Value Ordering for Finding All Solutions: Interactions with Adaptive Variable Ordering

Deepak Mehta, Barry O'Sullivan, and Luis Quesada

Cork Constraint Computation Centre, University College Cork, Ireland
{d.mehta|b.osullivan|l.quesada}@4c.ucc.ie

Abstract. We consider the impact of value ordering heuristics on the search effort required to find all solutions, or proving none exist, to a constraint satisfaction problem in k-way branching search. We show that when the variable ordering heuristic is adaptive, the order in which the values are assigned to variables can make a significant difference in all measures of search effort. We study in depth an open issue regarding the relative merit of traditional value heuristics, and their complements, when searching for all solutions. We also introduce a lazy version of k-way branching and study the effect of value orderings on finding all solutions when it is used. This paper motivates a new and fruitful line of research in the study of *value ordering* heuristics for proving unsatisfiability.

1 Introduction

The entire search space of a constraint satisfaction problem (CSP) is explored when one is interested in finding all solutions, counting the number of solutions, or proving that the problem is unsatisfiable. The latter case may also appear as a sub-problem while solving a satisfiable problem when a globally inconsistent assignment to a subset of the variables is being explored. While most research in the area of search heuristics has focused on variable ordering, the question of determining which value should be assigned to the current variable has not received much attention. One reason is that it has been generally accepted that when the entire search space of a CSP is explored by a search algorithm that backtracks chronologically, the order in which the values are selected does not affect its search effort, e.g. the number of visited nodes, or the number of failures [4]. A well-known theorem states that this is true for both static and dynamic variable ordering heuristics [4]. In the case of search algorithms that perform *binary branching* recent work has shown that search effort is affected by the choice of value ordering [11]. However, that work was supportive of the conventional wisdom in the case of k-way branching.

In the case of search algorithms that perform *k-way branching* we advance the conventional wisdom related to the role of value ordering heuristics in the context of CSPs. We show that the conventional wisdom only applies when non-adaptive static/dynamic variable ordering heuristics are used. However, when adaptive dynamic variable ordering heuristics are used, value ordering heuristics can make a *significant* difference in the search-effort of a chronological backtrack search algorithm, including the MAC [9] algorithm, even when the entire search space is explored using k-way branching. Furthermore, we show that static value ordering heuristics can make a difference in terms of the number of support checks for algorithms that maintain arc consistency during search even for k-way branching, even when there is no difference in the search effort. A preliminary investigation of how value ordering can affect the search to find all solutions is reported in [8]. Here, we present a more extensive investigation and explain how value ordering can affect search effort when using k-way branching.

We also introduce a *lazy version of k-way branching* whereby instead of selecting and assigning a value to a variable, a value is selected and *removed* from the domain of the selected variable. We show that postponing the assignment decision can help in reducing the number of failures. We further perform a detailed analysis and also demonstrate the effect of value ordering heuristics on the search effort when using lazy k-way branching. Finally, we perform a detailed study of value ordering heuristics and their corresponding anti-heuristics with respect to their relative efficiency. We demonstrate that one can dramatically out-perform the other depending on the context. A major contribution of this paper is that it motivates a new and fruitful line of research in the study of *value ordering* heuristics for proving unsatisfiability.

Although binary branching is theoretically more efficient than k-way branching [6], in practice the latter can be more efficient than the former on many classes of problem [1]. Our goal is not to compare the relative merits of different branching schemes but to show that value orderings can have a significant impact under all branching schemes. We show the various ways in which value ordering heuristics affect the various elements of search which contribute to an overall effect. The results presented in this paper complement and complete the analysis of the effects of value ordering on branching strategies introduced by Smith and Sturdy in [11] on binary branching.

2 Background

A CSP, $\mathcal{P}$, is a triple $(\mathcal{X}, \mathcal{C}, D)$ where $\mathcal{X}$ is a set of variables and $\mathcal{C}$ is a set of constraints. Each variable $X \in \mathcal{X}$ is associated with a finite domain, which is denoted by $D(X)$. We use n, d and e to denote the number of variables, the maximum domain size, and the number of constraints respectively. Each constraint is associated with a set of variables on which the constraint is defined. We restrict our attention to binary CSPs, where the constraints involve two variables. A binary constraint C_{XY} between variables X and Y is a subset of the Cartesian product of $D(X)$ and $D(Y)$ that specifies the allowed pairs of values for X and Y. We assume that there is only one constraint between a pair of variables. A value $b \in D(Y)$ is called a support for $a \in D(X)$ if $(a, b) \in C_{XY}$. Similarly $a \in D(X)$ is called a support for $b \in D(Y)$ if $(a, b) \in C_{XY}$.

A value $a \in D(X)$ is called arc-consistent (AC) if for every variable Y constraining X the value a is supported by some value in $D(Y)$. A CSP is AC if for every variable $X \in \mathcal{X}$, each value $a \in D(X)$ is AC. We use AC($\mathcal{P}$) to denote the CSP obtained after applying arc consistency. If there exists a variable with an empty domain in $\mathcal{P}$ then $\mathcal{P}$ is unsatisfiable and it is denoted by $\mathcal{P} = \bot$. Maintaining Arc Consistency (MAC) after each decision during search is one of the most efficient and generic approaches to solving CSPs. A *solution* of a CSP is an assignment of values to all the variables that satisfies all the constraints. A CSP is *satisfiable* if and only if it admits at least one solution; otherwise it is *unsatisfiable*. In general, determining the satisfiability of a CSP is NP-complete.

