2013

Improving No-Good Learning in Binary Constraint Satisfaction Problems

Mary D. Burke
University of Nebraska-Lincoln, mburke@cse.unl.edu

Berthe Y. Choueiry
University of Nebraska-Lincoln, choueiry@cse.unl.edu

Improving No-Good Learning in Binary Constraint Satisfaction Problems[*]

by

Mary D. Burke and Berthe Y. Choueiry
Email: {mburke|choueiry}@cse.unl.edu

UNL-CSE-2013-xxxx

Constraint Systems Laboratory
University of Nebraska-Lincoln, USA

Lincoln, Nebraska

May, 2013

Department of Computer Science & Engineering, University of Nebraska-Lincoln, Technical Report, TR-UNL-CSE-2013-0002

Abstract

Conflict-Directed Backjumping (CBJ) is an important mechanism for improving the performance of backtrack search used to solve Constraint Satisfaction Problems (CSPs). Using specialized data structures, CBJ tracks the reasons for failure and learns inconsistent combinations (i.e., no-goods) during search. However, those no-goods are forgotten as soon as search backtracks along a given path to shallower levels in the search tree, thus wasting the opportunity of exploiting such no-goods elsewhere in the search space. Storing such no-goods is prohibitive in practice because of space limitations. In this thesis, we propose a new strategy to preserve all no-goods as they are discovered and to reduce them into no-goods of smaller sizes without diminishing their pruning power. We show how our strategy improves the performance of search by exploiting the no-goods discovered by CBJ, and saves on storage space by generalizing them.

[*]A preliminary version of this technical report was submitted by Mary D. Burke as an Honors Thesis to the Honors Program of the University of Nebraska-Lincoln.

Contents

1 Introduction

Constraint Processing (CP) is a powerful and flexible framework for modeling and solving many real-world problems such as planning and resource allocation, design and product configuration, and hardware and software verification [**?**]. It is an active area of Artificial Intelligence and Theoretical Computer Science. Constraint Satisfaction Problems (CSPs) are in general NP-Complete. Backtrack search is the only complete and sound algorithm for solving a CSP instance. The performance of search is improved by several mechanisms. In this thesis, we focus on two such independent mechanisms: *forward checking* (FC) and *conflict-directed backjumping* (CBJ).

- FC propagates, after each variable instantiation, the impact of the decision to the rest of the problem. FC improves the performance of search by reducing the size of of the future subproblem.
- CBJ maintains data structures that allow search, at every dead-end, to recognize the deepest reason for failure and jump back to that instantiation past intermediate, irrelevant ones. It thus reduces the backtracking effort.

The combination of those orthogonal two mechanisms yields forward checking with conflict-directed backjumping (FC-CBJ) [**?**].

FC-CBJ traces the reasons for failure along a search path within its data structures and learns an inconsistent combination of variable-value pairs (i.e., no-good) at each backtrack. Unfortunately, those no-goods are not maintained throughout the search process but are discarded when the data structures are reinitialized to store new conflicts. As a result, potential benefits of the newly discovered no-goods are lost for the remaining of the search process. If remembered throughout the search and propagated whenever applicable, those no-goods could be used to prune unexplored subtrees of the search space and, consequently, reduce the overall effort of solving the CSP.

We propose a strategy for extracting, storing, and propagating the no-goods discovered during FC-CBJ search. Because space is a large concern of storing no-goods, we propose a technique for generalizing the set of learned no-goods into a reduced set. We show how our strategy of storing and utilizing the no-goods saves on overall search effort. Furthermore, we show how our strategy for reducing the set of no-goods saves on storage space.

This thesis is organized as follows. Section 2 provides background information of CSPs and SAT problems. Section 3 reviews how no-goods are learned and used in CP and SAT solvers. Section 4 describes the new algorithms for learning, propagating, and reduced the no-goods discovered in FC-CBJ. Section 5 proposes a new mechanism to handle backtracking in the presence of no-goods. Section 6 describes the actual algorithms and provided the corresponding pseudocode. Section 7 discusses how the discovered no-goods are summarized (i.e., reduced). Section 8 describes our method for propagating no-goods during search. Section 9 discusses our experimental results on randomly generated problems and on benchmark instances. Section 10 concludes this document. Finaly, Appendix A provides original pseudocode of the main function that we modify [**?**].

2 Background

In the following two sections, we introduce and CSPs and SAT problems and discuss the main techniques used for solving of each them.

2.1 Constraint Satisfaction Problems

Formally, a Constraint Satisfaction Problem (CSP) is defined as $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, where $\mathcal{V}$ is a set of variables, $\mathcal{D}$ is a set of domains, and $\mathcal{C}$ is a set of constraints. Each variable $V_i \in \mathcal{V}$ is associated with a domain $\mathcal{D}(V_i)$, which is a set of possible values for the variable V_i. A constraint $c_i \in \mathcal{C}$ is defined by a scope $scope(c_i)$ and a relation R_i. The scope of the constraint is a subset of the variables in $\mathcal{V}$ and R_i is a relation over the domains of the variables in $scope(c_i)$. The cardinality of $scope(c_i)$ is the arity of the constraint. In this thesis, we restrict ourselves to *binary* CSPs. Solving a CSP corresponds to assigning a domain value $d_i \in \mathcal{D}(V_i)$ to each variable $V_i \in \mathcal{V}$ such that all the constraints are satisfied. Determining whether a CSP is solvable or consistent is a *satisfiability* problem and is in general NP-complete). Counting the number of solutions of a CSP is the associated *model-counting* problem and is in general #P.

Graph coloring can easily be modeled as a CSP. The example given in Figure 1 is taken from **?** [**?**]. In this example,

- $\mathcal{V} = \{X_1, X_2, \ldots, X_6\}$.
- $\mathcal{D}(X_1) = \mathcal{D}(X_2) = \cdots = \mathcal{D}(X_6) = \{red, blue, green\}$.
- The constraints in this example require the two adjacent areas cannot be given the same color. They are shown as are shown as dotted edges linking the vertices corresponding to the variables.

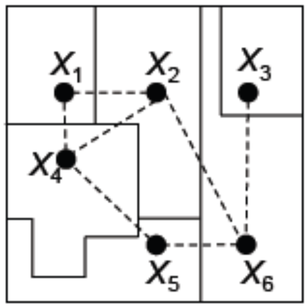


Figure 1: Graph coloring example.

- The representation of the variables and the constraints as vertices and edges of a graph define the constraint graph of the CSP.

A solution to the problem in Figure 1 is $\{X_1 = red, X_2 = blue, X_3 = blue, X_4 = green, X_5 = red, X_6 = green\}$.

A *variable-value pair* (V_i, v_i) is a tuple corresponding to the assignment to a variable V_i of a value v_i from its domain $v_i \in \mathcal{D}(V_i)$. A *partial solution* is set of variable-value pairs. The partial solution is consistent if it satisfies all the constraints defined on its variables. A *no-good* is a set of variable-value pairs that does not participate in any solution of the problem [**?**]. A no-good is said to be *minimal* if and only if every strict subset of it appears in some solution to the CSP.

Below we discuss methods for solving CSPs.

2.1.1 Backtrack Search

Backtrack search (BT) is an exhaustive, systematic, enumeration of the combinations of variables-values pairs of a CSP. It proceeds in a depth-first manner in order to maintain a linear space-requirement. Variables are considered in sequence for instantiation (i.e., value assignment). A path is extended only when the assignment of the last variable is consistent with all past assignments (i.e., consistent partial solution). A solution is found (and search is ended) when all variables have been instantiated and all constraints are satisfied. If a current partial solution reaches a dead-end (i.e., no consistent value is found in the domain of the considered variable), search backtracks to the latest assignment, undoes it, and attempts to find another labeling to the unlabeled variable. The process proceeds systematically until a solution is found or the problem is found to be unsolvable.

Figure 2 depicts the BT search in a simple example where the variable V_4 has no value in its domain consistent with the partial solution $\{V_1 \leftarrow 1, V_2 \leftarrow 1, V_3 \leftarrow 1\}$. Search backtracks to the most recently instantiated variable V_3, undoes the

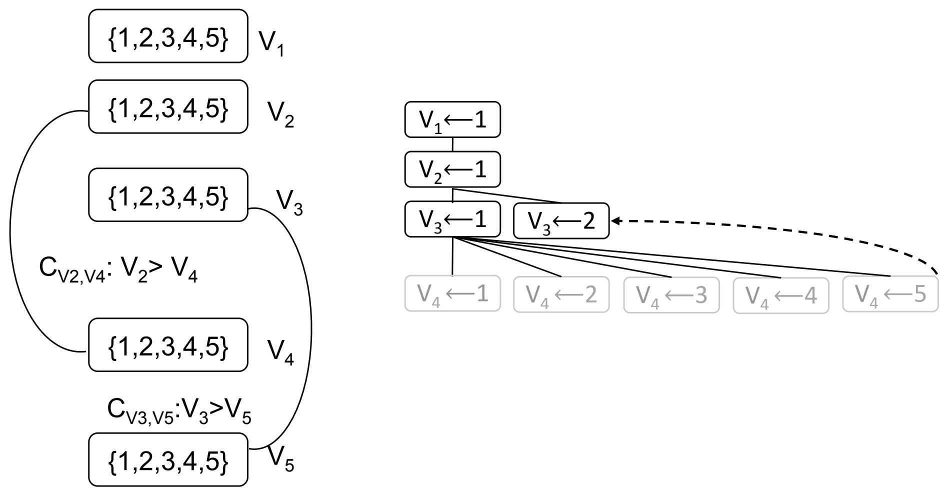


Figure 2: Simple BT example.

instantiation, then proceeds with another value for V_3.

2.1.2 Conflict-Directed Backjumping (CBJ)

Conflict-Directed Backjumping (CBJ) is an improvement to chronological backtracking. When search reaches a dead-end, instead of undoing the deepest assignment, CBJ jumps past those variables that could not have possibly caused the conflict that caused the dead-end. It recognizes the deepest assignment that may have yielded the conflict, jumps back to it, and undoes it after undoing all intermediary assignments. Thus, it avoids thrashing and is guaranteed not to lose any solution. CBJ is able to keep jumping across conflicts while guaranteeing completeness [**?**].

To achieve this mechanism, CBJ maintains a data structure where the $confSet(V_i)$ of each variable V_i stored for all variables in the current path of the search tree. The conflict set of a variable V_i is the set of all the variables in the current path whose current assignment is inconsistent with a value in the domain values of V_i that was discarded from instantiation. If, during search, there are no consistent domain values for the variable V_i,

1. $confSet(V_h)$ is updated to $(confSet(V_i) \setminus \{V_h\}) \cup confSet(V_h)$.

2. For all $V_x, x \in (h, i], confSet(V_x) \leftarrow \emptyset$.
3. The search jumps back to the deepest variable V_h in $confSet(V_i)$ undoing the current instantiation of V_h.

If immediately after backjumping, V_h is found to have no feasible domain values, CBJ continues backjumping again from V_h to the deepest variable in conflict set of V_h until a variable with feasible domain values is found [**?**]. Figure 3 illustrates the backtrack portion of CBJ. Notice after the first backjump to V_4, we backjump

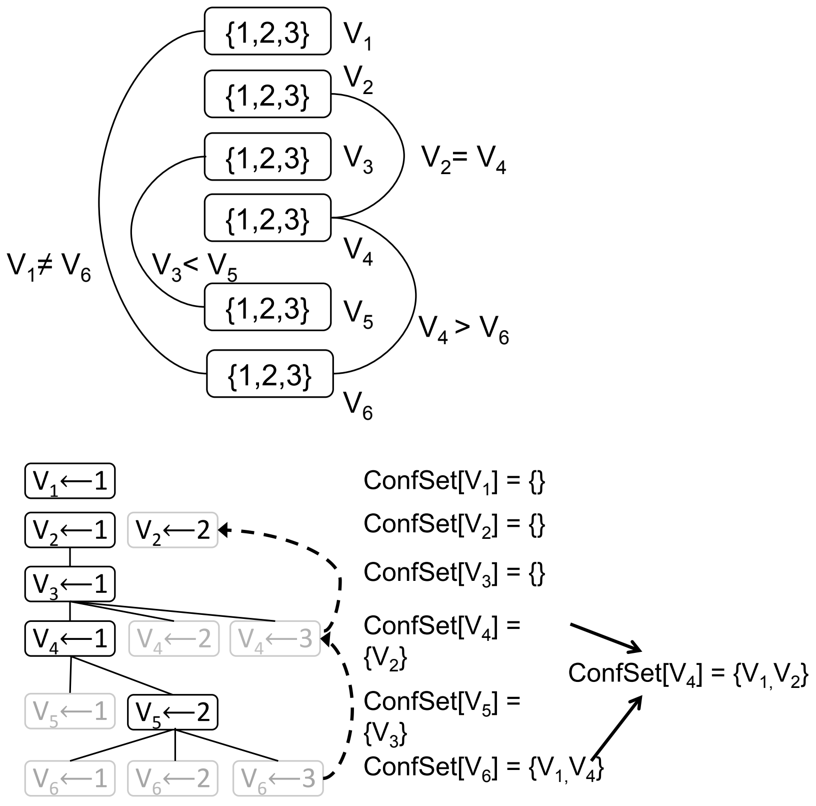


Figure 3: Simple CBJ example.

again to V_2 instead of backtracking chronologically to V_3.

