguer: 1998, 'Partial Lazy Forward Checking for MAX-CSP'. In: *Proc. of the 13^{th} ECAI*. Brighton, United Kingdom, pp. 229–233.
20. Larrosa, J., P. Meseguer, and T. Schiex: 1999, 'Maintaining Reversible DAC for Max-CSP'. *Artificial Intelligence* **107**(1), 149–163.
21. Lawler, E. L. and D. E. Wood: 1966, 'Branch-and-bound methods: A survey'. *Operations Research* **14(4)**, 699–719.
22. Meseguer, P., J. Larrosa, and M. Sanchez: 2001, 'Lower Bounds for Non-binary Constraint Optimization Problems'. In: *Proc. of the 7^{th} CP*. pp. 317–331.
23. Nudel, B.: 1988, 'Tree search and arc consistency in constraint satisfaction algorithms'. *Search in Artificial Intelligence* **999**, 287–342.
24. Rish, I. and R. Dechter: 2000, 'Resolution vs. SAT: two approaches to SAT'. *Journal of Automated Reasoning* **24**(1), 225–275.
25. Schiex, T.: 2000, 'Arc Consistency for Soft Constraints'. In: *Proc. of the 6^{th} CP*. Singapore, pp. 411–424.
26. Schiex, T., H. Fargier, and G. Verfaillie: 1995, 'Valued Constraint Satisfaction Problems: hard and easy problems'. In: *IJCAI-95*. Montréal, Canada, pp. 631–637.
27. Smith, B.: 1994, 'Phase transition and the mushy region in constraint satisfaction'. In: *Proceedings of the 11th ECAI*. pp. 100–104.
28. Verfaillie, G., M. Lemaître, and T. Schiex: 1996, 'Russian Doll Search'. In: *AAAI-96*. Portland, OR, pp. 181–187.