Boosting Search with Variable Elimination. *

Javier Larrosa

Dep. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya, Barcelona, Spain
larrosa@lsi.upc.es

Abstract. *Variable elimination* is the basic step of *Adaptive Consistency*[4]. 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, because the elimination of variables, reduces the search tree size.
In this paper we introduce *VarElimSearch*($\mathcal{S}$,k), a hybrid meta-algorithm that combines search and variable elimination. The parameter $\mathcal{S}$ names the particular search procedure and 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 also provide experimental evidence that the hybrid algorithm can outperform state-of-the-art algorithms in binary sparse problems. Experiments cover the tasks of finding *one solution* and the *best solution* (Max-CSP). Specially in the Max-CSP case, the advantage of our approach can be overwhelming.

1 Introduction.

Many problems arising in a variety of domains such as scheduling, design, diagnosis, temporal reasoning and default reasoning, can be naturally modeled as constraint satisfaction problems. A *constraint satisfaction problem* (CSP) consists of a finite set of *variables*, each associated with a finite *domain* of values, and a set of *constraints*. A *solution* is an assignment of a value to every variable such that all constraints are satisfied.

Typical tasks of interest are to determine if there exists a solution, to find one or all solutions and to find the best solution relative to a preference criterion. All these tasks are NP-*hard*. Therefore, general algorithms are likely to require exponential time in the worst-case.

Most algorithms for constraint satisfaction belong to one of the two following schemes: *search* and *consistency inference*. Search algorithms can be complete

* This work was carried out while the author was visiting the *University of California at Irvine* with grant from *Generalitat de Catalunya*. The author is thankful to Rina Dechter for many useful comments and suggestions on previous versions of this paper. This research is partially funded by the Spanish CICYT under the project TAP1999-1086-C03-03 and by the NSF under grant IIS-9610015.

or incomplete. In this paper we are concerned with complete ones. These algorithms transform a problem into a set of subproblems by selecting a variable and considering the assignment of each of its domain values. The subproblems are solved in sequence applying recursively the same transformation rule, often referred to as *branching* or *conditioning*. Each time a search algorithm assigns a value to a variable it is, in a way, making a *guess* about the right value for that variable. If the guess is not correct, the algorithm will eventually *backtrack* to that point and a new guess for the same variable will have to be made. Incorrect guesses at early levels of the search tree cause the algorithm to *thrash*. This is the main drawback of these kind of algorithms. On the other hand, they have the good property of having linear space complexity.

Consistency inference algorithms transform the original problem into an equivalent one (i.e. having the same set of solutions) by *inferring* constraints that are implicit in the original problem and adding them explicitly. Each time a new constraint is added, there is more knowledge available about the relations among variables and the problem becomes presumably simpler. Consistency inference algorithms include incomplete methods which only enforce some form of *local consistency* (such as arc-consistency) as well as complete methods which enforce *global consistency*. In a globally consistent problem, a solution can be computed in a *backtrack-free* manner.

Adaptive Consistency (*AdCons*) [4] is a complete consistency inference algorithm which relies on the general schema of *variable elimination*[5]. This algorithm proceeds by selecting one variable at a time and replacing it by a new constraint which summarizes the effect of the chosen variable. The main drawback of *AdCons* is that constraints (either from the original problem, or added by the algorithm) can have large arities (i.e. scopes). These constraints are exponentially hard to process and require exponential space to store. The exponential space complexity in particular limits severely the algorithm usefulness. However, a nice property of adaptive consistency is that it never needs to make a guess. Once a variable is replaced by the corresponding constraint, the process never has to be reconsidered.

In this paper we propose a general solving scheme, *VarElimSearch*, which combines search and variable elimination in an attempt to exploit the best of each. The meta-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, branches on the variable and transforms the problem into a set of smaller subproblems where the process is recursively repeated. *VarElimSearch* has two parameters, $\mathcal{S}$ and k, where $\mathcal{S}$ names a specific search algorithm and k controls the trade-off between variable elimination and search.

The idea of combining inference and search was presented earlier by Rish and Dechter [12] within the *satisfiability* domain. They combined Directional Resolution, a variable elimination scheme for SAT, with the Davis-Putnam search procedure. Different hybrids were considered. One of them, DCDR(i), has a direct correspondence to *VarElimSearch*. The contributions of this paper beyond this earlier work are: *a*) in extending this approach to general constraint satisfaction *decision* and *optimization* tasks, *b*) in providing a new worst-case time bound based on refined graph parameters and *c*) in the empirical demonstration that this can speed-up *state-of-the-art* algorithms.