2.1.3 Forward Checking (FC)

Forward Checking (FC) improves upon backtrack search by incorporating constraint propagation as a lookahead strategy [**?**]. At the instantiation of the current variable V_i the algorithm looks ahead to the uninstantiated variables (future variables) and clears the domain values of those future variables that inconsistent with the variable instantiation of V_i. This algorithm guarantees that all the values in the domains of the future variables are consistent with the current partial solution because all inconsistent values have already been removed. If a future variable's domain is "wiped out" by the instantiation of the current variable V_i, FC backtracks chronologically.

2.1.4 Hybrid FC-CBJ

? [**?**] proposed FC-CBJ as a hybrid backtrack search algorithm that combines FC and CBJ for which he provide the detailed pseudocode. This algorithm is the one that we will improve upon because it was empirically shown to be the most effective in practice [**?**]. The pseudocode has one important function FC-CBJ-UNLABEL, which we report in Algorithm 11 in Appendix A.

2.2 Satisfiability

A Satisfiability (SAT) sentence is defined by a set of Boolean variables and a set of constraints over these variables. In conjunctive normal form (CNF), a SAT sentence is specified as a conjunction of clauses, where each clause is a disjunction of literals and each literal is a term (i.e., Boolean variable) or its negation. Solving a SAT instance requires finding an assignment of truth values for the terms such that the SAT sentence holds. Sound and complete SAT solvers are based on backtrack search, more specifically the DPLL procedure, which relies on unit propagation to speed up the search [**?**; **?**]. Recently, DPLL has been extended to incorporate 'two-watched-literals' unit propagation, conflict-clause learning, and conflict resolution [**?**; **?**; **?**]. The resulting procedure is commonly called conflict-driven clause learning (CDCL).

SAT solvers frequently implement *watched literals* to assist in propagation [**?**; **?**; **?**]. Two literals per learned clause are selected to ensure that the learned clause is not violated. These literals are known as watched literals, w_1, w_2, and, while both watched literals are uninstantiated no propagation can be made with regard to the no-good associated to the watched literals. When one watched literal, say w_1, becomes false, we look for another uninstantiated literal in the no-good as a replacement. If none exists, w_2 is instantiated to verify the clause and propagated accordingly.

3 Previous Approaches to No-Good Learning

In this section, we report a wide review of the state of the art on no-good learning in CP and SAT.

The concept of no-good learning has originated from the conflict-sets discovered by CBJ. In the history of CP, one paper questioned the utility of CBJ in the presence of particularly effective heuristics for dynamic variable ordering [**?**]. That reason, along with the implementation 'difficulty' of CBJ, has resulted in CBJ being snubbed by the CP research community in favor of FC, a more aggressive lookahead strategy called than MAC [**?**], and dynamic variable ordering. As a result, the study of the no-goods generated by CBJ was overlooked by the CP community. However, the opportunity was embraced by the SAT community in an influential paper that brought a simple CP mechanism to empower SAT solvers [**?**], fueling up the power SAT solvers.

3.1 No-Good Learning in CSP

A number of learning techniques have been proposed to improve the performance of search. Those mainly include value-based learning, graph-based shallow learning, jumpback learning, and deep learning [**?**; **?**]. In those learning schemes, the learned conflict added is called a "conflict-set," which is a no-good. Generally speaking, a no-good discovered during search is a subset of the current path (i.e., partial assignment) that has yielded the dead-end.

- *Value-based learning* removes all irrelevant variable-value pairs from a conflict set before adding the conflict set to the problem. An irrelevant variable is a variable that is consistent with *all* of the domain values of the dead-end variable.

- *Graph-based shallow learning* derives conflict sets by analyzing the constraint graph after a dead-end is encountered.

- *Jumpback learning* uses the conflict-set maintained by CBJ. Because the new conflict-set is provided by CBJ itself, no additional analysis is required and the complexity of computing the conflict-set is constant.

- *Deep learning* records all and only minimal conflict sets during search. Implementation of deep learning can be done by means of enumeration: first considering all conflict-sets containing a single variable-value pair, then all those two, etc. This approach is costly and may require exponential time and space complexity at each dead-end [**?**].

Other techniques proposed by **?** [**?**; **?**] and **?** [**?**] incorporate no-good learning into FC-CBJ and GAC by means of both *standard* and *generalized* no-good learning. A *standard no-good* consists only of variable assignments that do not participate in any solution while a *generalized no-good* can contain both variable assignments and variable prunings that do not participate in any solution. For example, for a CSP with variables $V = \{V_1, V_2, V_3\}$ and domains $D(V_1) = D(V_2) = D(V_3) = \{1, 2, 3\}$:

- The standard no-good $ng_{st} = \{V_1 \leftarrow 1, V_2 \leftarrow 1\}$ states that the that assignments of $V_1 \leftarrow 1$ and $V_2 \leftarrow 1$ together are inconsistent. That is, if $V_1 \leftarrow 1$ then 1 must be pruned from $D(V_2)$ (and vice versa) because the combination $\{V_1 \leftarrow 1, V_2 \leftarrow 1\}$ is not part of any solution.

- The generalized no-good $ng_{gen} = \{V_1 \leftarrow 1, V_2 \not\leftarrow 1\}$ states that both the assignment $V_1 \leftarrow 1$ and the pruning of the domain value 1 from $D(V_2)$ are inconsistent. Thus, if $V_1 \leftarrow 1$ then the value 1 cannot be pruned from the domain of V_2 or, if the value 1 has pruned from $D(V_2)$, then the value 1 must be pruned from $D(V_1)$.

? [**?**] combine no-good learning with randomized restarts. Where no-goods are only minimized and recorded immediately before the restart.

Another mechanism for generating no-goods is *lazy clause-generation* [**?**]. Unlike other learning algorithms, this mechanism does not explicitly store the reason for pruning a value from a variable's domain as CBJ does. Rather, whenever a dead-end is encountered because of a domain wipe-out, it invokes a function to analyze the reason for failure and generate a no-good.

In addition to no-good learning, *no-good forgetting* was proposed to cope with the large overhead associated with storing and propagating no-goods. **?** [**?**] and **?** [**?**] implement first, second, third, and fourth-order learning for standard no-goods. In i-order learning, only no-goods consisting of i or fewer variables are recorded. **?** [**?**] and **?** [**?**] implicitly incorporate clause forgetting by means of external SAT solver usage. **?** [**?**] applies no-good forgetting to the generalized no-goods and further enhances lazy clause generation by incorporating no-good forgetting.

3.2 Clause Learning in SAT

Clause learning has become a fundamental technique at the heart of the success of modern SAT solvers. **?** [**?**] implemented CBJ with third-order learning, originally introduced for CSPs [**?**; **?**].

In SAT, a variable is assigned a value either by a decision of the algorithm (i.e., *decision variable*), or by means of propagation from either a clause in the original problem or a clause learned during search. Negative variable assignments correspond to the variables being assigned the value 'false.' Similarly, a positive variable assignment corresponds to the variable being assigned the value 'true.' A conflict corresponds to assigning both 'true' and 'false' values to the same variable V_i. An *implication graph* is a directed acyclic graph whose vertices represent variable assignments. The edges in the implication graph represent the reasons that a variable was assigned a value. For a given vertex V_i, the vertices at the originating end of the edges pointing to V_i are called the *antecedent vertices* of V_i. Note that decision variables have no edge pointing to them and, thus, have no antecedent vertices. When a conflict occurs, we collect every decision variable at the root of some path in the implication graph leading to the conflicting assignment. We form a *conflict clause* as the disjunction of the negations of those assignments. The learned conflict clause is added to the theory. We backtrack to the deepest decision variable in the conflict clause to avoid the conflict. This operation corresponds to learning no-goods and implementing CBJ. Such learned clauses are not desirable because the decisions on which they are built are unlikely to be encountered again. Preferable choices can be made by exploiting the implication graph.

Indeed, a conflict clause can be formed by performing a cut through the vertices of an implication graph partitioning it into two parts: the conflict portion and the reason portion. The decision variables are in the reason portion. The conflict portion obviously contains the conflicting clause variable $\{V_i \leftarrow true, V_i \leftarrow false\}$ and the two vertices corresponding the conflicting assignments of V_i. For any cut, the vertices in the reason side that have an edge leading to vertices in the conflict side are used to form a conflict clause. The question becomes which clause is more desirable.

Unique Implication Point (UIP) is another clause-learning techniques currently widely used in SAT solvers [**?**]. A unique implication point (of the decision variable at the current decision level) is an internal vertex of the implication graph through which go all the paths from the vertex corresponding to the decision variable of the current assignment to the conflicting variables. The first UIP is the one closest to the conflict. The method of choice is build the conflict clause that uses the first UIP [**?**].

4 No-good Learning and Propagation in FC-CBJ

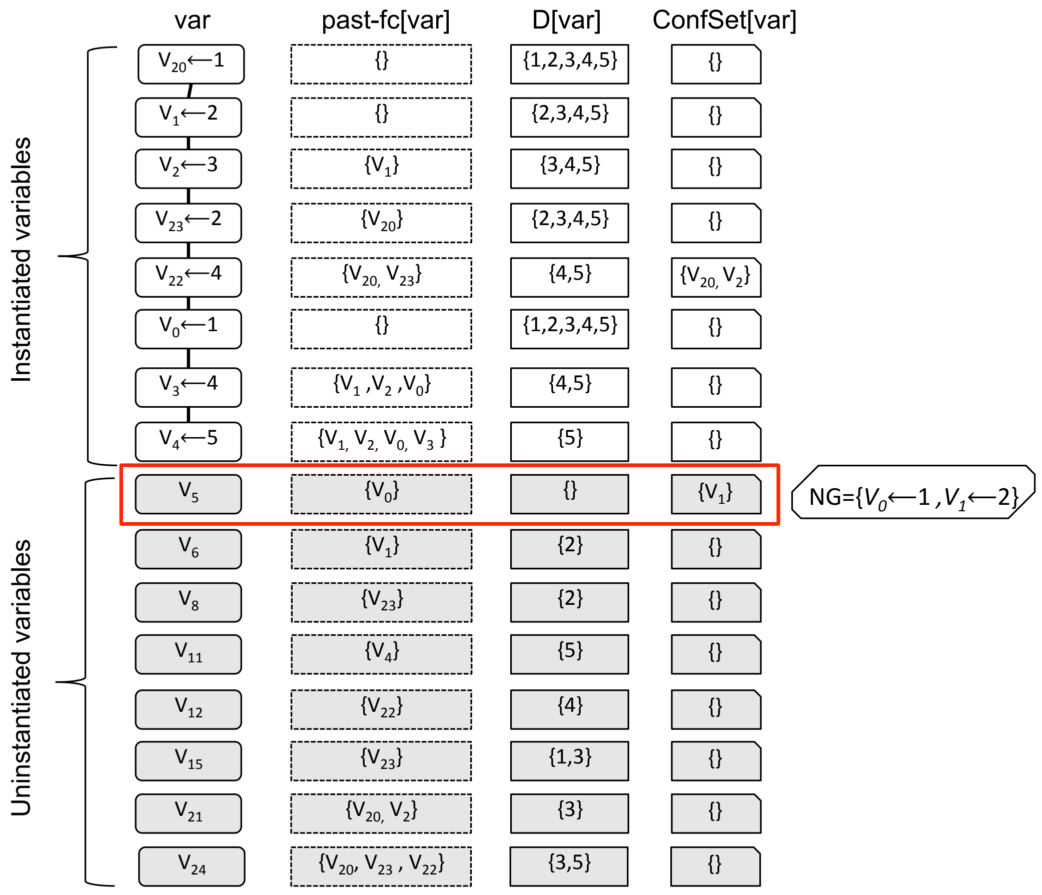
In this section, we discuss our proposition to implement no-good learning in FC-CBJ. As FC-CBJ builds a solution for a CSP, it maintains several lists to track the filtering in the domains of the uninstantiated variables. For each variable, V_i,

- $pastFC(V_i)$ maintains the set of past variables that pruned one or more values from the domain of V_i, $D(V_i)$.
- $confSet(V_i)$ is the list of variables that conflicted with V_i's instantiation.

Together, these two sets of variables encapsulate the reason why a specific value $v \in D(V_i)$, where $D(V_i)$ is the domain of V_i, was pruned from $D(V_i)$. If the domain of variable V_i is wiped-out and FC-CBJ must backtrack, the reason for the domain wipe-out is captured by $confSet(V_i) \cup pastFC(V_i)$. Thus, a new no-good, ng, can be defined by the variable-value pairs of the variables listed in $confSet(V_i) \cup pastFC(V_i)$.