Branching Strategies. A branching strategy defines a search tree. The well-known branching schemes are k-way branching, binary branching [10] and split branching. In k-way, when a variable X with k values is selected for instantiation, k branches are formed. Here each branch corresponds to an assignment of a value to the selected variable. In binary branching, when a variable X is selected, its values are assigned via a sequence of binary choices. If the values are assigned in the order $v_1, v_2, \ldots, v_k$, then two branches are formed for the value v_1, associated with $X = v_1$ and $X \neq v_1$ respectively. The left branch corresponds to a positive decision and the right branch corresponds to a negative decision. The first choice creates the left branch; if that branch fails, or if all solutions are required, the search backtracks to the choice point, and the right branch is followed instead. Crucially, the constraint $X \neq v_1$ is propagated, before selecting another variable-value pair. In split branching, when a variable X is selected, its domain is divided in to two sets: $\{v_1, \ldots, v_j\}$ and $\{v_{j+1}, \ldots, v_k\}$, where $j = \lceil k/2 \rceil$. Two branches are formed by removing each set of values from $D(X)$ respectively.

Variable Ordering Heuristics. When a dynamic variable ordering is used the selection of the next variable to be considered at any point during search depends on the current node of the search tree. Examples of dynamic variable ordering heuristics are: dom/deg [2] and dom/wdeg [3]. The dom/deg heuristic selects a variable which has the smallest ratio of the current domain size to the original degree of the variable, while the *dom/wdeg* heuristic selects a variable which has the smallest ratio of the current domain size to the weighted degree of the variable. The dom/wdeg heuristic is adaptive while the dom/deg is non-adaptive. By adaptive we mean that the heuristic measure of a variable at a given node of the search tree is dependent on previous experience with the search process. For a non-adaptive variable ordering, the heuristic measure at a particular node in the search tree is same before and after backtracking to the node. However, this is not necessarily true for an adpative variable ordering heuristic.

Value Ordering Heuristics. A value ordering heuristic is used to select a value from the domain of a variable to instantiate that variable during search. Three value ordering heuristics are *total-cost*, *cruciality*, and *promise*, which are primarily based on selecting the least constrained value for a variable and are proposed in [5]. The heuristic *total-cost* associates each value from the domain of a variable with the sum of incompatible values in the domains of the other variables. The values are then considered in the increasing order of this count. The heuristic *cruciality* differs slightly from *total-cost*. It aggregates the percentage of the incompatible values in future domains. The heuristic *promise* associates each value with the product of the number of compatible values in the domain of each variable. The value with the highest product is chosen subsequently. For all these heuristics, the compatibility of each value a in the domain of a variable x is tested with each value b in the domain of each variable y constrained with x. This process requires $\mathcal{O}(n\,d^2)$ support checks in the worst-case after each variable selection during search. Several value ordering heuristics including *min-conflict* are presented in [4], which is, in fact, the same as the total-cost of [5].

3 Impact of Value Orderings on MAC

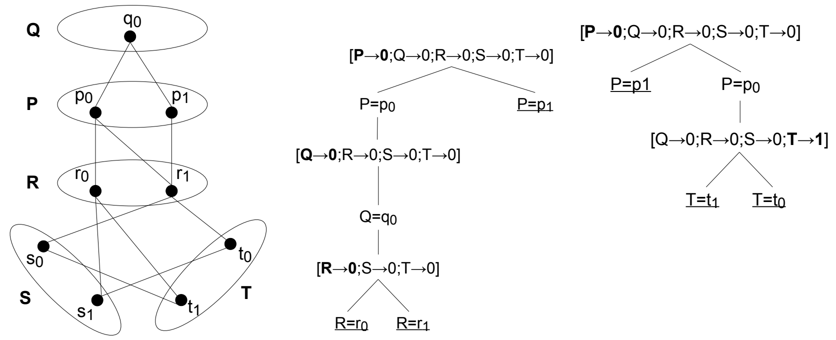
In this section we show that value ordering heuristics can affect the search effort (i.e. the number of visited nodes, failures etc.) of a backtrack search algorithm that forms k-way branching when the entire search space of a CSP is explored.

3.1 State-of-the-Art on Heuristic Interactions

Frost and Dechter [4] claim that value orderings have no impact on the search effort of a backtrack search algorithm, when all solutions of a CSP are searched, which includes when no solution exists. This claim has been made both for static and dynamic variable ordering heuristics [4]. Their argument is that when a variable X with k values is selected, k subtrees are explored *independently*, and the search spaces of these k subtrees are commutative. To find all solutions or to prove that there are none, every subtree must be explored and therefore the order in which values are assigned cannot make a difference in cumulative search effort.

Smith and Sturdy [11] claim that value orderings can make a difference in the algorithm's search effort when *binary-branching* is used, and agree that value ordering does not make a difference when *k-way branching* is used. Their argument is that in binary-branching, if a variable X and a value $x_1 \in D(X)$ is selected then two subtrees are created, $X = x_1$ and $X \neq x_1$. If $X = x_1$ fails, then the constraint $X \neq x_1$ is propagated, which can lead to further domain reduction. This propagation can remove one or more values from the current variable's domain which are not yet considered for instantiation. Hence, the order in which the values are assigned can affect the search effort even if the entire search space is explored.

In the following section we show that k subtrees are not always explored independently in k-way branching. It depends on whether a dynamic variable ordering heuristic is adaptive or non-adaptive. Notice that the heuristic measures of variables for a non-adaptive heuristic like *dom/deg* and an adaptive heuristic like *dom/wdeg* are changing during search. When the algorithm backtracks to a node the heuristic measures of all variables of a non-adaptive heuristic like dom/deg is restored to the same measures as they were before exploring that node. However, this does not hold for a dynamic variable ordering heuristic like



(a) An unsatisfiable CSP. (b) LEX (c) RLEX

Fig. 1. An unsatisfiable CSP with five variables (Figure 1(a)), and search trees with different value orderings (Figure 1(b) and Figure 1(c)).

dom/wdeg, which is adaptive. Consequently, the search spaces of k subtrees may not necessarily be commutative. Therefore, when an adaptive dynamic variable ordering is used for searching all solutions, k-way branching is sensitive to the choice of value ordering heuristic.