Our approach is applicable to many search strategies and a variety of tasks. In this paper, we report results of *VarElimSearch* with three search strategies: *forward checking* FC [7], *really full look-ahead* RFLA [10] and *partial forward checking* PFC [8]. We provide experimental results for the tasks of finding *one solution* and *the best solution* (namely, violating the least number of constraints, known as Max-CSP) in binary problems and fixed $k = 2$. In all cases, we show empirically that the hybrid algorithms improve the performance of plain search for *sparse* problems and has no worsening effect on dense problems. Higher levels of k should be explored in the context of non-binary solvers, and are likely to yield a tradeoff between inference and search that will be tied to the problem's structure, as was shown in [12]. This, however, is outside the scope of our current investigation.

This paper is organized as follows: The next Section introduces notation and necessary background. In Section 3 we describe the hybrid meta-algorithm *VarElimSearch*. In Section 4 we discuss the algorithm complexity both in terms of time and space. In Section 5 we provide experimental results supporting the practical usefulness of our approach. Finally, Section 6 contains some conclusions and directions of further research.

2 Preliminaries

A *constraint satisfaction problem* consists of a set of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$, domains $\mathcal{D} = \{D_1, \ldots, D_n\}$ and constraints $\mathcal{C} = \{R_{S_1}, \ldots, R_{S_t}\}$. A constraint is a pair (R, S) where $S \subseteq \mathcal{X}$ is its *scope* and R is a relation defined over S. Tuples of R denote the legal combination of values. The pair (R, S) is also denoted R_S. We denote by n and d the number of variables and the size of the largest domain, respectively. A *solution* for a CSP is a complete assignment that satisfies every constraint. If the problem is over-constrained, it may be of interest to find a complete assignment that satisfies the maximum number of constraints. This problem is denoted Max-CSP [6].

A *constraint graph* associates each variable with a node and connects any two nodes whose variables appear in the same scope. The *degree* of a variable, $degree(X_i)$, is its degree in the graph. The *induced graph* of G relative to the ordering o, denoted $G^*(o)$, is obtained by processing the nodes in reverse order from last to first. For each node all its earlier neighbors are connected, while taking into account old and new edges created during the process. 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.

Join and *projection* are two operations over relations. The join of two relations R_A and R_B denoted $R_A \bowtie R_B$ is the set of tuples over $A \cup B$ satisfying the two constraints R_A and R_B. Projecting a relation R_A over a set B ($B \subset A$),

Algorithm 1: Adaptive Consistency.

`AdCons`$((\mathcal{X},\mathcal{D},\mathcal{C}), o = X_1,...,X_n)$
Input: a CSP and a variable ordering.
Output: a solution, if there is any.
for $i = n$ downto 1 **do**
    Let $\{(R_{i_1}, S_{i_1}),...,(R_{i_q}, S_{i_q})\}$ be the set of constraints in $\mathcal{C}$ which contain X_i in their scope and do not contain any higher indexed variable
    $A \leftarrow \bigcup_{j=1}^{q}(S_{i_j} - \{X_i\})$
    $R_A \leftarrow R_A \cap \Pi_A(\bowtie_{j=1}^{q} R_{i_j})$
    $\mathcal{C} \leftarrow \mathcal{C} \cup R_A$
    if $R_A = \emptyset$ **then**
        the problem does not have solution
for $i = 1$ to n **do**
    assign a value to X_i consistent with previous assignments and all constraints in $\mathcal{C}$ whose scope is totally assigned
return the assignment to $X_1,\ldots,X_n$

written as $\Pi_B(R_A)$ removes from R_A the columns associated with variables not included in B and eliminates duplicate rows from the resulting relation.

Adaptive consistency (*AdCons*) [4] is a complete algorithm for solving constraint satisfaction problems (Algorithm 1). Given a variable ordering o, variables are processed (or, eliminated) one by one from last to first. For each variable, the algorithm *infers* a new constraint that summarizes the effect of the variable on the rest of the problem. The variable is then replaced by the constraint. Let R_A be the constraint generated by the elimination of variable X_i. The scope A is the set of neighbors of X_i in the remaining set of variables. The relation R, is the join of all constraints involving X_i in the current subproblem, projected on A and possibly intersected with any other existing constraint on A. If this process produces an empty constraint the problem does not have any solution. Otherwise, after all variables have been processed, a solution is generated in a backtrack-free manner. Variables are assigned from first to last. Variable X_i is assigned a value consistent with previous assignments and with all constraints whose scope is totally assigned. Note that the elimination of a variable X_i generates a new constraint of arity $degree(X_i)$ in the constraint graph of the remaining variables.

The complexity of *AdCons* along o is time $O(n\ d^{w^*(o)+1})$ and space $O(n\ d^{w^*(o)})$. Finding the ordering o with minimum $w^*(o)$ is an NP-*complete* problem [2].