For example, consider the constraint model of the Zebra puzzle [**?**]. The variables $V_0, V_1, \ldots, V_{24}$ represent five nationalities, five house colors, five pets, five cigarette brands, and five beverages. The domain values for each variable $D(V_0) = D(V_1) = \cdots = D(V_{24}) = \{1, 2, 3, 4, 5\}$ represent a house identifier. The goal is to assign a unique nationality, color, pet, cigar, beverage to (the person who lives in) each of the houses. Consider the following sequence of assignments depicted in Figure 4:

- The assignment $V_{20} \leftarrow 1$ prunes the value 1 from $D(V_{21})$, $D(V_{22})$, $D(V_{23})$, $D(V_{24})$. Thus, $pastFC(V_{21}) = pastFC(V_{22}) = \cdots = pastFC(V_{24}) = \{V_{20}\}$.
- Attempting the assignment $V_1 \leftarrow 1$ causes a domain wipe-out for the unassigned variable V_0. Thus $confSet(V_1) = confSet(V_1) \cup pastFC(V_0) = \emptyset$. Assigning $V_1 \leftarrow 2$ removes the value 2 from $D(V_2)$, $D(V_3)$, $D(V_4)$ and the values in $\{1, 3, 4, 5\}$ from $D(V_6)$. Thus, $pastFC(V_2) = pastFC(V_3) = pastFC(V_4) = pastFC(V_6) = \{V_1\}$.



var	past-fc[var]	D[var]	ConfSet[var]
V_{20}←1	{}	{1,2,3,4,5}	{}
V_1←2	{}	{2,3,4,5}	{}
V_2←3	{V_1}	{3,4,5}	{}
V_{23}←2	{V_{20}}	{2,3,4,5}	{}
V_{22}←4	{V_{20}, V_{23}}	{4,5}	{V_{20}, V_2}
V_0←1	{}	{1,2,3,4,5}	{}
V_3←4	{V_1 ,V_2 ,V_0}	{4,5}	{}
V_4←5	{V_1, V_2, V_0, V_3 }	{5}	{}
V_5	{V_0}	{}	{V_1}
V_6	{V_1}	{2}	{}
V_8	{V_{23}}	{2}	{}
V_{11}	{V_4}	{5}	{}
V_{12}	{V_{22}}	{4}	{}
V_{15}	{V_{23}}	{1,3}	{}
V_{21}	{V_{20}, V_2}	{3}	{}
V_{24}	{V_{20}, V_{23} , V_{22}}	{3,5}	{}

Figure 4: No-good generation example.

- Attempting the assignment $V_2 \leftarrow 1$ causes a domain wipe-out for the unassigned variable V_0. Thus, $confSet(V_2) = confSet(V_2) \cup pastFC(V_0) = \emptyset$. The assignment $V_2 \leftarrow 3$ removes the value 3 from $D(V_3), D(V_4)$ and the values in $\{2, 4, 5\}$ from $D(V_{21})$. Thus, the following updates are made: $pastFC(V_3) = pastFC(V_4) = \{V_1, V_2\}$ and $pastFC(V_{21}) = \{V_{20}, V_2\}$.

- The assignment $V_{23} \leftarrow 2$ removes the value 2 from $D(V_{22})$ and $D(V_{24})$, the values in $\{2, 4, 5\}$ from $D(V_{15})$, and the values in $\{1, 3, 4, 5\}$ from $D(V_8)$. As a result, the following updates are made: $pastFC(V_8) = pastFC(V_{15}) = \{V_{23}\}$ and $pastFC(V_{22}) = pastFC(V_{24}) = \{V_{20}, V_{23}\}$.

- The assignment $V_{22} \leftarrow 3$ causes a domain wipe-out for the unassigned variable V_{21}. Thus, $confSet(V_{22}) = confSet(V_{22}) \cup pastFC(V_{21}) = \{V_{20}, V_2\}$. The assignment $V_{22} \leftarrow 4$ removes the value 4 from $D(V_{24})$ and the values in $\{1, 2, 3, 5\}$ from $D(V_{12})$. Thus, the following updates are made: $pastFC(V_{12}) = \{V_{22}\}$ and $pastFC(V_{24}) = \{V_{20}, V_{23}, V_{22}\}$.

- The assignment $V_0 \leftarrow 1$ prunes the value 1 from $D(V_3)$ and $D(V_4)$ and the values in $\{1, 3, 4, 5\}$ from $D(V_5)$. Thus, the following updates are made: $pastFC(V_3) = pastFC(V_4) = \{V_1, V_2, V_0\}$ and $pastFC(V_5) = \{V_0\}$.
- The assignment $V_3 \leftarrow 4$ removes the value 4 from $D(V_4)$. Thus, $pastFC(V_4) = \{V_1, V_2, V_0, V_3\}$.
- The assignment $V_4 \leftarrow 5$ removes the values in $\{1, 2, 3, 4\}$ from $D(V_{11})$. Thus, $pastFC[V_{11}] = \{V_4\}$.
- The assignment $V_5 \leftarrow 2$ causes a domain wipe-out in the unassigned variable V_6. Thus, $confSet(V_5) \leftarrow confSet(V_5) \cup pastFC(V_6) = \{V_1\}$. Because $D(V_5) - \{2\} = \emptyset$, we must backtrack.

Note that values in $\{1, 3, 4, 5\}$ were pruned from $D(V_5)$ by the assignment $V_0 \leftarrow 1$, the reason for these prunings is recorded in $pastFC(V_5)$. Furthermore, the remaining domain value 2 in $D(V_5)$ is pruned as a result of a future domain wipe-out by V_6 whose domain had already been pruned by the assignment $V_1 \leftarrow 2$, the reason the pruning of the value 2 is then recorded in $confSet(V_5)$. Thus, the reason for the backtrack is encapsulated by $\{confSet(V_5) \cup pastFC(V_5)\} = \{V_0 \leftarrow 1, V_1 \leftarrow 2\}$.

At each backtrack in FC-CBJ, the no-good $ng \leftarrow confSet(V_i) \cup pastFC(V_i)$, where V_i is the current variable, is the reason for the backtrack, and should be added to the current list of no-goods. To do so, we alter the original pseudocode of FC-CBJ-UNLABEL [**?**], provided for reference as Algorithm 11 in Section A in the appendix. The new no-good must be recorded before the $confSet(i)$ and $pastFC(i)$ lists are updated. We introduce FC-CBJ-UNLABEL+NG (Algorithm 1), which calls in lines 8 and 9 BUILDNG (Algorithm 5) to generate the new no-good.

5 Blame Variables

As stated above, for a variable V_i, $confSet(V_i)$ and $pastFC(V_i)$ contain the reasons that each domain value of V_i was pruned. In FC-CBJ-UNLABEL+NG (Algorithm 1), the current instantiation, $val[V_h]$ to the past variable, V_h, is pruned. However, $val[V_h]$ is ***not*** added to the reductions list. Furthermore, neither $confSet(V_h)$ nor $pastFC(V_h)$ is updated to reflect the reason of pruning the domain value $val[V_h]$. Given that $V_h \leftarrow val[V_h]$ is consistent with the current partial solution of variables $\{V_0, V_1, \ldots, V_{h-1}\}$ and did not explicitly cause a domain wipe-out among the uninstantiated variables $\{V_{h+1}, V_{h+2}, \ldots, V_n\}$, it is not immediately obvious *why* this value was pruned. However, not assigning a reason for the pruning of this domain value can result in unnecessary prunings immediately after the backtrack is complete.

For example, consider the Zebra puzzle again. It is defined by the set of variables $V = \{V_0, V_1, \ldots, V_{25}\}$ with domains $D(V_0) = D(V_1) = \ldots = D(V_{25}) = \{1, 2, 3, 4, 5\}$.

- Consider the following the current partial solution: $\{V_0 \leftarrow 1$, $V_{13} \leftarrow 3$, $V_5 \leftarrow 2$, $V_6 \leftarrow 3$, $V_{23} \leftarrow 5$, $V_8 \leftarrow 5$, $V_{12} \leftarrow 1$, $V_{16} \leftarrow 3$, $V_3 \leftarrow 2$, $V_4 \leftarrow 5$, $V_{19} \leftarrow 2$, $V_{11} \leftarrow 5$, $V_{17} \leftarrow 5$, $V_{10} \leftarrow 4\}$. Consider also that the uninstantiated variable V_7 has the following sets:

$$\begin{aligned}
&confSet(V_7) = \{V_5, V_6, V_8, V_{10}\}\\
&pastFC(V_7) = \{V_8, V_6, V_5\}\\
&D(V_7) = \{4\}.
\end{aligned}$$

- The instantiation $V_7 \leftarrow 4$ causes a domain wipe-out in a future variable and the no-good, $ng_1 \leftarrow \{V_{10} \leftarrow 4, V_8 \leftarrow 5, V_6 \leftarrow 3, V_5 \leftarrow 2\}$ is added to $ngList$ at the unlabel.

- Because V_{10} is the deepest variable in $confSet(V_7) \cup pastFC(V_7)$, we backtrack to V_{10}, which has only one value (i.e., value 2) left in its domain. Because $V_{10} \leftarrow 4$ is consistent with the past, the value 4 is not added to the reduction list of V_{10}, but it is also pruned from the current domain of V_{10} because the assignment $V_{10} \leftarrow 4$ has just been undone by the unlabel (note that this is the issue that needs be addressed). Because V_{10} has been identified as the past variable, $confSet(V_{10}) \leftarrow confSet(V_{10}) \cup confSet(V_7) \cup pastFC(V_7)$.

- At the next label, the instantiated $V_{10} \leftarrow 2$ causes a domain wipe-out with a future variable and, because the value 2 is the last domain value left in $D(V_{10})$ we must unlabel V_{10}.

- In unlabel, a second no-good, ng_2, is learned, and we backtrack to V_{11}. The domain of V_{10} is updated to $D(V_{10}) \leftarrow \{2, 4\}$ although the variable value pair $V_{10} \leftarrow 4$ violates the ng_1. This inconsistency will be discovered at the next label.

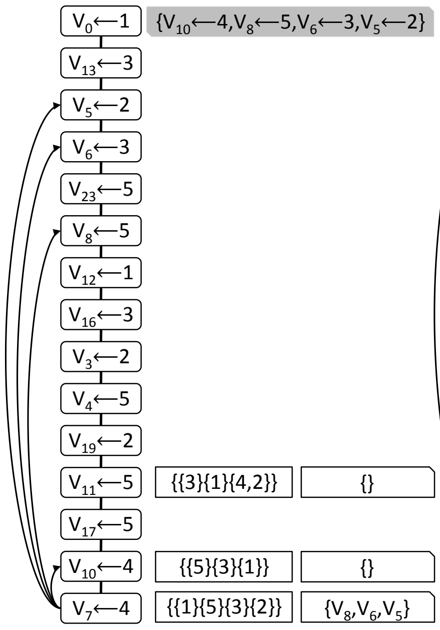


Figure 5: Search at first unlabel.

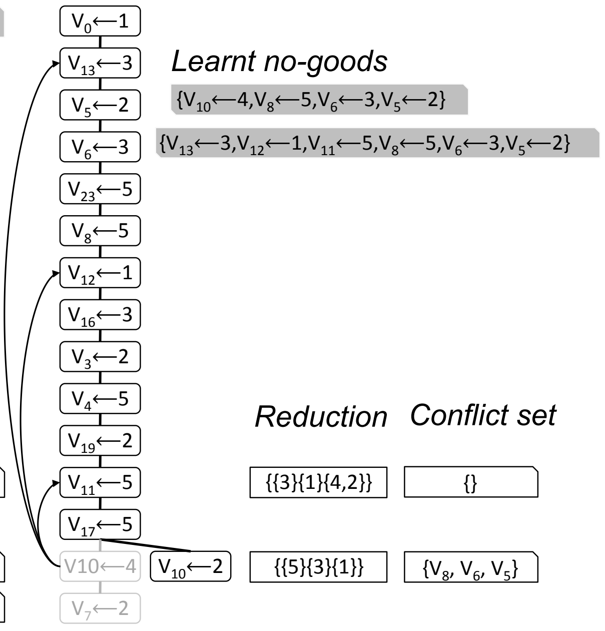


Figure 6: Search at second unlabel. Notice that the value 4 is returned to the domain of V_{10} although it violates the first no-good.

To avoid this unnecessary propagation of ng_1, the domain value 4 must be added to the reduction list of V_{10} as soon as the assignment $V_{10} \leftarrow 4$ is undone and before we make the assignment $V_{10} \leftarrow 2$. We add the deepest variable of the no-good, in this case ng_1, to $pastFC(V_{10})$. This deepest variable of the no-good that is credited for the pruning of $V_{10} \leftarrow 4$ is referred to as the *blame variable*. Adding the blame variable to $pastFC$ of the current variable in the unlabel prevents unnecessary propagations of the no-goods at the next label. In this case, the value 4 is not restored to $D(V_{10})$ at the second backtrack, which will prevent ng_1 from being propagated at the next unlabel.

Figures 5 and 6 depict this situation. The current partial solution is shown in the left column. The variables in $pastFC(V_i)$ are depicted by the back arrows from variable V_i to previously instantiated variables. The rectangular boxes in the middle column represent the domain values that have been pruned by past variables (i.e., the reductions list of V_i). The rectangles in the right column depict the variables in $confSet(V_i)$. At the top of each figure, the list of no-goods in $ngList$ are given in the shaded boxes.