3.2 The Role of Value Ordering: New Insights

Let us consider a trivial CSP whose micro-structure is depicted in Figure 1(a). There are five variables P, Q, R, S and T. Their domains are $D(P) = \{p_0, p_1\}$, $D(Q) = \{q_0\}$, $D(R) = \{r_0, r_1\}$, $D(S) = \{s_0, s_1\}$ and $D(T) = \{t_0, t_1\}$. There are five binary constraints. Here an edge corresponds to a pair of values that satisfy the constraint. For any two values a_i and a_j, we write $a_i <_l a_j$ if a_i is lexicographically smaller than a_j. Notice that the CSP is inconsistent. The reason for the inconsistency is the sub-problem restricted to the variables R, S and T.

The search trees shown in Figures 1(b) and 1(c) are the results of applying the MAC algorithm. The variable ordering heuristic employed by the search algorithm *selects a variable having the maximum number of wipeouts with a lexicographical tie breaker*. Initially, the number of wipeouts associated with each variable is set to 0. The number of wipeouts, v_x, associated with a variable x is written as $x \rightarrow v_x$. In the search trees, uninstantiated variables are enclosed in the square brackets in the lexicographical order, e.g. $[x \rightarrow v_x, \ldots, z \rightarrow v_z]$. The selected variable is indicated by making it bold. Each assignment of a value to the selected variable represents a node visited in the search tree. When a node is underlined, it indicates a failure.

The search tree in Figure 1(b) is the result of using the lexicographical value ordering heuristic (LEX). Initially, the number of wipeouts associated with each variable is 0, so the lexicographically smallest variable P is selected and it is instantiated to the lexicographically smallest value p_0. Enforcing AC at this point does not remove any value from any domain. The next lexicographically smallest variable is Q, which is then initialized to q_0. Again, enforcing AC makes no change in the domains. When the variable R is selected, each of its instantiations leads to a domain wipeout. When R is instantiated to r_0, first the domain of S is revised against the domain of R and s_0 is removed. Next, the domain of T is revised against the domains of R and S. This results in removing t_0 and t_1 and hence the domain wipeout occurs. When R is initialized to r_1, there is again a domain wipeout associated with T. The search process backtracks to P and initializes it to p_1. This again results in the domain wipeout associated with T, since r_0 is removed from R while revising its domain against the domain of P, which eventually results in the domain wipeout associated with T.

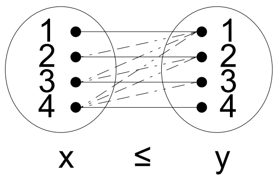
The search tree shown in Figure 1(c) is the result of using the reverse lexicographical value ordering heuristic (RLEX). First P is initialized to p_1 which results in a domain wipeout. This happens while revising the domain of T. At this point the number of wipeouts associated with T is incremented by 1. This influences the selection of the variable T after initializing P to p_0 in Figure 1(c). However, in Figure 1(b), due to a different value ordering, when P is initialized to p_0, Q is selected instead of T, which results in a different number of nodes.

When LEX is used the number of search nodes is 5 and when RLEX is used the number of nodes is 4. This difference is due to the interaction between the variable ordering and the value ordering heuristics. When an assignment fails the number of wipeouts associated with a variable changes. Different value ordering heuristics may change the number of wipeouts associated with different variables. Consequently, the ordering of the values in the previously explored subtrees may influence the decision of selecting the next variable in the subtrees that are yet to be explored. This shows that claims made in [4] and [11] that a search algorithm that forms k-way branching explores k subtrees independently is not always true. Hence, value orderings can affect the search tree of a backtrack algorithm when adaptive dynamic variable ordering heuristics are used with k-way branching for exploring the entire search space.

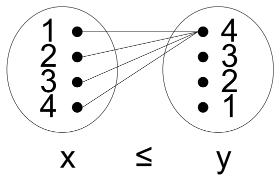
4 Impact of Value Orderings on AC

We show that static value ordering heuristics can have an impact on the efficiency of arc consistency algorithms. We focus on the static versions of the heuristics *total-cost*, *cruciality* and *promise*. The ordering based on these heuristics can be viewed as arranging the values in increasing order of their constrainedness. This can be advantageous while revising the domains of the variables, when trying to make the problem AC. More specifically, putting the least constrained value at the beginning of the domain list might help values of other domains to find their support more quickly during revision (on average). This may save failed support checks since the further the first support is from the start of the domain list, the more are the failed checks required to find that support for a given value.

To illustrate this, let us consider a constraint $X \leq Y$ and study the revision of $D(X)$ against $D(Y)$ as shown in Figures 2(a) and 2(b). If the values in $D(Y)$



(a) Increasing order.



(b) Decreasing order.

Fig. 2. Visualization of the checks of the constraint $X \leq Y$ where dashed lines refer to failed checks and solid lines refer to successful checks.

are arranged in $\langle 1, 2, 3, 4\rangle$ order, as shown in Figure 2(a), and the search for a support starts from 1, then the revision of $D(X)$ against $D(Y)$ will require 10 support checks in total, using the revise function of AC-3. If the values in $D(Y)$ are arranged in $\langle 4, 3, 2, 1\rangle$ order, as shown in Figure 2(b), then the revision of $D(X)$ against $D(Y)$ will require only 4 support checks in total, using the revise function of AC-3. Obviously, different constraints may require different orderings of values. However, these orderings can be aggregated. For example, one can use total-cost or promise as measures to sort the values of the domain accordingly.

The fact that ordering the values can reduce support checks during revisions and thereby improves the average revision time does not seem to have been observed before. This is the reason that when a static value ordering heuristic is used in a search algorithm such as MAC, it can make a difference at least in terms of support checks even when the entire search space is explored.