Example:
Consider a binary CSP with the constraint graph depicted in Figure 1.*a* (each edge corresponds to a binary constraint) and the lexicographical ordering of its variables. *AdCons* starts by eliminating X_9 which causes the addition of a new ternary constraint $R_{\{1,4,5\}} = \Pi_{\{1,4,5\}}(R_{\{1,9\}} \bowtie R_{\{4,9\}} \bowtie R_{\{5,9\}})$. Next, the

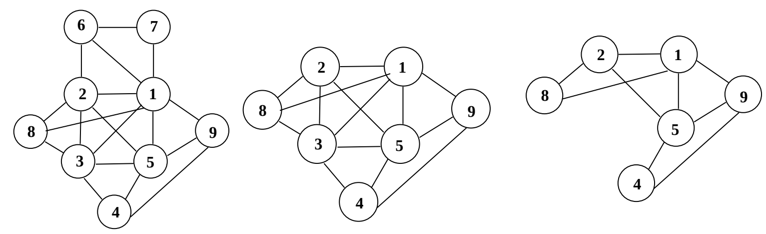

Fig. 1.

elimination of X_8 causes a new ternary constraint $R_{\{1,2,3\}} = \Pi_{\{1,2,3\}}(R_{\{1,8\}} \bowtie R_{\{2,8\}} \bowtie R_{\{3,8\}})$. The process continues until every variable is eliminated.

The induced graph $G^*(o)$, whose width gives a bound for the space and time complexity of *AdCons*, is obtained by adding two new edges $(1,4)$ and $(2,4)$. Its width $w^*(o)$ is 4. It indicates that *AdCons* computes and stores constraints of arity up to 4.

3 Combining Search and Variable Elimination.

In this section we introduce *VarElimSearch($\mathcal{S}, k$)*, a meta-algorithm that combines search and variable elimination. Let's suppose that we have a problem that we cannot solve with *AdCons* due to our limited space resources. We can still use *AdCons* as an approximation algorithm and eliminate some variables. It will transform the problem into an equivalent one having fewer variables. Subsequently, we can solve the reduced problem with a search algorithm. The recursive application of this idea is the basis of *VarElimSearch*. It is illustrated in the following example.

Example:
Consider the *binary* CSP of Figure 1.*a*. We want to choose a variable for its elimination. If our search algorithms are geared for binary CSPs, a natural criterion is to choose those variables whose elimination adds unary or binary constraints. That is, variables connected to at most two variables. In our example, variable X_7 is the only one that can be eliminated while maintaining the problem binary. Its elimination modifies a constraint $R_{\{1,6\}}$ (it becomes $R_{\{1,6\}} \cap \Pi_{\{1,6\}}(R_{\{1,7\}} \bowtie R_{\{6,7\}})$). After the elimination, X_6 has its degree decreased to two, so it can also be eliminated. The constraint graph of the current subproblem is depicted in Figure 1.*b*. 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 subproblems, one for each value of X_3. All of them have the same constraint graph, depicted in Figure 1.*c*. Now, 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. At this point, a solution can be computed in a back-track free manner. Only one branching has

been made. The elimination of variables has reduced the search space size from d^9 to d, where d is the size of the domains.

Observe that we have not made any assumption about the branching strategy. The only condition is that after the assignment of a variable, the variable stops being relevant in the corresponding subproblem. Most look-ahead algorithm satisfy this condition, because they prune all future values that are inconsistent with the assignment. Look-back search strategies may also be used but they may require some more elaborate integration. Therefore, *VarElimSearch($\mathcal{S}, k$)* has a parameter $\mathcal{S}$ which instantiates the search strategy of choice.

In the example, we limited the arity of the new constraints to two. However, in general *VarElimSearch($\mathcal{S}, k$)* bounds the arity of the new constraints to k. This parameter ranges from -1 to $n-1$ and controls the tradeoff between variable elimination and branching. Low values of k allow recording 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 search according to $\mathcal{S}$. When k is set to 0, only variables disconnected from the problem are eliminated. If k is set to its maximum value, $n-1$, every variable elimination is permitted, so the algorithm becomes *AdCons*.

Algorithm `VES` (Algorithm 2) is a recursive description of *VarElimSearch($\mathcal{S}, k$)*, where context restoration and domain updating is omitted for the sake of clarity. Each recursive call receives the search algorithm $\mathcal{S}$, the control parameter k and the current subproblem. The current subproblem is defined by the current assignment t, the set of future F and eliminated E variables, and the current set of constraints $\mathcal{C}$. In the initial call $t = \emptyset$, $F = \mathcal{X}$, $E = \emptyset$.

If `VES` is called with an empty set of future variables, the problem has a solution. It can be generated by processing variables in the opposite order in which they were selected. Eliminated variables are assigned as they would with *AdCons*, branched variables are assigned the obvious value.