It is important to note that blame variables are *only* useful for avoiding redundant propagations. While the blame variable provides the correct point along the search path at which the pruned value should be added back to the domain of the variable, the blame variable does *not* fully encapsulate the reason why the domain value was pruned. The full reason that the value was pruned is the no-good itself. The remaining variables of the no-good must be stored in another data structure that maps the domain values of a variable to a list of no-good variables causing the filtering of the domain values. We refer to this structure as $pastNgVariables[V_i, d_i]$, where V_i is the variable whose domain value was pruned implicitly by the no-good and $d_i \in D(V_i)$ is the domain value that was pruned from the domain of V_i. When a new no-good is generated, we must add the variable-value pairs associated to the variables in $pastNgVariables[V_i, val[V_i]]$. Furthermore, when a variable-value pair $V_i \leftarrow val[V_i]$ has been uninstantiated, the list $pastNgVariables[V_i, val[V_i]]$ can be cleared.

6 Algorithms for Learning No-Goods in FC-CBJ

In order to incorporate no-good learning in FC-CBJ, we provide the following updated or new algorithms:

- FC-CBJ-UNLABEL+NG (Algorithm 1) is an updated version of FC-CBJ-UNLABEL originally proposed by **?** [**?**], which is reported as Algorithm 11 in Section A for reference.

- FINDBLAMEVARIABLE (Algorithm 2) determines the blame variable based on a particular no-good, ng; the blame variable cannot be a particular variable var.

- ADDBLAMETOREDUCTION (Algorithm 3) adds the blame variable to the reductions list of the variable whose domain is being pruned if the blame variable is not already part of the pruned variable's past variable set. That is, $prev$ is the previous variable that the blame variable is being added to, $blame$ is the blame variable that is being added to $pastFc(prev)$, value is the domain value of $prev$ that $blame$ pruned.

- REMOVEPASTNGVARIABLES (Algorithm 4) resets the past no-good variables of the variable V_j.

- BUILDNG (Algorithm 5) creates the new no-good based upon the following sets of the input variable V_i: $pastFC(V_i)$, $confSet(V_i)$, and $pastNgVariables[V_i, d_k]$ where $k = 0, 1, \ldots, |D(V_i)|$.

The global variables *propagateWipeout* and *nogoodConflict* are Boolean flags, and *ngCreatedFrom* stores a CSP variable.

Algorithm 1: FC-CBJ-UNLABEL+NG(h).

```
Input: $index$ index of current variable, $consistent$ indicates the status of
        the partial solution
Output: $h$ the index of the next variable to instantiated
1  $i \leftarrow index$;
2  if $\neg$propagateWipeout $\wedge\neg$nogoodConflict then
3      $V_h \leftarrow$ DEEPESTVAR(UNION($confSet(V_i), pastFC(V_i)$))
4  else
5      $i \leftarrow propagateBlameVar$
6      $h \leftarrow propagateBlameVar$
7      $index \leftarrow$# instantiated variables
8  if $\neg$propagateWipeout then $ng \leftarrow$ BUILDNG($V_i$)
9  else $ng \leftarrow$BUILDNG($ngCreatedFrom$)
10 $V_{blm} \leftarrow$ FINDBLAMEVARIABLE($ng$)
11 ADD-NO-GOOD($ng$)
12 $confSet(V_h) \leftarrow$ UNION($confSet(V_h), confSet(V_i), pastFC(V_i)$)) $\setminus\{V_h\}$
13 UPDATENGLISTS($V_i$)
14 for $j \leftarrow index$ DOWNTO $h + 1$ do
15     $confSet(V_j) \leftarrow \{\emptyset\}$
16     UNDOREDUCTIONS($V_j$)
17     UPDATECURRDOM($V_j$)
18     REMOVEPASTNGVARIABLES($V_j$)
19 UNDOREDUCTIONS($V_h$)
20 REMOVEPASTNGVARIABLES($V_h$)
21 $D^{curr}[V_h] \leftarrow$ REMOVE($val[V_h], D^{curr}[V_h]$)
22 $consistent \leftarrow D^{curr}[V_h] \neq nil$
23 UPDATECURRDOM($V_i$)
24 REMOVEPASTNGVARIABLES($V_i$)
25 UPDATENGLISTS($V_h$)
26 ADDBLAMETOREDUCTION($V_h$, $blame$, $val[h]$)
27 foreach $\{(var, value)\} \in newNg$ do
28     if $var \neq blame$ then
29         $PastNgVariables[V_h, val[V_h]] \leftarrow$
           $PastNgVariables[V_h, val[V_h]] \cup \{var\}$
30 return $h$
```

Algorithm 2: FINDBLAMEVARIABLE(ng, var).

```
   Input: $ng$ a no-good set within which to find the blame variable, $var$ a
          variable that should not be blamed.
   Output: $blame$ the deepest variable in $ng$ credited for the pruning
1  $blame \leftarrow$ variable of first variable-value pair in $ng$
2  foreach $\{(ngVar, value)\} \in ng$ do
3      if DEPTH($blame$) $<$ DEPTH($ngVar$) and $blame \neq var$ then
4          $blame \leftarrow ngVar$
5  return $blame$
```

Algorithm 3: ADDBLAMETOREDUCTION($prev, blame, value$).

```
   Input: $prev, blame$ two variables; $value$: a value for $prev$
1  $reduction \leftarrow nil$
2  if $blame \in pastFC(prev)$ then $reduction \leftarrow reductions[prev, blame]$
   else $pastFC(prev) \leftarrow pastFC(prev) \cup \{blame\}$
3  $reduction \leftarrow reduction \cup \{value\}$
4  $reductions[prev, blame] \leftarrow reduction$
```

Algorithm 4: REMOVEPASTNGVARIABLES(V_j)

```
   Input: $j$ the index of the variable to reset the pastNgVariable set.
1  foreach $d_j \in D^{curr}[V_j]$ do
2      $pastNgVariables[V_j, d_j] \leftarrow \{\emptyset\}$
```

Algorithm 5: BUILDNG(V_i).

```
   Input: $V_i$: the variable from which the no-good is built
   Output: $ng_{new}$ : the resulting no-good
1  $vars_{ng} \leftarrow confSet(V_i) \cup pastFC(V_i)$
2  foreach $d_i \in D(V_i)$ do
3      $vars_{ng} \leftarrow vars_{ng} \cup pastNgVariables[V_i, d_i]$
4  $ng_{new} \leftarrow \emptyset$
5  foreach $var \in vars_{ng}$ do $ng_{new} \leftarrow ng_{new} \cup \{(var, value(var))\}$
6  return $ng_{new}$
```

7 Reducing the List of No-goods

To reduce space requirements, each time a new no-good, ng, is generated, it is compared with each of the existing no-goods $ng_i \in ngList$. If $ng_i \subseteq ng$, ng is not added to $ngList$. Similarly, if $ng \subset ng_i$, ng_i is removed from $ngList$, and ng is added to $NgList$. If ng is unrelated to all of the no-goods in $ngList$, that is $(ng \not\subset ng_i) \wedge (ng_i \not\subset ng)$ for each $ng_i \in ngList$, ng is added to $ngList$. This method of adding no-goods ensures that the smallest known set of no-goods is maintained throughout the search without searching for the minimal no-goods [**?**]. If the new no-good is unary, the domain value of the no-good is removed from the initial domain of the variable associated with this no-good. Otherwise, two watchers are assigned to the first two variables in the no-good. These watcher variables assist in the propagation of the no-good. ADDNOGOOD (Algorithm 6) determines whether or not the new no-good ng must be added to $ngList$ and adds the corresponding watcher when appropriate. The watcher is denoted by the tuple $(watcher1_{ng}, watcher2_{ng})$ where $watcher1_{ng}$ and $watcher2_{ng}$ are composed of the variables associated to the first and second tuples of the no-good.

Algorithm 6: ADDNOGOOD(ng).

```
Input: $ng$ the new no-good generated at the FC-CBJ-Unlabel
1  foreach $ng_i \in ngList$ do
2      if $ng_i \subset ng$ then return
3      if $ng \subset ng_i$ then $ngList \leftarrow ngList \setminus \{ng_i\}$
4  $ngList \leftarrow ngList \cup \{ng\}$
5  if $|ng| = 1$ then
6      $ngVar \leftarrow ng[0]_{variable}$
7      $ngValue \leftarrow ng[0]_{value}$
8      $D(ngVar) \leftarrow D(ngVar) \setminus \{ngValue\}$
9  else
10     $watcher1_{ng} \leftarrow$ first variable-value pair in $ng$
11     $watcher2_{ng} \leftarrow$ second variable-value pair in $ng$
12     $watcherList \leftarrow watcherList \cup \{(watcher1_{ng}, watcher2_{ng})\}$
13 return
```

This reduced list of no-goods can be divided into two sublists: *open no-goods* and *used no-goods*. An open no-good is a no-good that has not been propagated. In contrast, a used no-good is one that has already been propagated. Thus, only the no-goods in the open no-good list need to be checked for propagation. Both lists,

$openNg$ and $usedNg$ must be updated when a no-good has been propagated or when a variable has been uninstantiated. When a no-good is propagated, the no-good is simply moved from $openNg$ to $usedNg$. However, when a variable V_i is uninstantiated, we must move the no-goods containing the tuple $\{V_i, val[V_i]\}$ from $usedNg$ to $openNg$ because the no-good can now be propagated. The pseudocode for updating these two lists during an unlabel of V_i is given in UPDATENGLISTS (Algorithm 7).

Algorithm 7: UPDATENGLISTS(V_i).

```
Input: Variable $V_i$ is being uninstantiated
foreach $ng \in usedNg$ do
    if $\{(V_i, val[i])\} \in \{ng\}$ then
        $openNg \leftarrow openNg \cup \{ng\}$
        $usedNg \leftarrow usedNg \setminus \{ng\}$
        $watcher1_{ng} \leftarrow ng[0]_{variable}$
        $watcher2_{ng} \leftarrow ng[1]_{variable}$
        $watcherList \leftarrow watcherList \cup \{(watcher1_{ng}, watcher2_{ng})\}$
```

8 No-good Propagation

In order to ensure that the no-goods are not in violation before a new variable is instantiated, the no-goods must be propagated at the beginning of FC-CBJ-LABEL, which is the only modification done to the original FC-CBJ-LABEL function [**?**]. Two *watchers* are assigned to each no-good to assist in the propagation. In the propagate method, both watchers for each no-good are checked in order to determine if the no-good must be propagated or if a watcher needs to be updated. When the variable-value pair of a watcher has been instantiated, we say that the watcher is active. This technique is inspired from SAT solvers. Two cases can occur:

1. Both watchers of a no-good remain inactive.
2. Only one watcher of a no-good has become active. Further, in this case,
    (a) Another variable in the no-good is available to become a watch variable.
    (b) No other variables in the no-good are available to become watch variables.

In the first case, neither watcher is incremented. In case (2a) the watcher is incremented to the next available variable and we continue with the next no-good. In case (2b), the variable-value pair associated with the inactive watcher is pruned, the no-good is removed from $openNg$ and added to $usedNg$. A backtrack is also necessary in the case of (2b) when the domain value pruned is the last remaining domain value of the variable or when the pruned variable-value pair has already been instantiated (i.e., the no-good is in conflict).

A no-good is *propagated* when only one variable-value pair in the no-good is not part of the current partial solution or when the no-good is in conflict with the current partial solution. The variable whose domain value is pruned is denoted V_{prn} and the value being removed from its domain prune val_{prn}. At each pruning, we must update the reductions list and $pastFC$ of the pruned variable. Like in FC-CBJ-UNLABEL+NG (Algorithm 1), we must determine V_{blm}, the variable responsible for the pruning. The blame variable will be the deepest instantiated variable in the no-good. If val_{prn} has not already been pruned from the domain of V_{prn}, we can simply add V_{blm} to $pastFC(V_{prn})$. However, it could be the case that the domain value that is to be pruned from the domain of V_{prn} has already been pruned by another variable during the search. If this is the case, we still must determine the blame variable. If the new blame variable is deeper then the variable currently credited for the pruning, V_{blm}^{curr}, of the pruned value then if the search backtracks past the current blame variable but not past the new blame variable, the no-good will be propagated again. To avoid this redundant propagation, the reductions list must be updated to credit the blame variable for the pruning if the new blame variable is deeper then the current blame variable. As in Section 5, the blame variable does not completely encapsulate the reason for the pruning, thus we must add the rest of the no-good variables to $pastNgVariables[V_{prn}, val_{prn}]$.

PROPAGATE (Algorithm 8) provides the pseudocode for the propagate method. In this algorithm, we check each of the watchers to determine if a no-good needs to be propagated. If propagation is necessary and the variable-value pair has already been pruned, we determine the new blame variable. If a backtrack is necessary the remaining variables of the no-good are added to the $pastNgVariables$ of the variable-value pair being pruned. RESETBLAME (Algorithm 9) determines if the new blame variable V_{blm} should replace the current variable credited for the pruning of the variable-value pair (V_{prn}, val_{prn}). If V_{blm} should be credited for the pruning, the reductions list and $pastFC(V_{prn})$ list of V_{prn} are updated accordingly. REMOVEFROMWATCHERLIST (Algorithm 10) removes a particular watcher $watcher$ at index $index$ in $ngList$.