5 Lazy K-way Branching

In k-way branching a decision corresponds to an assignment of a value to a given variable and its dual can be seen as removing all but one value from the domain of the variable. An example of k-way branching is illustrated in Figure 3, where a box denotes a variable selection and an ellipse denotes selecting and assigning a value to the selected variable. Here X is the selected variable whose domain is $\{a_1, a_2, a_3, a_4, a_5\}$ and $k = 5$. A (positive) decision $X = a_1$, is equivalent to removing a_2, a_3, a_4 and a_5 from the domain of X, or in other words enforcing a conjunction of inequalities, i.e., $X \neq a_2 \wedge X \neq a_3 \wedge X \neq a_4 \wedge X \neq a_5$.

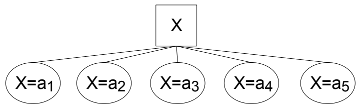
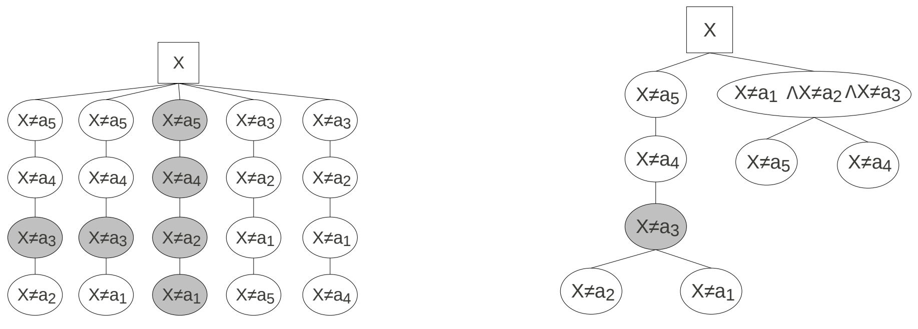


Fig. 3. k-way branching.

We propose a *lazy version* of the *k-way branching* scheme whereby instead of enforcing all $(k-1)$ inequalities at once each inequality is enforced separately, e.g., each assignment of a value in Figure 3 corresponds to a sequence of inequalities in Figure 4(a). The k-way branching scheme can be seen as being optimistic whereby, based on some heuristic measure, a most optimistic value is selected and assigned to the selected variable. Lazy k-way branching can be seen as being pessimistic whereby a most pessimistic value is selected and removed from the selected variable. Postponing the instantiation of a variable may help in making better decisions, thus reducing the number of mistakes.



(a) Without Sharing Nodes. (b) Sharing Nodes.

Fig. 4. Lazy k-way branching.

An additional advantage of enforcing each inequality separately is that one can infer dependencies between *explicitly* removed values of the selected variable as a result of making (negative) decisions, and the *implicitly* removed values of the selected variable as a result of enforcing local consistency, such as arc consistency in the case of MAC. These dependencies can be exploited to reduce the number of decisions. For example, let us assume that a_3 is removed from $D(X)$ when arc consistency is enforced after taking the negative decisions $X \neq a_5$ and $X \neq a_4$ in the first branch of Figure 4(a), which is shown by shading the decision node $X \neq a_3$. One can infer the following implication: $X \neq a_5 \wedge X \neq a_4 \rightarrow X \neq a_3$. This effectively means that there does not exist any solution in the resulting subproblem after selecting variable X where $X = a_3$. Therefore, if the decision of instantiating X to a_3 has not yet been tried, then there is no need to try it. Hence, the third branch of Figure 4(a) is not explored, which is equivalent to $X = a_3$. This is shown by shading all the corresponding negative decisions: $X \neq a_5$, $X \neq a_4$, $X \neq a_2$, and $X \neq a_1$. This is an original and novel way of reducing the number of useless branches and failures.

In k-way branching, if k branches are explored after selecting a variable, then each value is removed at most $(k-1)$ times, and in algorithms like MAC, the

Algorithm 1 $\text{MAC}_{LK}(\mathcal{P}, Y)$

```
Require: $\mathcal{P}$ : input CSP $(\mathcal{X}, \mathcal{C}, D)$; $Y$: current variable
 1: if $\mathcal{X} = \emptyset$ then
 2:     solution found
 3: else
 4:     if $Y =$ null then select and remove any variable $X$ from $\mathcal{X}$
 5:     else $X \leftarrow Y$
 6:     $V \leftarrow \emptyset$; $D' \leftarrow D$
 7:     while $\mathcal{P} \neq \perp \wedge |V| < |D(X)|$ do
 8:         select and remove any value $v$ from $D(X)$
 9:         $V \leftarrow V \cup \{v\}$; $\mathcal{P} \leftarrow$ AC$(\mathcal{P})$
10:     if $\mathcal{P} \neq \perp$ then
11:         if $|D(X)| = 1$ then MAC$_{LK}(\mathcal{P}$, null) else MAC$_{LK}(\mathcal{P}, X)$
12:     $D \leftarrow D'$; $D(X) \leftarrow V$; $\mathcal{P} \leftarrow$ AC$(\mathcal{P})$
13:     if $\mathcal{P} \neq \perp$ then
14:         if $|D(X)| = 1$ then MAC$_{LK}(\mathcal{P}$, null) else MAC$_{LK}(\mathcal{P}, X)$
```

work required for propagating the impact of removing a value will be repeated. This repetition can be reduced in lazy k-way branching. Remember that in lazy k-way branching each assignment of a value to a variable can be seen as a path consisting of at most $k-1$ negative decisions starting after the selection of the variable and ending at a node when the variable's domain is singleton. Instead of having disjoint paths for each assignment of a value, as shown in Figure 4(a), the idea is to maximize the sharing of negative decisions among different paths in order to minimize the work required for propagation.