If there are future variables, `VES` starts selecting one. This selection can be done using any heuristic, either static or dynamic. Then, if the variable has less than k neighbors in F, it is eliminated (`VarElim`). Else, the algorithm branches on its values (`VarBranch`).

If `VarElim` receives a variable connected to zero neighbors, the scope of the new constraint is empty, so there is no constraint to add and the variable is just discarded. If the variable is connected to one neighbor, a unary constraint is added which in practice means a domain pruning. Therefore, only when the variable is connected to more than one variable there is a real addition of a new constraint (or a tightening of a previously existing one). If the elimination causes an empty constraint, the current subproblem does not have solution and

Algorithm 2: Recursive description of *VarElimSearch($\mathcal{S}, k$)*.

`VES`($\mathcal{S}$,k, $(t, F, E, \mathcal{C})$)
if *($F = \emptyset$)* **then**
  compute solution from t, E and $\mathcal{C}$ then stop
else
  $X_i \leftarrow SelectVar(F)$
  Let $\{(R_{i_1}, S_{i_1}), ..., (R_{i_q}, S_{i_q})\}$ be the set of constraints in $\mathcal{C}$ whose scope is in F and includes X_i
  if $q \leq k$ **then**
    `VarElim`($\mathcal{S}$,$k, X_i, t, F, E, \mathcal{C}$)
  else
    `VarBranch`($\mathcal{S}$,$k, X_i, t, F, E, \mathcal{C}$)

Procedure `VarElim`($\mathcal{S}$,$k, X_i, t, F, E, \mathcal{C}$)
$A \leftarrow \bigcup_{j=1}^{q}(S_{i_j} - \{X_i\})$
$R_A \leftarrow R_A \cap \Pi_A(\bowtie_{j=1}^{q} R_{i_j})$
if $R_A \neq \emptyset$ **then**
  `VES`($\mathcal{S}$,$k, P, F - \{X_i\}, E \cup \{X_i\}, \mathcal{C} \cup R_A$)

Procedure `VarBranch`($\mathcal{S}$,$k, X_i, t, F, E, \mathcal{C}$)
foreach $a \in D_i$ **do**
  `LookAhead`($\mathcal{S}$,$X_i, a, F, \mathcal{C}$)
  if *(no empty domain)* **then**
    `VES`($\mathcal{S}$,$k, t \bowtie (i, a), F - \{X_i\}, E, \mathcal{C}$)

the algorithm backtracks. Otherwise, the eliminated variables is shifted from F to E, the new constraint is added to $\mathcal{C}$ and a recursive call to `VES` is made.

`VarBranch` iterates over the set of feasible values of the current variable. For each value, a call to `LookAhead` is made. The precise effect of `LookAhead` depends on the actual search method $\mathcal{S}$. In general, it prunes future values that are inconsistent with the current assignment. If `LookAhead` causes an empty domain, the next value is attempted. Otherwise, a recursive call is made in which the current variable is removed from F and the current assignment is extended to (i, a).

4 Complexity Analysis.

VarElimSearch($\mathcal{S}, k$) 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, *VarElimSearch($\mathcal{S}, k$)* has $O(n\ d^k)$ space complexity. Regarding time, the algorithm visits at most $O(d^n)$ nodes, because in the worst-case it performs plain search on a tree of depth n and branching factor d. The actual time complexity depends on the effort per node of the search algorithm $\mathcal{S}$ and it is the product of the search tree size and the computation per node. Clearly, this worst-case bound 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 and the time complexity, if we assumed that *VarElimSearch*($\mathcal{S}, k$) is executed with a static variable ordering. The bound is based on the following definition:

Definition:
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 less* 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 higher than k is denoted $z(o, k)$. The number of nodes in $G^*(o, k)$ with width lower than or equal to k is denoted $e(o, k)$. Clearly, $z(o, k) + e(o, k) = n$. Note also that $z(o, k) < n - k$ because the first k nodes cannot have width higher than k.

In what follows, and for the sake of analysis, we assume that *VarElimSearch*($\mathcal{S}, k$) is executed with a static variable ordering 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 lower than or equal to k are exactly the variables that *VarElimSearch*($\mathcal{S}, k$) eliminates. The edges added during the computation of $G^*(o, k)$ coincide with the constraints that the algorithm adds when it performs variable elimination. The nodes that are ignored during the computation of $G^*(o, k)$ have width higher than k and thus correspond to the variables in which *VarElimSearch*($\mathcal{S}, k$) branches.

Proposition:
VarElimSearch($\mathcal{S}, k$) with a static variable ordering o runs in time $O(d^{z(o,k)} \times (L(\mathcal{S}) + e(o, k)d^{k+1}))$ where $L(\mathcal{S})$ is the cost of the look-ahead in $\mathcal{S}$.