Algorithm 8: PROPAGATE().

```
    Output: $true$ if backtrack is needed, $false$ otherwise.
1  foreach $\{(watcher1, watcher2)\} \in watcherList$ do
2      $ng \leftarrow$ no-good associated to watcher
3      $i \leftarrow$INDEXOF$(ng)$
4      $W_1 \leftarrow$ ISACTIVE$(watcher1)$
5      $W_2 \leftarrow$ ISACTIVE$(watcher2)$
6      if $W_1$ then $watcher1 \leftarrow watcher2$
7      else if $W_2$ then $watcher_2 \leftarrow$ NEXTAVAILABLEVAR$(ng)$
8      if $(W_1 \vee W_2) \wedge watcher2 = nil$ then
9          $val_{prn} \leftarrow value(watcher1)$
10         $V_{prn} \leftarrow variable(watcher1)$
11         if $val[V_{prn}] = val_{prn}$ then
12             $V_{prn} \leftarrow$ deepest variable in $ng$
13             $V_{blm} \leftarrow$ FINDBLAMEVAR$(ng, V_{prn})$
14             ADDBLAMETOREDUCTION($V_{prn}$,$V_{blm}$, $val[V_{prn}]$)
15             $D^{curr}[V_{prn}] \leftarrow D^{curr}[V_{prn}] \setminus \{val[V_{prn}]\}$
16             $V_{blm}^{prop} \leftarrow V_{prn}$
17             if $D^{curr}[V_{prn}] = \emptyset$ then $V_{blm}^{prop} \leftarrow V_{blm}$
18             REMOVEFROMWATCHERLIST$(\{(watcher1, watcher2)\}, i)$
19             $nogoodConflict \leftarrow true$
20             return *true*
21         $V_{blm} \leftarrow$ FINDBLAMEVAR$(ng, V_{prn})$
22         $V_{blm}^{curr} \leftarrow V_{blm}$
23         if $val_{prn} \notin D^{curr}[V_{prn}]$ then $V_{blm} \leftarrow$
           RESETBLAME$(V_{prn}, val_{prn}, V_{blm})$
24         if $V_{blm} = V_{blm}^{curr}$ then
25             foreach $d_i \in D^{curr}[V_{prn}]$ do
26                 $pastNgVariables[V_{prn}, d_i] \leftarrow$
                   $pastNgVariables[V_{prn}, d_i] \cup \{V_{blm}\}$
27         REMOVEFROMWATCHERLIST$(\{(watcher1, watcher2)\}, i)$
28         $D^{curr}[V_{prn}] \leftarrow D^{curr}[V_{prn}] \setminus \{val_{prn}\}$
29         if $D^{curr}[V_{prn}] \leftarrow \emptyset$ then
30             $ngCreatedFrom \leftarrow V_{prn}$
31             $propagateVar \leftarrow V_{blm}$
32             $propagateWipeout \leftarrow true$
33             return *true*
34 return *false*
```

Algorithm 9: ResetBlame($V_{prn}, val_{prn}, V_{blm}$).

```
Input: $V_{prn}$ the variable that the blame variable is being changed for, $val_{prn}$
       specific domain value blame variable is associated to, $V_{blm}$ current
       blame variable based on no-good
Output: $V_{blm}^{new}$ the updated blame variable
1  $V_{blm}^{curr} \leftarrow nil$
2  for each $var \in pastFC(V_{prn})$ do
3      if $val_{prn} \in reductions[V_{prn}, var]$ then
4          $V_{blm}^{curr} \leftarrow var$
5          break
6  $V_{blm}^{new} \leftarrow V_{blm}^{curr}$
7  if Depth($V_{blm}^{curr}$) <Depth($V_{blm}$) then
8      $reductions[V_{prn}, V_{blm}^{curr}] \leftarrow reductions[V_{prn}, V_{blm}^{curr}] \setminus \{val_{prn}\}$
9      AddBlameToReductions($V_{prn}, V_{blm}, val_{prn}$)
10     $V_{blm}^{new} \leftarrow V_{blm}$
11 return $V_{blm}^{new}$
```

Algorithm 10: RemoveFromWatcherList($watcher, index$)

```
Input: $watcher$ the watcher to be removed from the watcherList, $index$ the
       index of the no-good
Output: No return value
1  $watcherList \leftarrow watcherList \setminus \{watcher\}$
2  $usedNGs \leftarrow usedNGs \cup \{nogoodList[index]\}$
3  $openNGs \leftarrow openNGs \setminus \{nogoodList[index]\}$
4  return
```

9 Experiments

In this section, we compare the performance of FC-CBJ and FC-CBJ+RedNG over binary CSP taken from benchmark from the CP Solver Competition [1] as well as randomly generated problem instances.

[1]http://www.cril.univ-artois.fr/CPAI09/.

9.1 CSP Parameters & Phase Transition

These random instances are generated using the Model B generator [?]. An instance is fully described by the following parameters:

- n is the number of variables.
- a is the number of values per variable, or domain size. All variables have the same domain size.
- d is the constraint density. The constraint density of a binary CSP is $d = \frac{e}{e_{max}}$ where $e_max = \frac{n(n-1)}{2}$, e is the number of constraints.
- $t = \frac{|forbidden\ tuples|}{|all\ tuples|}$ is the constraint tightness. All constraints have the same tightness value.

Randomly generated problems have been qualitatively characterized by the macroscopic behavior encountered by *any* algorithm for solving them, called the *phase transition* phenomenon. This phenomenon links both the likelihood of the existence of a solution and the cost of finding it to the value of an *order parameter*. For CSPs, this order parameter is the density d or the tightness t.

- For small values of the order parameter, the problem is likely to have many solutions and they are on average easy to find for most algorithms.
- For large values of the order parameter, the problem is likely to have no solutions and this fact is on average quickly determined for most algorithms.
- Around a critical value of the order parameter, known as the *phase transition*, typically, half of the instances are solvable and half are inconsistent. Very few solutions exist for a given instance, and most the paths in the search tree are 'almost solutions' with means partial solutions that fail to complete into complete solutions. For that reason, almost all algorithms encounter a peak of their cost around the critical values of the order parameter.

9.2 Random Instances with Variable Tightness

We tested our algorithm, FC-CBJ+REDNG, against FC-CBJ on 450 random instances with the following parameters: $\langle n = 32, a = 8, t \in \{0.1, 0.2, \ldots, 0.9\}, d = 20\%\rangle$. We collected the following data for both algorithms: number of nodes visited, number of constraint checks, and CPU time in milliseconds. Nodes visited refers to the number of attempted variable instantiations, constraint checks refers to the number of constraints checked when attempting to instantiate the variables, and CPU time measures the amount of search time required in milliseconds. For FC-CBJ+REDNG we collected the following additional data with regards to the no-goods: number of reduced no-goods remaining after termination of search, number of variable-value pairs pruned by the no-goods, max, min, median, and average size of reduced no-good, and the largest no-good learnt throughout search.

Figures 7, 8, and 9 compare the performance of FC-CBJ+REDNG to FC-CBJ in terms of nodes visited, constraint checks, and CPU time. Figure 10 depicts parameters of interest regarding the no-goods: number of reduced no-goods, number of values pruned by no-goods, and the sizes of no-goods. At low tightness, where the instances are easily solvable, and at high tightness, where instances are easily found to be unsolvable, only a few no-goods are generated and pruning power is limited. Notice the largest and greatest number of no-goods occur at the middle tightnesses. Also, at the middle tightnesses we see the most number of prunings performed by the reduced no-goods.

We see fairly impressive performance improvement of FC-CBJ+REDNG with respect to nodes visited and constraint checks particularly at the phase transition. The number of no-goods, values pruned by no-goods and sizes of no-goods increase at $t = 4$ and $t = 5$, the tightnesses where we see the most improvement of FC-CBJ+REDNG over FC-CBJ. The no-goods, particularly at those tightnesses, are being propagated frequently and pruning subtrees of the search space thus decreasing the number of the nodes visited and constraint checks. We also save on CPU time in FC-CBJ+REDNG, however the savings are not as dramatic as the savings in constraint checks and nodes visited. Particularily at $t = 4$ and $t = 5$, we spend a significant amount of time processing the large number of no-goods. The small savings in CPU time could also be an effect implementation or due to cost of reducing the set of no-goods. Howver, we see an overall improvement of FC-CBJ+REDNG over FC-CBJ.

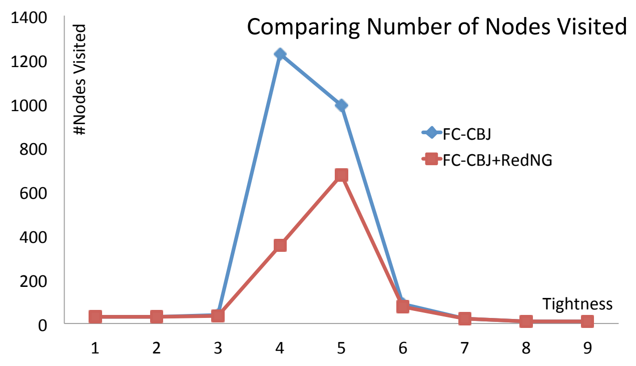


Figure 7: FC-CBJ vs. FC-CBJ+RedNG: Comparing the number of nodes visited for $\langle n = 32, a = 8, t \in \{0.1, 0.2, \ldots, 0.9\}, d = 20\%\rangle$.

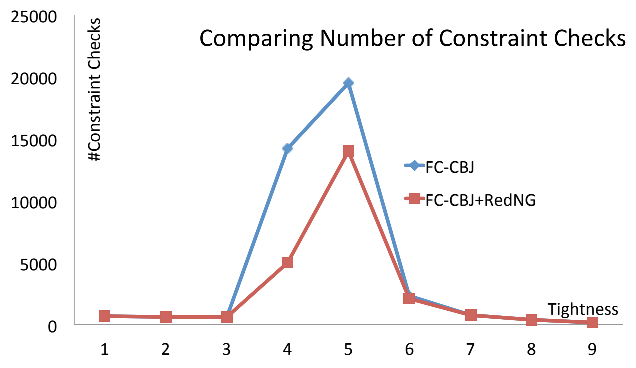


Figure 8: FC-CBJ vs. FC-CBJ+RedNG: Comparing the number of constraint checks for $\langle n = 32, a = 8, t \in \{0.1, 0.2, \ldots, 0.9\}, d = 20\%\rangle$.

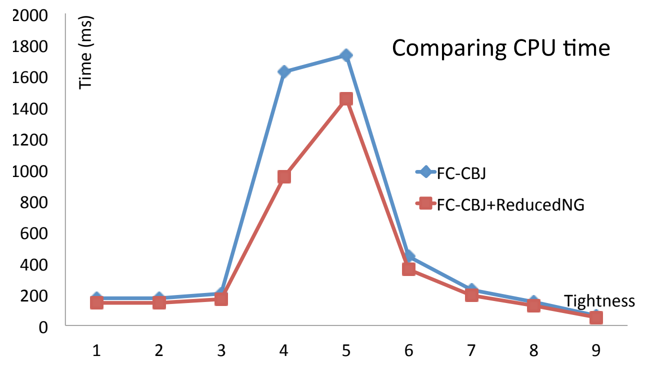


Figure 9: FC-CBJ vs. FC-CBJ+RedNG: Comparing the CPU time for $\langle n = 32, a = 8, t \in \{0.1, 0.2, \ldots, 0.9\}, d = 20\%\rangle$.