One way of implementing lazy k-way branching in order to share nodes (propagation) is shown in Algorithm 1. Notice that MAC_{LK} requires CSP $\mathcal{P}$ and the current variable Y. If Y is null then a new current variable is selected (Line 4–5). After the current variable, X, is determined, a set V for storing negative decisions is initialized to $\emptyset$, and the domains of the variables are saved in D' (Line 8). While $|V| < |D(X)|$ and there is no domain wipe-out, a value v is selected and removed from $D(X)$, it is added to the set V, and AC is enforced (Line 7–9). When the loop is terminated and if $\mathcal{P}$ is not empty then the left branch is created (Line 10–11). The right branch is created by restoring the domains to D' and setting $D(X)$ to the set of values that were removed in the loop (Line 12–14). If X is instantiated then MAC_{LK} is invoked by setting the current variable to `null`, otherwise it is invoked with the current variable X.

An example of sharing nodes is presented in Figure 4(b). In this example the loop is exited after visiting the node corresponding to $X \neq a_4$ (in the left branch). Notice that the decision associated with the last node and all those decisions preceding it are shared by all branches originating from this node, thus reducing the work required for propagation. When the algorithm backtracks it first removes all those values that are already tried as assignments to the current variable, e.g., $X \neq a_1 \wedge X \neq a_2$, as well as those values of X that were implicitly removed while enforcing arc consistency, e.g., $X \neq a_3$. This is done in Line 12 of

MAC$_{LK}$ when $D(X)$ is set to V, since V contains only a_4 and a_5. Notice that $X = a_3$ is never tried since it was inferred as inconsistent in the subproblem resulting from the selection of variable X.

Similar to the lazy version of k-way branching, lazy versions of binary branching and split branching are also possible. One can infer and exploit the dependencies between inequality constraints involving values of the same domain to reduce the number of failures in a lazy version of any branching scheme. However, this is beyond the scope of the current paper.

6 Experimental Results

The experiments were conducted using MAC as a backtrack search algorithm. AC-3 was used as its arc consistency component which was equipped with the residual support mechanism and revision condition [7]. We conducted experiments with the static versions of *min-conflict*, *max-conflict*, *cruciality*, *anti-cruciality*, *promise* and *anti-promise* value ordering heuristics. The information required for these value ordering heuristics was computed prior to search as a pre-processing step after making the problem initially arc-consistent. We also present the results obtained by using the default ordering of the values as specified by the problem instance which is denoted by *default*. Of course, these heuristics might not be the best or might be very expensive/inapplicable for one or more classes of problems. Nevertheless, the purpose of these experiments is not to prove the efficiency of these value ordering heuristics, but to show that different value ordering heuristics can have a significant impact on the search effort when the entire search space of a CSP is explored using MAC with k-way branching. We also wish to demonstrate the effectiveness of lazy k-way branching when compared with k-way branching.

Search effort was measured in terms of support checks (#c), visited nodes (#n), failures (#f) and the solution time (time) in seconds. All algorithms are written in C. The experiments were carried out as a single thread on Dual Quad Core Xeon CPU, running Linux 2.6.25 x64, with 11.76 GB of RAM, and 2.66 GHz processor speed. We perform experiments on many instances of the problems that were used as benchmarks in the CP solver competition of the CPAI'05 workshop.[1]

Table 1 presents results for different value ordering heuristics when the dom/deg variable ordering is used to explore the complete search space. The search nodes for all value orderings is the same as depicted in the first column, which is consistent with the conventional wisdom. However, there is a difference in terms of support checks. On average fewer checks are required when values are ordered based on a heuristic than with the corresponding anti-heuristic. The difference in terms of checks is not significant. The reason is that a huge number of support checks are replaced and reduced by auxiliary checks performed by the residual support mechanism and revision condition. For example, for the queens-knights instance qk_12_12_5_mul , 2% of the checks are saved when values are

[1] `http://cpai.ucc.ie/05/Benchmarks.html`

Table 1. Results for exploring the entire search space with dom/deg and k-way branching. Instances are: (a) bqwh $- 15 - 106 - 32$ (#n = 33168), (b) frb50 $- 23 - 3 -$ *bis* (#n = 230746522), (c) graph12_$w1$ (#n=177059), (d) dual_ehi $- 85 - 297 - 10$ (#n 377649), (e) qk_12_12_5_mul (#n 1996472).

inst		default	min-conflict	max-conflict	promise	anti-promise
(a)	#c	1,174,152	1,181,071	1,176,787	1,178,857	1,179,353
	time	0.535	0.534	0.528	0.537	0.533
(b)	#c	98,657,596,677	93,687,374,052	103,998,537,736	93,537,311,214	104,177,841,288
	time	19,924.439	19,761.988	20,168.810	19,747.415	19,865.577
(c)	#c	27,573,288	28,114,893	28,012,342	28,114,952	28,012,414
	time	2.565	2.695	2.610	2.624	2.619
(d)	#c	374,657,413	363,435,461	382,601,651	362,105,766	382,003,776
	time	400.085	400.772	400.746	396.993	399.856
(e)	#c	6,283,619,236	6,148,687,117	6,330,315,375	6,148,556,738	6,317,352,246
	time	127.459	124.576	127.909	125.882	126.916

ordered by min-conflict when compared with that of max-conflict. However, if a standard AC-3 is used then 14% of the checks are saved by min-conflict when compared with max-conflict.

Table 2 presents results for different problem classes when the *dom/wdeg* variable ordering is used with different value ordering heuristics and the complete search space is explored using k-way branching. We have computed the ratio with respect to the default for each heuristic different to the default. For each instance, the highest/lowest result is written in bold/italic. The results clearly show that value ordering heuristics can make a significant difference in terms of the number of search nodes and time. It is worth emphasizing that for some problems a heuristic like min-conflict, cruciality, or promise performs better while on some others, its corresponding anti-heuristic performs better. We did some further investigation by solving the same instance with 2500 random value orderings. The results for some of them are presented in Figure 5; in these plots each point represents the probability of the search effort to solve an instance exceeding the corresponding number of search nodes on the x-axis. It again shows that value ordering heuristics can make a difference up to several orders-of-magnitude in terms of search nodes when the entire search space is explored; the first two graphs, in fact, exhibit a heavy-tail distribution in search effort.