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

This proposition shows that *VarElimSearch*($\mathcal{S}, k$) 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 2 illustrates this fact. Given an ordered constraint graph G (Figure 2 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 side of each graph, along with the corresponding $z(o, k)$. Grey nodes are those with width lower than or equal to k and correspond to the variables that *VarElimSearch*($\mathcal{S}, k$) eliminates. Dotted edges are those added

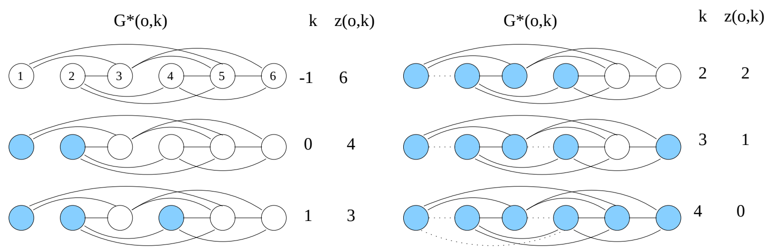

Fig. 2. An ordered constraint graph G and its k-*restricted induced graphs* $G^*(o,k)$ for $-1 \leq k \leq 4$. Note that $G = G^*(o,-1)$. Ordering o is lexicographic.

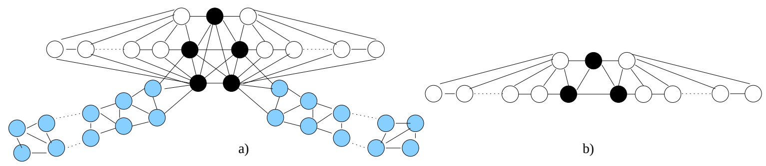

Fig. 3. A constraint graph and a subgraph after the elimination of grey variables and branching on two black variables.

during the computation of $G^*(o,k)$. White nodes are those with width higher 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	$L(\mathcal{S})$
0	4	6 d^0	d^4	$L(\mathcal{S}) + 2\ d^1$
1	3	6 d^1	d^3	$L(\mathcal{S}) + 3\ d^2$
2	2	6 d^2	d^2	$L(\mathcal{S}) + 4\ d^3$
3	1	6 d^3	d^1	$L(\mathcal{S}) + 5\ d^4$
4	0	6 d^4	d^0	6 d^5

The time complexity of the algorithm suggests a class of problems for which *VarElimSearch*($\mathcal{S},k$) 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 *AdCons* infeasible. Similarly, a search procedure may branch on the low connectivity variables causing the algorithm to thrash. *VarElimSearch*($\mathcal{S},k$) with a low k may efficiently eliminate the low-connectivity variables and search on the dense subproblems.

Example:
Consider a problem having the constraint graphs of Figure 3.a. There is a clique of size 5 (black nodes), which means that the complexity of *AdCons* is at least $O(nd^4)$ and $O(nd^5)$ space and time, respectively. Search algorithms have, in general, time complexity $O(d^n L(\mathcal{S}))$. However, *VarElimSearch*($\mathcal{S},k$), 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. The resulting subgraph of the remaining subproblems is depicted in Figure 3.*b*. At this point, the problems can be completely solved with variable elimination. Thus, the space complexity of the process was $O(n\ d^2)$, the search space size $O(d^2)$ and the time complexity $O(d^2 \times (L(\mathcal{S}) + (n-2)d^3))$. So we were able to achieve time bounds similar to *AdCons* with a lower $O(n\ d^2)$ space.

A special class of constraint problems that received substantial attention in the past two decades are binary CSPs. Many efficient algorithms that were developed were geared particularly to this class of problems. Two well known algorithms for binary problems are *forward checking* FC [7] and *really full look-ahead* RFLA [10]. Both algorithms are worst-case exponential in n, but in practice demonstrate better performance. The following proposition provides the time complexity of *VarElimSearch*(FC,2) and *VarElimSearch*(RFLA,2) (note that parameter k cannot be set to values higher than 2 because it would add non-binary constraints).

Proposition:
1. *VarElimSearch*(FC,2) has time complexity $O(n\ d^{z(o,2)+3})$.
2. *VarElimSearch*(RFLA,2) has time complexity $O(n^2\ d^{z(o,2)+3})$.

The proposition shows that the complexity of the algorithms depends on the constraint graph topology. Since $z(o,2) < n-2$, both algorithms are, at most, exponential in n.

5 Empirical Results.

In this section we provide some empirical evaluation of the potential of our approach. We restrict the experiments to binary constraint problems and we use search algorithms geared for binary problems with parameter k equal to two. The main reason for this restriction is that search algorithms are well defined, well understood and widely available for the binary case. Clearly, the extention of this empirical work to the general case and the evaluation of the effect of larger values of k is a necessary future work.