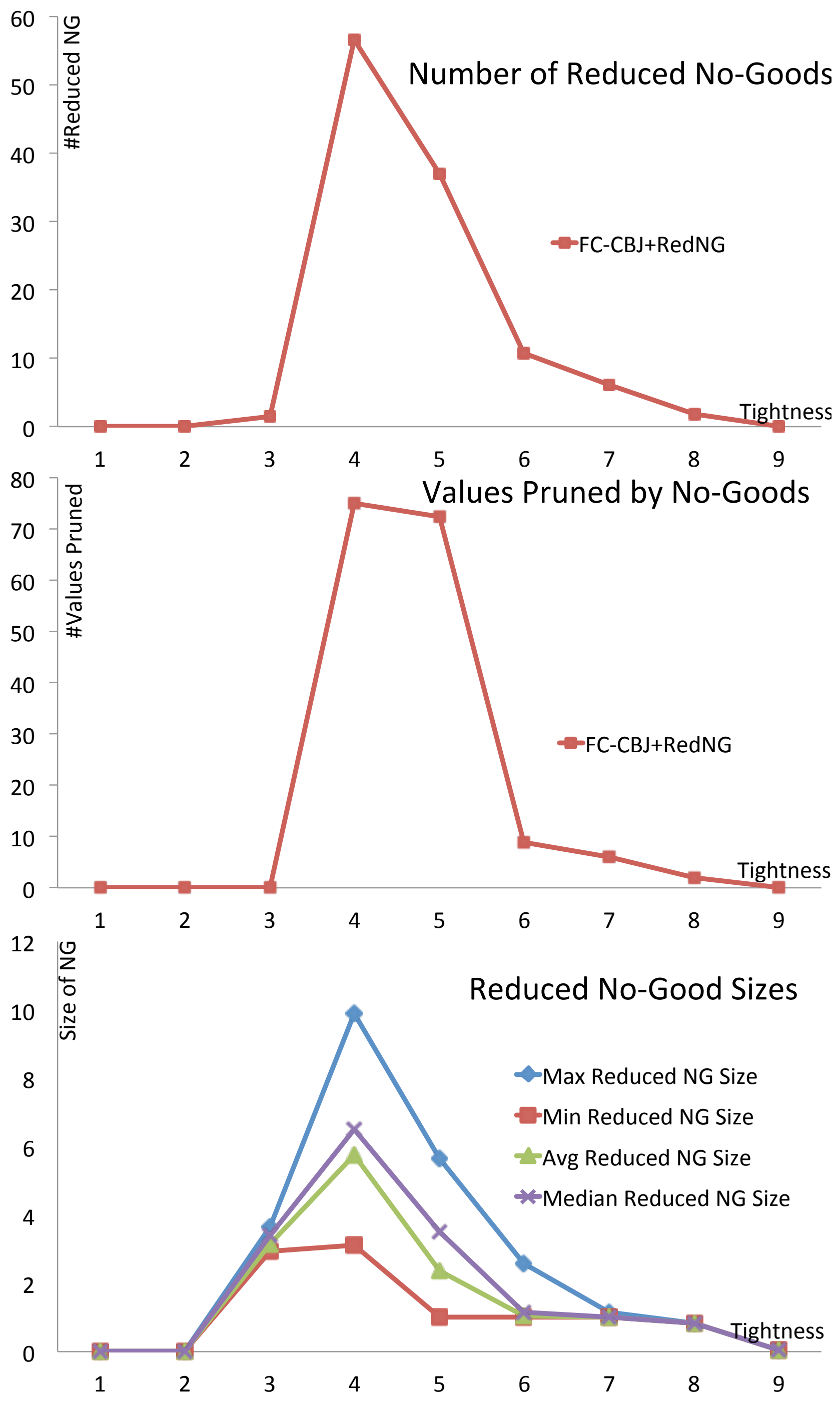


Figure 10: FC-CBJ+RedNG on $\langle n = 32, a = 8, d = 20\%, t\rangle$: number of reduced no-goods stored, filtering by no-goods, and size of stored no-goods.

9.3 Random Instances with Variable Density and Tightness

In this section we experimented with 810 randomly generated instances with parameters $\langle n = 40, a = 10, d, t \rangle$, 10 instances per tightness-density combination. For each of those instances the same data as in Section 9.2 was collected. Figures 11, 12, 13 depict the ratio of FC-CBJ to FC-CBJ+REDNG with respect to the number nodes visited, constraint checks, and CPU time. Additionally Figures 14, 15, and 16 display the number of values pruned by the no-goods, the number of reduced no-goods generated, and the median size of reduced no-goods. Each figure contains a 3D graph as well as two cross section graphs.

Like in Section 9.2 we see the most performance increase with respect to constraint checks and nodes visited. There are five and nine tightness-density combinations in which FC-CBJ out-performed FC-CBJ+REDNG on nodes visited and constraint checks respectively. Otherwise, FC-CBJ+REDNG performed as well or better then FC-CBJ. In the best cases FC-CBJ+REDNG had slightly more than two hundred times fewer nodes visited and constraint checks then FC-CBJ. In the worst cases FC-CBJ+REDNG had only slightly over one time as many constraint checks and nodes visisted. Thus, even when FC-CBJ outperformed FC-CBJ+REDNG it was by a much smaller margin then when FC-CBJ+REDNG outperformed FC-CBJ.

In terms of CPU time, the majority of the time FC-CBJ+REDNG outperforms FC-CBJ, however in the best case FC-CBJ+REDNG is only ten times faster than FC-CBJ. In the worst case ($d = 40\%$, $t = 30\%$), FC-CBJ+REDNG is thirty-four times slower than FC-CBJ. The no-good graphs show that the greatest number of reduced no-goods and pruning done by these no-goods occurs at this worst case data point for CPU time. Thus, the slow-down in CPU time could due to the frequent processing of a large number of no-goods or implementation of the no-good processing functions.

With respect to the no-goods, the largest reduced no-goods appear in at low densities ($d = 10\%$ or $d = 20\%$). The median size of no-goods is close to 1 for the majority of density-tightness combinations. The largest number of reduced no-goods were learned at $d = 10\%$, $d = 20\%$, and $d = 30\%$. The large median size at $d = 10\%$ and $d = 20\%$ could contribute to the large amount of reduced no-goods at those densities because no-goods with a large number of variable-value pairs will be less likely to be a subset the existing no-goods. In terms of CPU time, the savings resulting the pruning of a large set of no-goods could be outweighed by the cost of processing that large set at each label.

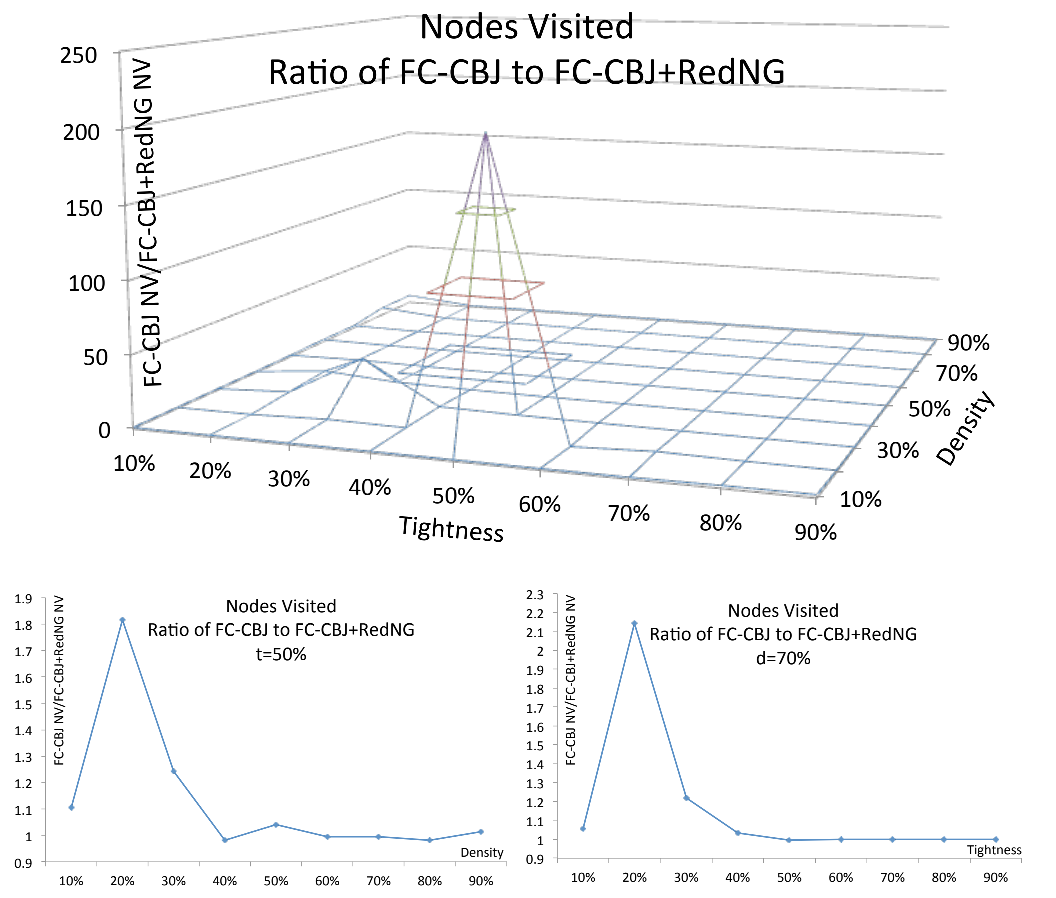
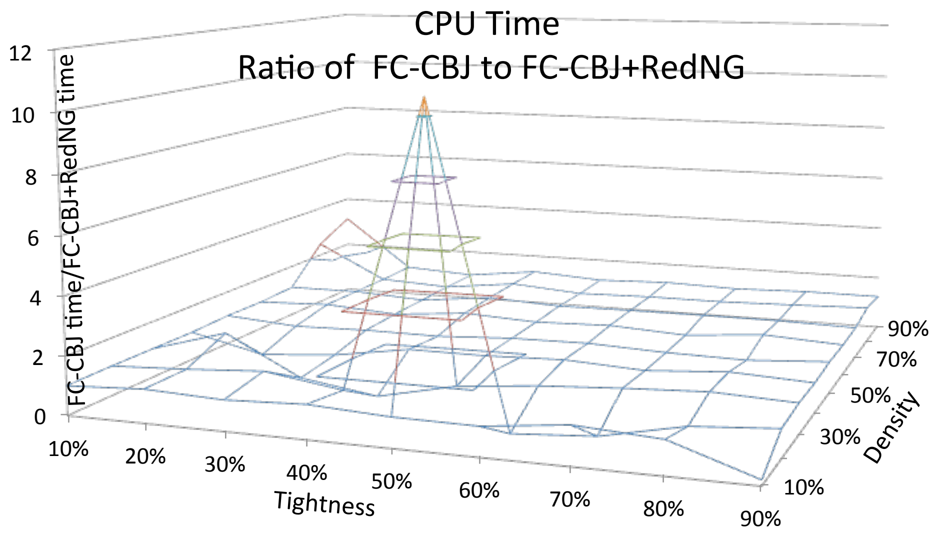
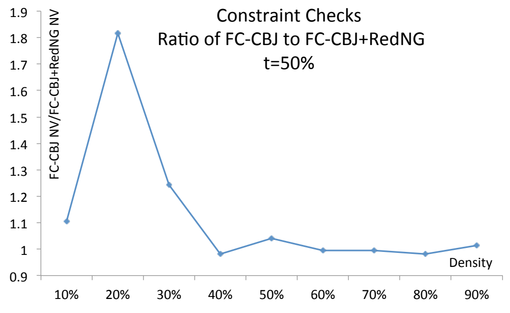


Figure 11: FC-CBJ vs. FC-CBJ+RedNG: Comparing number of nodes visited for $\langle n = 40, a = 10, d, t \rangle$.





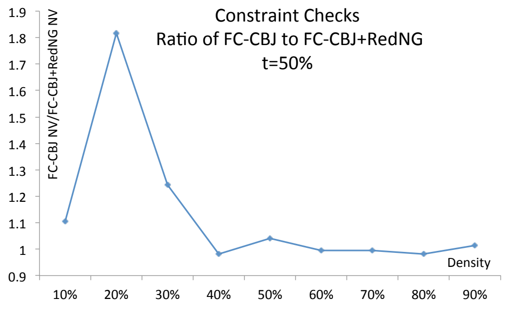
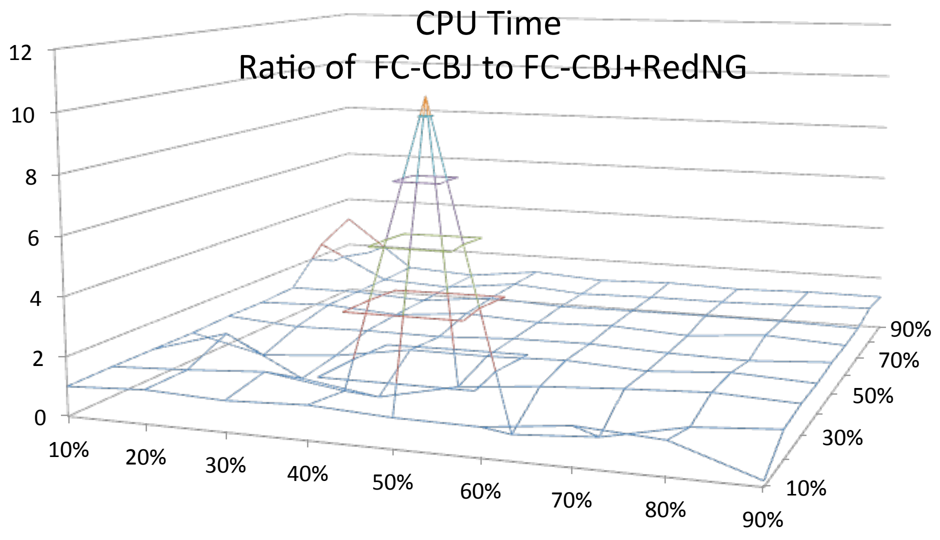
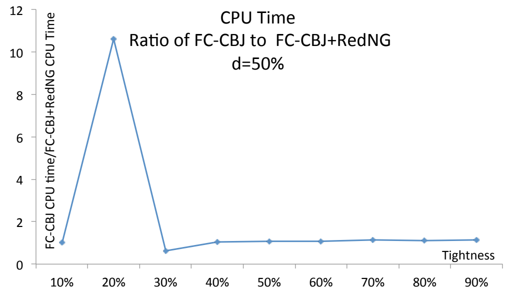


Figure 12: FC-CBJ vs. FC-CBJ+RedNG: Comparing number of constraint checks for $\langle n = 40, a = 10, d, t\rangle$.