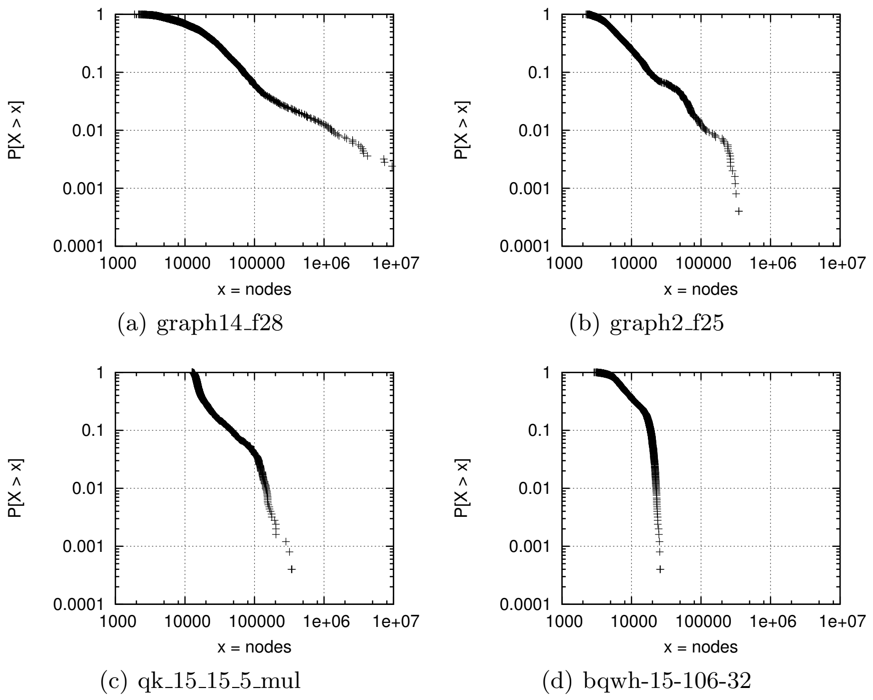
Table 3 presents results for lazy k-way branching and k-way branching when the entire search space was explored with different value ordering heuristics. In the first three rows *dom/deg* was used while for the remaining *dom/wdeg* was used. The first observation is that when different value orderings are used in conjunction with lazy k-way branching they can result in different search effort when the entire search space is explored. More importantly, unlike k-way branching, the difference in the search effort is also observed for dom/deg as shown in 2^{nd} and 3^{rd} rows. This is because in the lazy k-way branching scheme a decision corresponds to selecting and removing a value, with AC being enforced after each value removal. Therefore, depending on the order in which the values are removed dependencies involving inequalities amongst the values of the same domain are inferred, which are exploited to reduce the number of decisions. Moreover, when dom/wdeg is used, the difference in the search effort is also due

Table 2. Results for exploring the entire search space with dom/wdeg and k-way branching.

instance		**default**	**min-conflict**	**max-conflict**	**cruciality**	**anti-cruciality**	**promise**	**anti-promise**
bqwh − 15 − 106 − 32	#c	*411,235.000*	**2.996**	1.665	1.452	2.451	1.460	2.452
(sat)	#n	*7,383.000*	**3.292**	1.664	1.483	2.520	1.486	2.520
	time	*0.197*	**2.995**	1.670	1.452	2.508	1.472	2.518
bqwh − 18 − 141 − 40	#c	602,681,284.000	**2.086**	1.087	1.788	*0.932*	1.470	0.988
(sat)	#n	10,779,400.000	**2.190**	1.075	1.846	*0.929*	1.502	0.981
	time	321.383	**2.083**	1.085	1.773	*0.926*	1.450	0.984
bqwh − 18 − 141 − 68	#c	21,995,315.000	0.942	0.786	1.595	*0.740*	**2.129**	0.929
(sat)	#n	371,375.000	0.951	0.748	1.596	*0.716*	**2.086**	0.898
	time	12.504	0.937	0.783	1.561	*0.734*	**2.099**	0.921
frb50 − 23 − 3 − bis	#c	81,752,058,491.000	*0.960*	1.082	0.960	1.082	1.012	**1.106**
(sat)	#n	*187,967,335.000*	1.007	1.023	1.007	1.023	**1.064**	1.042
	time	*16,512.675*	1.006	1.032	1.020	1.019	**1.053**	1.034
qa − 6	#c	*21,950,814,589.000*	**1.121**	1.019	1.117	1.084	1.097	1.015
(sat)	#n	134,884,052.000	1.089	1.002	**1.095**	1.063	1.071	*0.999*
	time	3,083.689	1.106	1.011	**1.108**	1.072	1.096	*0.998*
graph14_f28	#c	5,142,167.000	0.528	1.387	0.528	**1.387**	*0.528*	1.387
(unsat)	#n	28,177.000	*0.348*	1.310	0.348	1.310	0.348	**1.310**
	time	0.751	*0.397*	1.611	0.399	1.610	0.406	**1.617**
graph2_f25	#c	65,664,798.000	0.950	0.994	*0.029*	1.016	0.030	**1.016**
(unsat)	#n	275,917.000	1.000	1.004	*0.015*	1.029	0.015	**1.029**
	time	8.021	0.982	1.014	*0.019*	**1.037**	0.020	1.035
scen6_w1_f2	#c	19,978,058.000	1.114	*0.690*	1.115	0.741	**1.125**	0.697
(unsat)	#n	52,325.000	1.084	*0.627*	1.084	0.674	**1.107**	0.654
	time	0.662	**1.166**	*0.690*	1.156	0.740	1.162	0.705
dual_ehi − 90 − 315 − 97	#c	**11,283,716.000**	0.280	0.961	0.283	0.228	*0.219*	0.977
(unsat)	#n	6,288.000	0.308	1.022	0.314	*0.228*	0.253	**1.100**
	time	10.166	0.235	1.024	0.234	0.206	*0.160*	**1.069**
qk_20_20_5_add	#c	3,001,318,053.000	95.518	0.794	**95.518**	0.794	84.080	*0.304*
(unsat)	#n	237,139.000	**102.069**	0.785	28.996	*0.207*	91.023	0.291
	time	51.371	**96.175**	0.800	27.350	*0.222*	85.000	0.308
qk_20_20_5_mul(unsat)	#c	4,257,470,286.000	59.509	0.602	16.949	*0.149*	**69.584**	0.171
	#n	360,924.000	64.125	0.587	18.229	*0.131*	**75.216**	0.154
	time	78.117	58.306	0.586	16.619	*0.144*	**68.420**	0.167
composed − 75 − 1 − 40 − 7	#c	205,748.000	**1.170**	0.756	1.170	0.751	1.167	*0.707*
(unsat)	#n	1,089.000	1.448	0.684	1.448	*0.613*	**1.460**	0.613
	time	0.020	1.150	0.650	1.150	*0.600*	**1.200**	0.650
cril_unsat_b_1	#c	**297,178,340.000**	*0.810*	0.930	0.816	0.930	0.818	0.930
(unsat)	#n	**1,672,114.000**	0.886	*0.851*	0.886	0.851	0.888	0.851
	time	**32.854**	0.904	*0.879*	0.904	0.884	0.909	0.885