The experiments were performed on Sun WorkStations. Although different machines were used, competing algorithms were always executed on the same machine. All implementations have been coded in C, and algorithms share code and data structures whenever possible. In all the experiments *VarElimSearch* algorithms select first variables that can be eliminated. If there are none that qualify, the current domain size divided by the number of future constrained variables is computed for each future variable and the variable with the lowest ratio is selected for branching. Algorithms that do not eliminate variables, select always the variable with the lowest ratio.

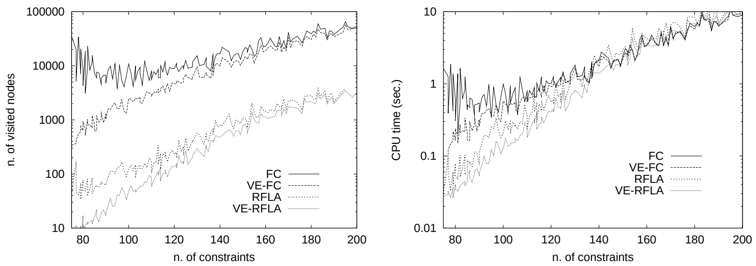

Fig. 4. Average search effort of four algorithms on the classes $\langle 50, 10, \frac{75}{1225} : \frac{200}{1225}, p_2^* \rangle$. Mean number of visited nodes and CPU time is reported. Note that plot curves come in the same top-to-bottom order than legend keys.

In our experiments we have used binary *random* CSPs as defined in [11] [1]. A binary random CSP class is characterized by $\langle n, d, p_1, p_2 \rangle$ where n is the number of variables, d the number of values per variable, p_1 the graph *connectivity* defined as the ratio of existing constraints to maximum number of constraints, and p_2 is the constraint *tightness* defined as the ratio of forbidden value pairs. The constrained variables and the forbidden value pairs are randomly selected. Instances with a disconnected constraint graph are discarded. The *average degree* in a random problem is $\overline{dg} = p_1 \times (n-1)$.

We denote $\langle n, d, p_1 : p_1', p_2 \rangle$ the consecutive classes of problems ranging from $\langle n, d, p_1, p_2 \rangle$ to $\langle n, d, p_1', p_2 \rangle$ and making the smallest possible increments (that is, each class has problems with one more constraint than the previous class). A similar notation is used to denote sequences of problem classes with respect n, d and p_2. In all the experiments, samples have 50 instances.

5.1 Finding one solution (CSP)

In our first set of experiments we consider the task of finding one solution (CSP). We compare FC and RFLA versus *VarElimSearch*(FC,2) and *VarElimSearch*(RFLA,2) (VE-FC and VE-RFLA, for short). Our implementation of RFLA 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. We also report the number of visited nodes.

In our first experiment we ran the four algorithms on $\langle 50, 10, \frac{75}{1225} : \frac{200}{1225}, p_2^* \rangle$, where p_2^* denotes the *cross-over* tightness (tightness that produces 50% satisfiable problems and 50% unsatisfiable) or the closest approximation. In these set of problems $\overline{dg}$ increases with the connectivity. It ranges from 3 to 8. Figure 4 reports the average number of visited nodes and average CPU time for each algorithm (note the *log* scale). We observe that VE-FC (resp. VE-RFLA)

[1] Our benchmarks are not affected by the result of [1] because, as we will show, problem instances cannot be trivially solved with algorithms that enforce arc-consistency. Therefore, they cannot have flawed variables.

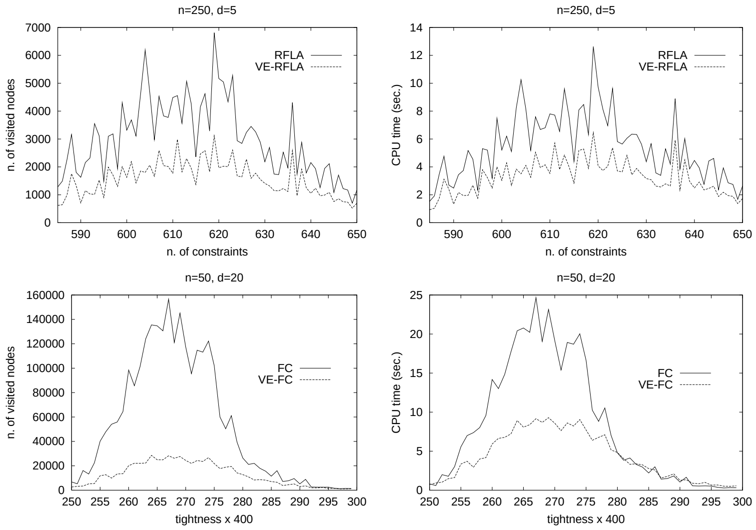

Fig. 5. Experimental results on the classes $\langle 250, 5, \frac{585}{31125} : \frac{650}{31125}, p_2^* \rangle$ and $\langle 50, 20, 125, \frac{250}{400} : \frac{300}{400} \rangle$. Mean number of visited nodes and CPU time is reported.