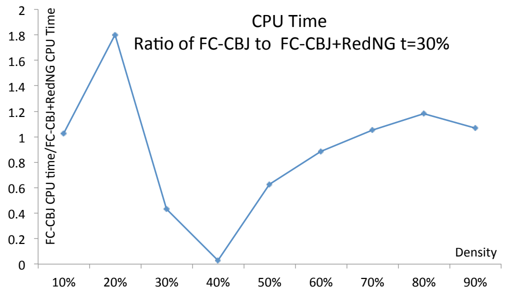
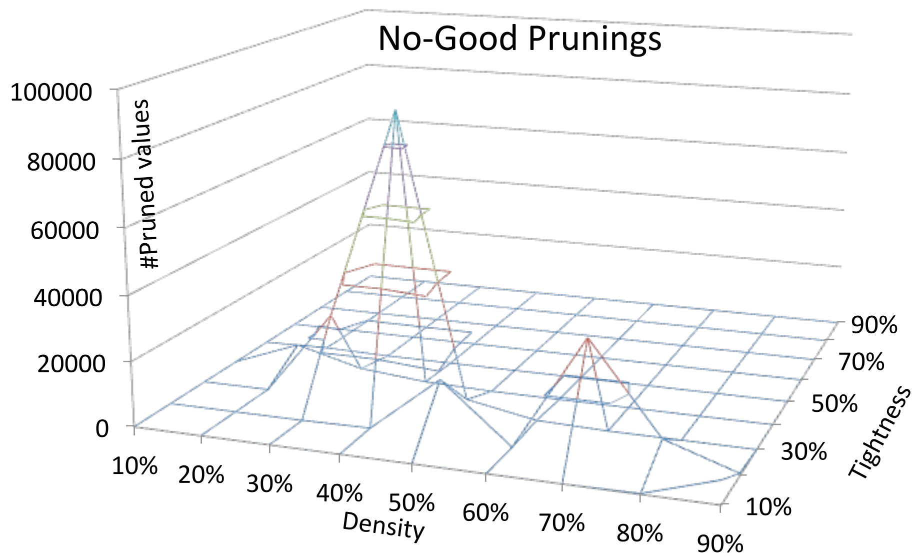
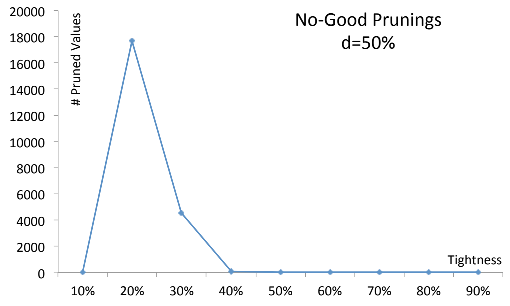


Figure 13: FC-CBJ vs. FC-CBJ+RedNG: Comparing number of CPU time(ms) for $\langle n = 40, a = 10, d, t\rangle$.





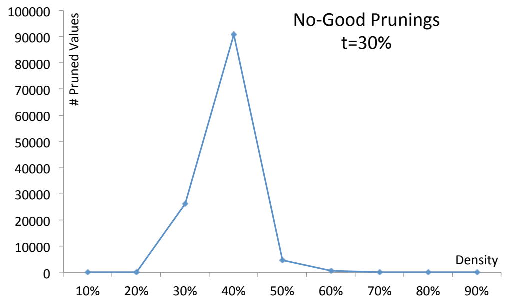


Figure 14: FC-CBJ+RedNG: Pruning achieved learned no-goods for $\langle n = 40, a = 10, d, t\rangle$.

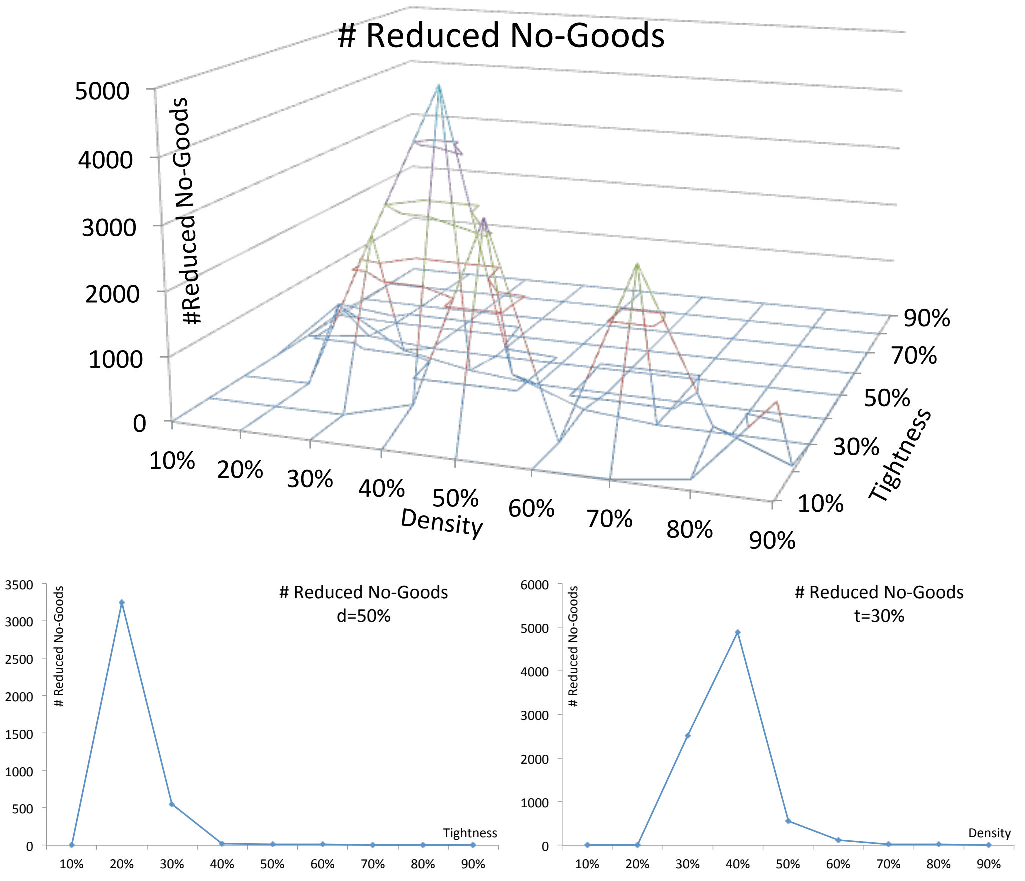


Figure 15: FC-CBJ+RedNG: Number of reduced no-goods for $\langle n = 40, a = 10, d, t\rangle$.

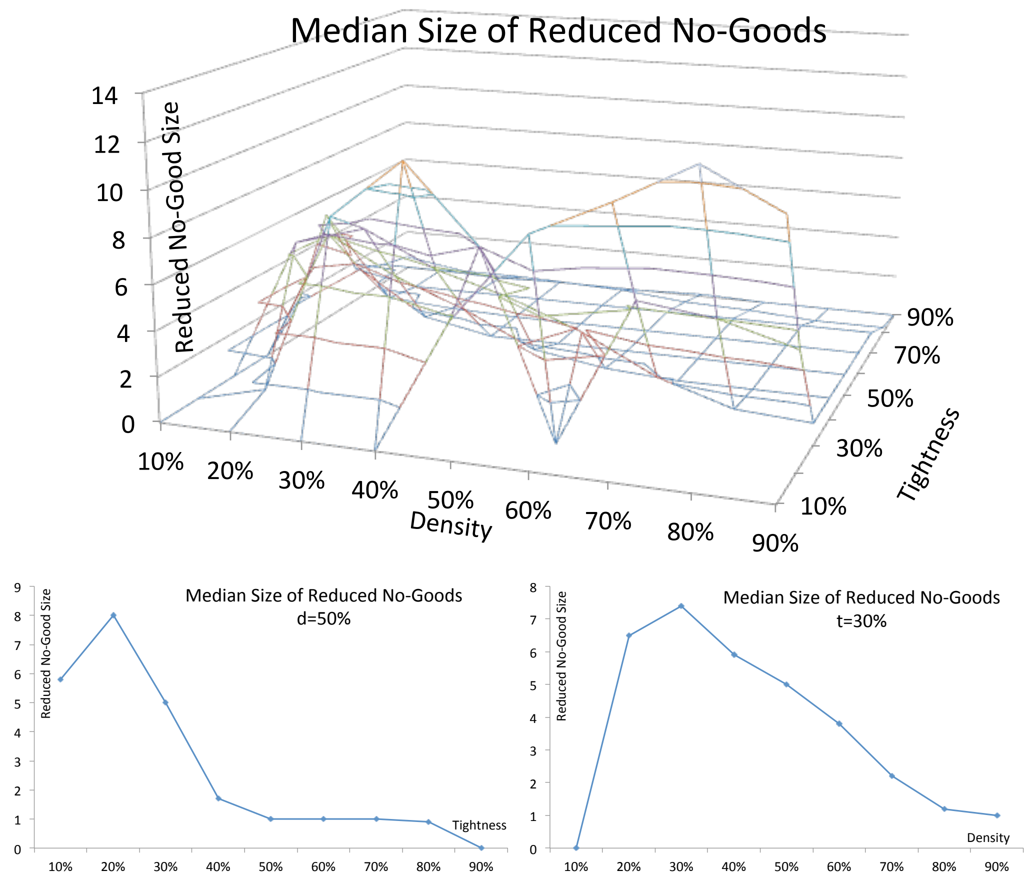


Figure 16: FC-CBJ+RedNG: Median Size of reduced no-goods for $\langle n = 40, a = 10, d, t\rangle$.

9.4 Benchmarks Instances

We compared the performance of FC-CBJ+REDNG to FC-CBJ to with 77 benchmark problems. We report the results of 13 of those benchmark instances in Tables 1 and 2. We measured total number of no-goods learned and the maximum size of no-good that was learned during search in addition to the same data that was recorded for the instances in Section 9.2.

Similar to the random instances, we see the most consistent savings in number of constraint checks and nodes visited. In the best case FC-CBJ+REDNG outperforms FC-CBJ by five orders of magnitude for both constraint checks and nodes visited. However, in terms of CPU time FC-CBJ+REDNG is not consistently better than FC-CBJ. The lack of consistent savings on CPU time could be due to the overhead associated with processing a large amount of no-goods with a large amount of variable-value pair. Another factor influencing the CPU time for FC-CBJ+REDNG is the cost of reducing the set of no-goods. Both of these operations can become very costly when the no-good set is large. In Table 2, we can compare the sum of the domain sizes to the number of no-good pruning. In most cases the number of pruning is greater than the sum of the domain sizes. Having a large number of redundant prunings implies that the no-goods causes the redundant prunings are not reduced enough and could negatively affect the CPU time.

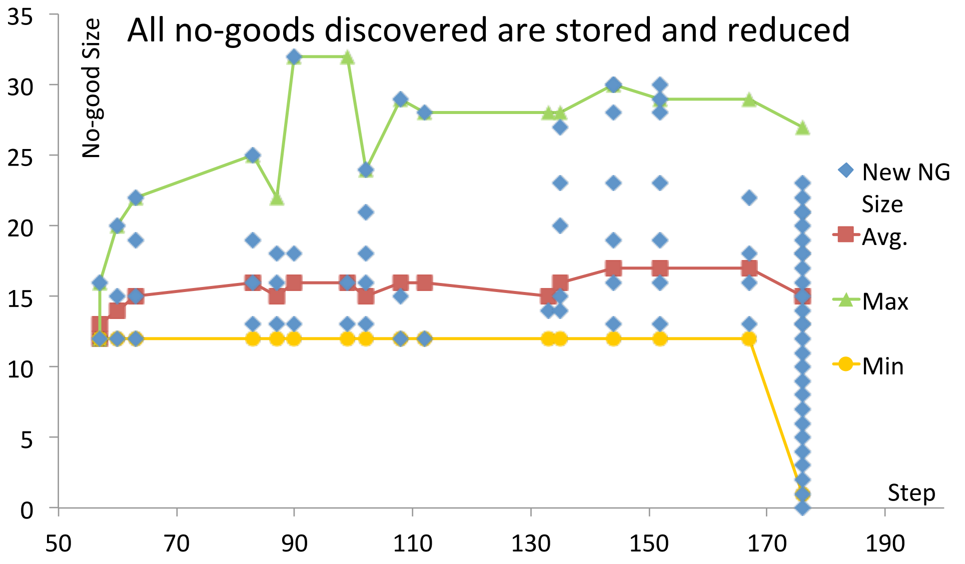
In Table 2 the power of reducing the no-goods is seen by comparing the total number of no-goods to the number of reduced no-goods. In all cases the number of reduced no-goods is less the total number of no-goods. In the worst case and best cases the number of reduced no-goods is smaller than the total number of no-goods by a factor of 0.77 and $9.07E^{-07}$ respectively. To further explore the power of reducing the set of no-goods on two of those benchmark instances: qcp-10-67-13 (unsolveable) and qwh-10-57-0 (solveable). We record at each back track the maximum, minimum, average, and median size of the no-goods. We ran each of these instances on FC-CBJ+REDNG and FC-CBJ+NG, where FC-CBJ+NG records but does not reduce the set of no-goods. Figure 17 depicts the unsolvable instance; in those graphs we see the most change between FC-CBJ+REDNG and FC-CBJ+NG in the maximum no-good size. Figure 18 depicts the solvable instance. In that solvable instance we see a clearer advantage of reducing the no-goods. The maximum, average, and median no-good are considerably smaller in for FC-CBJ+REDNG. Reducing the no-good set, though initially costly, has obvious benefits because it saves in the long-term when we have to iterate through the no-goods during propagation.