to the interaction between variable and value ordering heuristics as explained in Section 3. Table 3 also confirms that lazy k-way branching can reduce the number of failures by up to one order-of-magnitude for some instances when compared with k-way branching. The minimum number of failures between lazy k-way and k-way branching schemes for each value ordering is made bold in each row of Table 3.

In some cases, despite failing more, the ratio of checks per node ($\#c/\#n$) is less for lazy k-way branching than with that of k-way branching, e.g., see the row corresponding to the qk_20_20_5_add instance. The reason is that when the problem is relatively hard, more nodes are explored. Consequently more nodes are shared among different branches. Hence, work required for constraint



(a) graph14_f28

(b) graph2_f25

(c) qk_15_15_5_mul

(d) bqwh-15-106-32

Fig. 5. Search effort for exploring the entire search space of different instances with 2500 random value ordering heuristics and dom/wdeg as a variable ordering heuristic.

propagation is also shared, which improves the trade-off between the number of decision nodes and the work done on each of them.

Value ordering heuristics like min-conflict, cruciality, and promise are proposed in the literature to find one solution quickly, and not for exploring the entire search space. Thus, when it comes to exploring the entire search-space, it is not surprising to see that, for some problem instances, value ordering heuristics like min-conflict perform significantly better than anti-heuristics like max-conflict, while for others it is the other way around, and for some there is only a marginal difference in their performance. We are not aware of any work on value ordering heuristics for finding all solutions with k-way branching in the CSP context. In fact for a long time it has been believed that value orderings do not make any difference in the search effort of backtrack algorithm with k-way branching. Contrary to that, we have shown results where the difference in the search effort is up to several orders-of-magnitude because of using different value orderings. These results raise an interesting question: *what kind of value ordering heuristics should be used with (lazy) k-way branching and adaptive variable ordering heuristics like dom/wdeg when it comes to exploring the entire search space.*

Table 3. Results for lazy k-way branching and k-way branching with different value orderings.