clearly outperforms FC (resp. RFLA) both in terms of nodes and CPU time. The maximum gain is observed in the most sparse problems where VE-FC (resp. VE-RFLA) can be 2 or 3 times faster than FC (resp. RFLA). As a matter of fact, in the most sparse problems variable elimination usually does most of the work and search only takes place in a few variables. 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 lower than or equal to 2. Therefore, the gain of VE-FC and VE-RFLA gracefully vanishes. Interestingly, detecting that no variable elimination is possible can be done while searching for the variable with the smallest domain, so in those problems where no variables are eliminated, the overhead of variable elimination is negligible.

Next, we investigate how the algorithms scale up, while keeping the average degree around 5. The four algorithms were executed in the classes $\langle 250, 5, \frac{585}{31125} : \frac{650}{31125}, p_2^* \rangle$ and $\langle 50, 20, 125, \frac{250}{400} : \frac{300}{400} \rangle$. In the first set of problems, FC performed very poorly compared to RFLA, in the second it was RFLA who performed poorly compared to FC, so we report the results with the better algorithm for each class only. Figure 5(top) reports the average results of executing RFLA and VE-RFLA in the 250 variables problems. The superiority of VE-RFLA is again apparent in this class of problems. VE-RFLA is always faster than RFLA and the gain ratio varies from 1.2 to 3. Figure 5(bottom) reports the average results of executing FC and VE-FC in the $\langle 50, 20, 125, \frac{250}{400} : \frac{300}{400} \rangle$ problems. VE-FC is also clearly faster than FC. At the complexity peak, VE-FC is 2.5 times faster than FC.

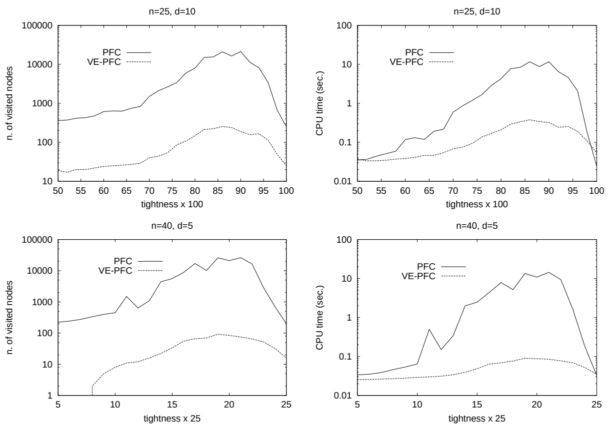

Fig. 6. Average search effort of PFC and VE-PFC on $\langle 25, 10, \frac{37}{300}, p_2 \rangle$ and $\langle 40, 5, \frac{55}{780}, p_2 \rangle$. Mean number of visited nodes and CPU time is reported.

5.2 Finding the best solution (Max-CSP)

In our second set of experiments we consider the task of finding a solution that violates the least number of constraints in over-constrained problem instances. This task is known as Max-CSP [6]. We experiment with *Partial Forward Checking* algorithms, which are the extension of forward checking to *branch & bound*. Specifically, we consider PFC-MRDAC [8], a state-of-the-art algorithm that uses arc-inconsistencies to improve the algorithm pruning capabilities. We compare PFC-MRDAC with *VarElimSearch*(PFC-MRDAC,2) (PFC and VE-PFC, for short).

The implementation of VE-PFC requires the adaptation of *VarElimSearch* to optimization tasks. This requires to modify the search schema to *branch and bound* and the variable elimination schema to Max-CSP. We interested reader can find a precise description of it in a full version of this paper ([9]).

In our first experiment on Max-CSP we select the same class of sparse random problems that was used in [8] to prove the superiority of PFC over other competing algorithms. Namely, the classes $\langle 25, 10, \frac{37}{300}, p_2 \rangle$ and $\langle 40, 5, \frac{55}{780}, p_2 \rangle$. Figure 6 reports the average results. As can be seen, the superiority of VE-PFC over PFC is impressive both in terms of nodes and CPU time (note the *log* scale). VE-PFC is sometimes 30 times faster than PFC and can visit 100 times fewer nodes in the 25 variable problems. In the 45 variable problems, the gain is even greater. VE-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, VE-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 bellow 3). So the next experiment considers denser problems. The two algorithms are executed in the three following sequences of classes: $\langle 10 : 39, 10, 1.5n, \frac{85}{100} \rangle$, $\langle 10 : 27, 10, 2n, \frac{85}{100} \rangle$ and $\langle 10 : 17, 10, 3n, \frac{85}{100} \rangle$. In these problem