Table 1: FC-CBJ vs. FC-CBJ+RedNG: Comparing constraint check, nodes visited, and CPU time on benchmark problems.

			#CC		#NV		CPU	
Instance Name	**#Vars**	**SAT/UNSAT**	**Basic**	**RedNG**	**Basic**	**RedNG**	**Basic**	**RedNG**
driverlogw-05c	351	1	32,584,587	**232,465**	466,724	**1,894**	750,026	**9,804**
driverlogw-02c	301	1	8,151,150	**685,901**	175,065	**11,709**	114,752	**56,542**
qcp-10-67-13	100	0	858,789,330	**6,631**	50,625,326	**176**	4,860,911	**998**
qcp-10-67-5	100	1	35,125	**19,128**	2,317	**1,123**	**2,939**	6,469
qcp-15-120-4	225	1		476,654		18,683		**206,576**
qcp-15-120-1	225	1	183,757,769	**461,293**	9,266,828	**20,068**	**2,941,590**	**212,431**
qwh-10-57-0	100	1	27,206	**15,871**	1,695	**884**	**3,194**	4,956
qwh-15-106-5	225	1	2,068,713	**693,012**	105,511	**31,890**	**35,683**	607,379
qwh-15-106-6	225	1	120,369	**27,395**	6,908	**783**	6,038	**4,674**
rand-23-23-253-131-46021	23	0	1,107,161,309	**904,069,150**	22,967,968	**18,319,518**	**4,313,958**	5,936,376
rand-23-23-253-131-55021	23	0	1,084,019,218	**875,330,454**	22,618,726	**17,826,276**	4,606,702	**4,095,486**
rand-24-24-276-139-48021	24	1	443,457,359	**379,472,842**	8,926,163	**7,469,836**	2,273,366	**2,151,662**
rand-24-24-276-139-49021	24	0	2,528,012,848	**2,199,184,173**	50,183,470	**42,710,983**	**10,194,928**	13,742,483

Table 2: FC-CBJ+RedNG: Domain filtering by no-goods, number and size of no-goods on benchmark problems.

Instance Name	Sum dom sizes	NG-Pruning	Total#NG	#RedNG	Max NG Size	Max RedNG Size	Min RedNG Size	Median RedNG Size	Avg RedNG Size
driverlogw-05c	1,345	159	483	170	22	21	2	6	6
driverlogw-02c	1,161	2,219	4,336	1,562	51	45	1	20	20
qcp-10-67-13	703	9	83	38	32	27	1	15	15
qcp-10-67-5	703	197	703	385	65	64	12	39	38
qcp-15-120-4	1,905	9,485	12,292	3,121		157	19	80	80
qcp-15-120-1	1,905	8,569	13,205	3,683		162	19	99	91
qwh-10-57-0	613	76	487	299	60	55	12	35	34
qwh-15-106-5	1,709	16,227	21,196	5,626	171	165	19	106	95
qwh-15-106-6	1,709	58	313	242	146	139	19	62	65
rand-23-23-253-131-46021	529	222,997	11,386,887	23	15	1	1	1	1
rand-23-23-253-131-55021	529	222,973	11,078,189	23	15	1	1	1	1
rand-24-24-276-139-48021	576	108,482	4,666,734	33	15	10	1	5	5
rand-24-24-276-139-49021	576	598,379	26,531,815	24	16	1	1	1	1



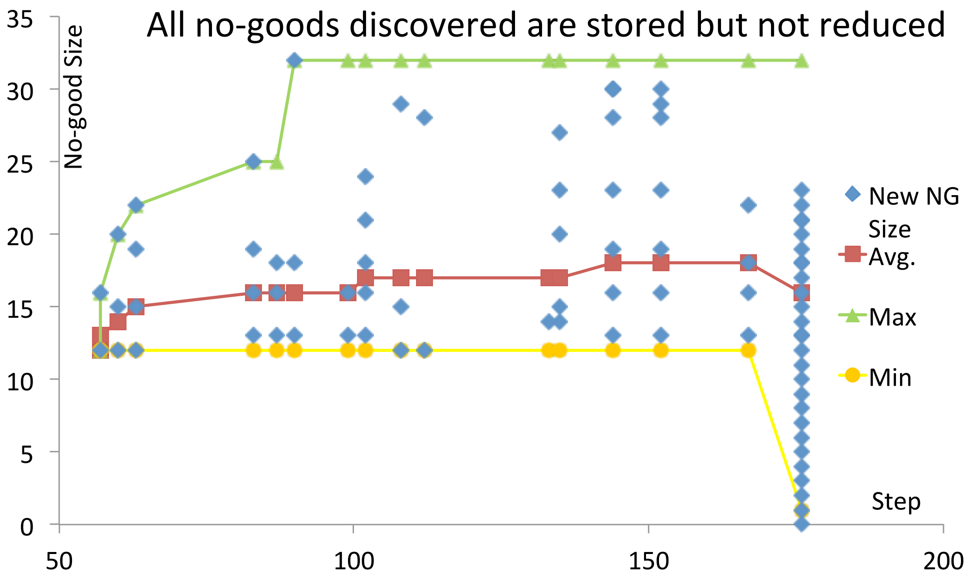
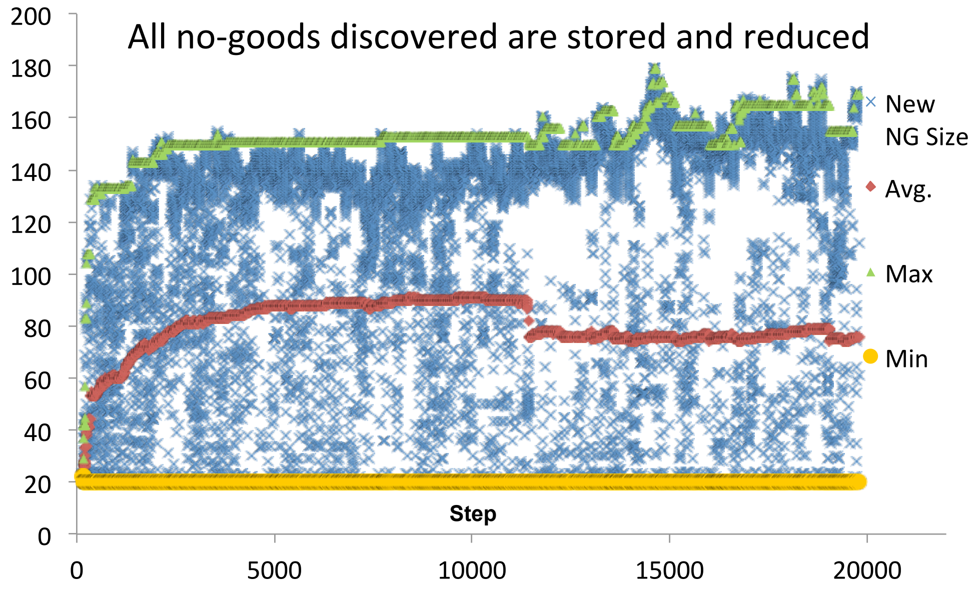


Figure 17: FC-CBJ+RedNG vs. FC-CBJ+AllNG: Comparing the sizes of no-goods learned between FC-CBJ+RedNG and FC-CBJ+AllNG for unsolvable random instances.

10 Conclusions and Future Research

The addition of no-good learning and no-good reduction to FC-CBJ is beneficial in general. For FC-CBJ+REDNG , we see consistent and significant savings



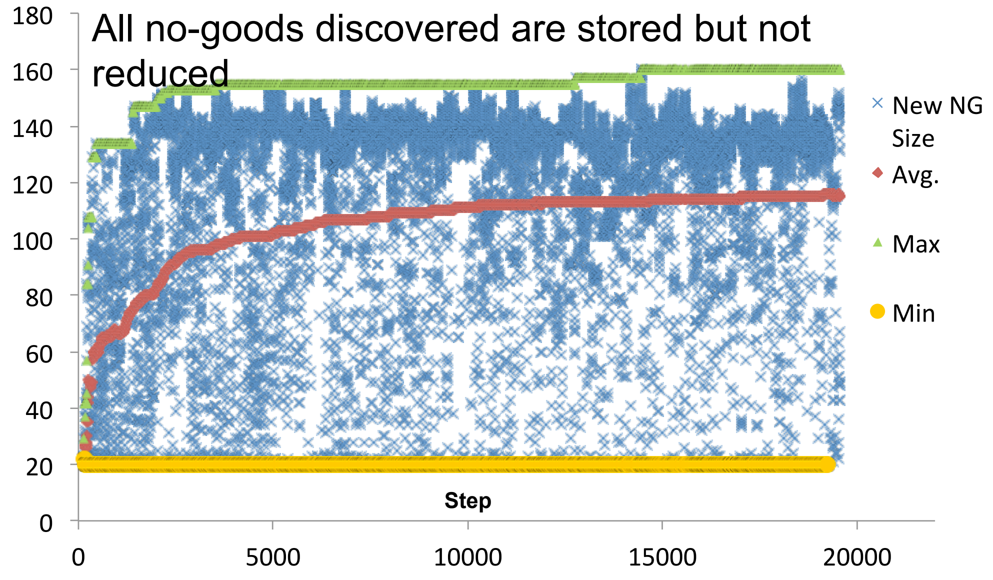


Figure 18: FC-CBJ+RedNG vs. FC-CBJ+AllNG: Comparing the sizes of no-goods learned between for solvable random instances.

in terms of the number of constraint checks and nodes visited. The CPU time, however, has less consistent savings. The lack of consistent savings for CPU time could be due to a number of factors including: overhead of reducing the set of no-goods, the overhead of processing the no-goods, or inefficient implementation. If the no-good set consists of a large number of no-goods with several variable-value pair when FC-CBJ+REDNG begins learning no-goods that are subsets of the existing no-goods, it can be very costly to reduce the no-goods set. Also, with a large set of no-goods, each no-good consisting of several variable-value pairs, the overhead for processing the no-goods will increase. Finally, a more efficient implementation of the key functions for reducing the no-good set and propagating the no-goods could improve the performance of FC-CBJ+REDNG in terms of CPU time.

Our approach for learning and propagating no-goods as well as reducing the set of no-goods provides an opening for interesting future research. Instead of adding the learnt no-goods to the problem, our approach could be adjusted to alter the original problem constraints. If the scope of a no-good is the same as the scope of an original problem constraint, that original constraint can be filtered or added to depending on if the constraint is a support or conflict. However, if the new no-good's scope does not match any original problem constraint, it still must be added to the list of no-goods. Analysis on the scopes of the no-goods in the no-good list could result in combining no-goods with the same scope. The combination of altering original problem constraints and combining no-goods could result further savings during search. Both our original approach of adding no-goods and the approach of altering the original problem constraints and no-goods list could be extended to non-binary CSPs. The latter approach could be particularly effective in non-binary CSPs because the scopes of the original problem constraints are more likely to have the same scope as a learned no-good.

Acknowledgments

I am very grateful to my thesis advisor, Dr. Berthe Y. Choueiry, for her continuous support, encouragement, and mentoring. Her feedback and ideas have been invaluable to me during the full course of my research. I would also like to thank Dr. Berthe Y. Choueiry, Dr. Matthew Dwyer, Mr. Robert J. Woodward, and Dr. Elena Sherman their support and mentoring during a year-long group research study of CSPs and SAT that inspired the topic of my thesis. Finally, I am very appreciative of the financial support provided by CREU-W of the CRA, Undergraduate Creative Activities and Research Experiences Program of the University of Nebraska-Lincoln, and National Science Foundation REU supplements for Grant #CNS-0720654 and Grant #RI-111795.

A Pseudocode

Algorithm 11: FC-CBJ-UNLABEL(i, $consistent$).

```
Input: $i$ index of current variable, $consistent$ indicates the status of the
       partial solution
Output: $h$ the index of the variable backtrack to/instantiate
1  $V_h \leftarrow$ DEEPESTVAR(UNION($confSet(V_i), pastFC(V_i)$))
2  $confSet(V_h) \leftarrow$ UNION($confSet(V_h), confSet(V_i), pastFC(V_i)$) $\setminus \{V_h\}$
3  for $j \leftarrow i$ DOWNTO $h + 1$ do
4      $confSet(v_j) \leftarrow \{0\}$
5      UNDOREDUCTIONS($V_j$)
6      UPDATECURRDOM($V_j$)
7  UNDOREDUCTIONS($V_h$)
8  $D^{curr}[V_h] \leftarrow D^{curr}[h] \setminus \{val[V_h]\}$
9  $consistent \leftarrow D^{curr}[h] \neq nil$
10 return $h$
```

The function FC-CBJ-UNLABEL. UNDOREDUCTIONS and UPDATECURRDOM are taken from [**?**].