instance		**lex**		**min-conflict**		**max-conflict**		**promise**		**anti-promise**	
		lazy k-way	**k-way**	**lazy k-way**	**k-way**	**lazy k-way**	**k-way**	**lazy k-way**	**k-way**	**lazy k-way**	**k-way**
bqwh − 15 − 106 − 32	#c/#n	35	35	35	35	35	35	35	35	35	35
(sat)	#f	16,568	16,568	16,568	16,568	16,568	16,568	16,568	16,568	16,568	16,568
(dom/deg)	time	0.547	0.535	0.545	0.534	0.549	0.528	0.548	0.537	0.548	0.533
graph12_w1	#c/#n	227	155	204	158	565	158	204	158	564	158
(unsat)	#f	**62,029**	146,056	**62,041**	146,056	**42,058**	146,056	**62,041**	146,056	**42,058**	146,056
(dom/deg)	time	2.295	2.565	2.406	2.695	2.667	2.610	2.398	2.624	2.662	2.619
qk_12_12_5_mul	#c/#n	578	3,147	676	3,079	825	3,170	676	3,079	839	3,164
(unsat)	#f	**1,800,233**	1,933,202	**1,585,215**	1,933,202	**1,602,612**	1,933,202	**1,585,225**	1,933,202	**1,607,642**	1,933,202
(dom/deg)	time	100.182	127.459	107.223	124.576	122.685	127.909	107.625	125.882	124.593	126.916
bqwh − 18 − 141 − 40	#c/#n	53	55	52	53	52	56	51	54	52	56
(sat)	#f	6,060,698	**5,355,527**	**8,525,678**	11,829,875	6,340,885	**5,757,040**	11,304,683	**8,091,911**	5,688,069	**5,258,850**
(dom/wdeg)	time	370.067	321.383	499.935	669.521	381.778	348.624	652.630	466.117	346.182	316.384
bqwh − 18 − 141 − 68	#c/#n	54	59	54	58	57	62	55	60	55	61
(sat)	#f	262,036	**190,020**	**158,538**	179,877	165,934	**141,998**	**342,944**	393,687	**163,859**	169,572
(dom/wdeg)	time	16.906	12.504	10.276	11.720	11.277	9.796	22.456	26.247	10.928	11.510
qa − 6	#c/#n	111	162	110	167	112	165	110	166	113	165
(sat)	#f	**83,018,862**	83,567,571	**76,845,848**	93,016,524	**74,749,673**	83,676,659	**77,295,678**	90,932,653	84,307,042	**83,458,694**
(dom/wdeg)	time	3,288.929	3,083.689	3,110.823	3,410.738	2,961.077	3,117.568	3,121.990	3,381.146	3,304.104	3,077.301
graph14_f28	#c/#n	117	182	107	276	164	193	107	276	164	193
(unsat)	#f	**11,524**	21,831	38,497	**7,840**	**9,736**	28,355	38,497	**7,840**	**9,736**	28,355
(dom/wdeg)	time	0.578	0.751	1.876	0.298	0.547	1.210	1.879	0.305	0.555	1.214
graph2_f25	#c/#n	170	237	151	226	158	235	246	481	158	234
(unsat)	#f	**44,278**	210,765	**175,584**	211,438	**28,556**	210,651	**2,877**	3,292	**169,689**	217,072
(dom/wdeg)	time	2.689	8.021	8.839	7.876	1.471	8.131	0.175	0.158	8.696	8.304
scen6_w1_f2	#c/#n	178	381	254	392	315	420	258	387	307	406
(unsat)	#f	**15,128**	46,091	**22,016**	49,963	**12,512**	28,894	**19,300**	51,022	**16,125**	30,158
(dom/wdeg)	time	0.485	0.662	0.783	0.772	0.512	0.457	0.710	0.769	0.698	0.467
dual_ehi − 90 − 315 − 97	#c/#n	1,289	1,794	1,314	1,631	1,488	1,686	1,513	1,552	1,492	1,594
(unsat)	#f	**2,436**	3,522	1,409	**1,123**	**3,430**	3,566	**797**	907	**3,078**	3,889
(dom/wdeg)	time	5.936	10.166	3.039	2.392	10.174	10.411	1.730	1.628	9.340	10.867
qk_20_20_5_add	#c/#n	1,215	12,656	1,374	11,844	1,587	12,805	1,374	11,690	1,606	13,232
(unsat)	#f	**197,976**	229,562	**19,271,246**	22,899,364	182,798	**181,067**	25,928,724	**20,321,780**	**45,971**	67,920
(dom/wdeg)	time	23.677	51.371	2,648.326	4,940.583	29.024	41.113	3,580.520	4,366.546	7.480	15.800
qk_20_20_5_mul	#c/#n	1,184	11,796	1,315	10,946	1,533	12,099	1,342	10,912	1,606	13,106
(unsat)	#f	**327,484**	347,085	25,290,613	**21,841,327**	210,111	**205,860**	**24,401,630**	25,633,491	**43,872**	54,894
(dom/wdeg)	time	40.181	78.117	3,446.613	4,554.656	33.562	45.809	3,379.101	5,344.769	7.675	13.037
cril_unsat_b_1	#c/#n	138	177	111	162	147	194	111	163	147	194
(unsat)	#f	**677,349**	1,305,510	**723,522**	1,158,780	**700,502**	1,110,120	**730,559**	1,162,132	**700,502**	1,110,120
(dom/wdeg)	time	32.862	33.145	30.141	29.815	32.113	28.920	30.429	29.831	32.103	29.042
composed − 75 − 1 − 40 − 7	#c/#n	128	188	109	152	188	208	111	150	176	217
(unsat)	#f	**475**	901	**756**	1,334	**296**	602	**682**	1,353	**305**	532
(dom/wdeg)	time	0.018	0.020	0.028	0.023	0.013	0.013	0.027	0.024	0.014	0.013

7 Conclusions

Given recent developments in the area of variable ordering heuristics, the conventional wisdom with respect to k-way branching and value ordering needed to be reconsidered. We have presented an analysis in this paper demonstrating that value ordering can make a considerable difference in search effort. We demonstrated this phenomenon across multiple problem classes, and for two forms of k-way branching. One of our k-way branching schemes, lazy k-way, is very novel and merits a deeper investigation in the context of CSP solving.

A major contribution of this paper is that it motivates a new and fruitful line of research in the study of *value ordering* heuristics for proving unsatisfiability.

Acknowledgments

We thank Marc van Dongen for providing the example of Figure 1(a) and Diarmuid Grimes for helpful discussions. This work was supported by Science Foundation Ireland Grant 08/CE/I1423. Barry O'Sullivan is also supported by Science Foundation Ireland Grant 10/IN.1/I3032.

References

1. Thanasis Balafoutis and Kostas Stergiou. Adaptive branching for constraint satisfaction problems. In *ECAI*, pages 855–860, 2010.
2. C. Bessière and J.-C. Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ ?) on hard problems. In *Procs of CP'1996*, pages 61–75, 1996.
3. F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *Procs of ECAI'2004*, 2004.
4. D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the IJCAI'95*, pages 572–578, 1995.
5. P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of ECAI'92*, pages 31–35, New York, NY, USA, 1992.
6. J. Hwang and D.G. Mitchell. 2-way vs. d-way branching for csp. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming*, pages 343–357, 2005.
7. C. Lecoutre. *Constraint Networks Techniques and Algorithms.* Wiley, 2009.
8. D. Mehta and M.R.C. van Dongen. Static value ordering heuristics for constraint satisfaction problems. In *Proceedings of CPAI05 workshop held with CP05*, pages 49–62, 2005.
9. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 125–129. John Wiley and Sons, 1994.
10. D. Sabin and E.C. Freuder. Understanding and improving the MAC algorithm. In G. Smolka, editor, *Principles and Practice of Constraint Programming*, pages 167–181. Springer Verlag, 1997.
11. B.M. Smith and P. Sturdy. Value ordering for finding all solutions. In *Proceedings of the IJCAI'2005*, pages 311–316, 2005.