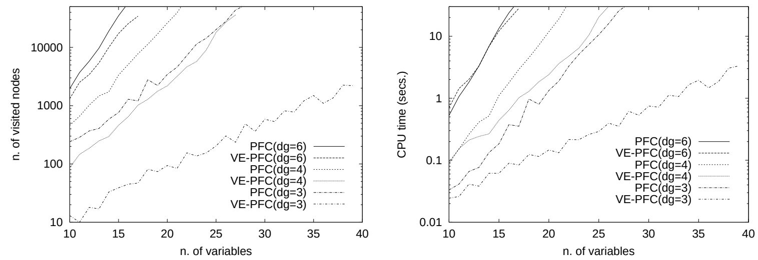

Fig. 7. Average search effort of PFC and VE-PFC on three classes $\langle 10 : 39, 10, 1.5n, \frac{85}{100}\rangle$, $\langle 10 : 27, 10, 2n, \frac{85}{100}\rangle$ and $\langle 10 : 17, 10, 3n, \frac{85}{100}\rangle$. In each class, the number of variables increases while the average degree is fixed ($\overline{dg}$ is 3, 4 and 6, respectively). 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.

sets, we increase the number of variables and constraints proportionally, so the average degree remains fixed ($\overline{dg}$ is 3, 4 and 6, respectively). Figure 7 reports the search measures obtained in this experiment. Again, the superiority of VE-PFC is crystal clear. However, as could be expected, the gain decreses as $\overline{dg}$ increases. In the problems with $\overline{dg} = 3$, VE-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 of about 9 times in terms of time and 18 times 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 VE-PFC is 30% faster in the largest instances. Regarding visited nodes, the gain of VE-PFC ranges from 40% in the smallest instances, to 250% in the largest. The plots also indicate that the advantage of VE-PFC over PFC seems to increase with the size of the problems if the average degree remains fixed.

6 Conclusions and Future Work.

Variable elimination is the basic step of *Adaptive Consistency*. 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 [12] in the satisfiability domain. We have introduced *VarElimSearch*, a hybrid meta-algorithm for constraint satisfaction that combines variable elimination and search. The tradeoff between the two solving strategies is controlled by a parameter.

We have introduced a worst-case bound for the algorithm's time complexity that depends on the control parameter and on 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 can 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 sparse binary random problems for a fixed value of $k = 2$. The results show that augmenting search with variable elimination is very effective for both decision and optimization constraint satisfaction. In fact, they demonstrate a general method for boosting the behavior of search procedures. The gains are very impressive (sometimes up to two orders of magnitude). This results are even more encouraging than those reported in [12], where the hybrid approach was sometimes counter-productive. Clearly, we need to extend our evaluation to the non-binary case, since its practical importance is more and more recognized in the constraints community.

Finally, we want to note that the idea behind our algorithm can be applied to a variety of constraint satisfaction and automatic reasoning tasks, because variable elimination and search are widely used in a variety of domains [5]. We have demonstrated its usefulnes in decision and optimization constraint satisfaction tasks. An ultimate goal of our research is to understand the synergy between these two schemes within a general framework of automated reasoning.

References

1. D. Achlioptas, L.M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy and c. Stamatious. Random constraint satisfaction: A more accurate picture.*CP'97*, 1997.
2. S.A. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - A survey. *BIT*, 25:2–23, 1985.
3. C. Bessiere. Arc-Consistency and Arc-Consistency Again. *Artificial Intelligence*, 65(1):179–190, 1994.
4. R. Dechter and J. Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.
5. R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.
6. E.C. Freuder and R.J. Wallace. Partial Constraint Satisfaction *Artificial Intelligence*, 58:21–70, 1992.
7. R.M. Haralick and G.L. Elliot. Increasing tree seach efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
8. J. Larrosa, P. Meseguer and T. Schiex. Maintaining Reversible DAC for Max-CSP. *Artificial Intelligence*, 107:149–163, 1998.
9. J. Larrosa and R. Dechter Dynamic Combination of Search and Variable Elimination in CSP and Max-CSP. available at `http://www.ics.uci.edu/ dechter`.
10. B. Nudel. Tree search and arc consistency in constraint satisfaction algorithms. L.N. Kanal and V. Kumar, editors, *Search in Artifical Intelligence*, 287–342, Springer-Verlag, 1988.
11. B.M. Smith. Phase transition and mushy region in constraint satisfaction problems. In *Proceedings of ECAI'94*. pg. 100–104. 1994.
12. I. Rish and R. Dechter. Resolution vs. SAT: two approaches to SAT To appear in *Journal of Approximate Reasoning. Special issue on SAT.*