Lazy Branching for Constraint Satisfaction

Deepak Mehta, Barry O'Sullivan, Lars Kotthoff and Yuri Malitsky
Cork Constraint Computation Centre
Department of Computer Science, University College of Cork, Ireland
{*d.mehta—b.osullivan—l.kotthoff—y.malitsky*}*@4c.ucc.ie*

***Abstract*—When solving a constraint satisfaction problem using a systematic backtracking method, the branching scheme normally selects a variable to which a value is assigned. In this paper we refer to such strategies as *eager* branching schemes. These contrast with the alternative class of novel branching schemes considered in this paper whereby having selected a variable we proceed by removing values from its domain. In this paper we study such *lazy* branching schemes in depth. We define three lazy branching schemes based on k-way, binary and split branching. We show how each can be incorporated into MAC, and define a novel value ordering heuristic that is suitable in this setting. Our results show that lazy branching can significantly out-perform traditional branching schemes across a variety of problem classes. While, in general, neither lazy nor eager branching dominates the other, our results clearly show that choosing the correct branching scheme for a given problem instance can significantly reduce search effort. Therefore, we implemented a variety of branching portfolios for choosing amongst all of the branching strategies studied in this paper. The results demonstrate that a good branching scheme can be automatically selected for a given problem instances and that including lazy branching schemes in the portfolio significantly reduces runtime.**

I. Introduction

The traditional approach to solving a constraint satisfaction problem (CSP) is based on depth-first search combined with polynomial-time inference at each node in the search tree [12]. While the CSP is well-known to be NP-Complete in general, the research community has focused significant efforts in studying alternative heuristics for selecting the order in which variables should be instantiated, and heuristics for choosing which value to use [2]. The objective of such research is to improve the efficiency of search for particular classes of CSP. Other recent work has studied the effect of ordering heuristics and branching schemes on finding all solutions [8].

The traditional branching schemes used when solving constraint satisfaction problems (CSPs) are k-way, binary and split branching. In each case a variable is selected for assignment. In both k-way and binary branching schemes a value is selected from the domain of the current variable to assign to it, while in split branching the domain of the current variable is restricted to half of its current domain. We refer to such strategies as *eager* branching schemes.

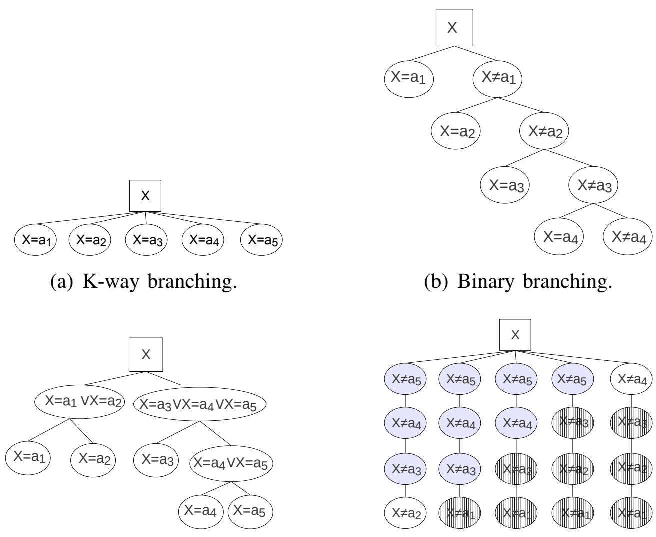
In this paper we propose and study *lazy* versions of each of the traditional branching schemes. In these schemes each time a variable is considered, rather than selecting a value, or several candidates as in the case of split branching, we restrict its domain by eliminating one or more values. We define three lazy branching schemes based on the eager schemes mentioned above. We show a specific modification of the MAC algorithm in each case to exploit the lazy branching strategy used. We also propose a novel value ordering heuristic that is appropriate for lazy branching.

The conventional wisdom is that such strategies are not interesting because they increase the number of nodes in the search tree, which has knock-on consequences for the cost of propagation. However, our experimental results show that lazy branching can significantly out-perform traditional branching schemes across all performance metrics for a variety of problem classes. While, in general, neither eager nor lazy branching dominates the other, we show that depending on the problem instance at hand that each can significantly out-perform the other, i.e. that the set of branching strategies we consider, both lazy and eager, are complementary. Therefore, we implemented a variety of branching portfolios for choosing amongst all of the branching strategies studied in this paper. The results demonstrate that a good branching scheme can be automatically selected for a given problem instances and that including lazy branching schemes in the portfolio significantly reduces runtime.

II. 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 and 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. Without loss of generality, 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



(a) K-way branching.
(b) Binary branching.
(c) Split branching.
(d) lazy branching.

Figure 1. Different branching schemes. The ordering over the values of X is assumed to be a_1, a_2, a_3, a_4 and a_5.

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 $\mathcal{P} = \perp$. 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. Solving a CSP involves either finding one (or more) solution or proving that the CSP is unsatisfiable.

A branching strategy defines a search tree. The well-known branching schemes are k-way branching, binary branching [12] and split branching. An empirical study of these branching strategies is performed in [10]. 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. An example of k-way branching is illustrated in Figure 1(a), 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 so $k = 5$.

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 $a_1, a_2, \ldots, a_k$, then two branches are formed for the value a_1, associated with $X = a_1$ and $X \neq a_1$ 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 $X \neq a_1$ is propagated, before selecting another variable-value pair. An example is illustrated in Figure 1(b), where a box denotes a variable selection and an ellipse denotes positive/negative decision.

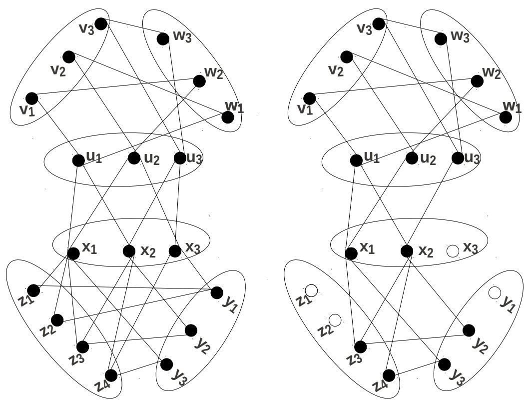
In split branching, when a variable X is selected, its domain is divided into two sets: $\{a_1, \ldots, a_j\}$ and $\{a_{j+1}, \ldots, a_k\}$, where $j = \lfloor k/2 \rfloor$. Two branches are formed by removing each set of values from $D(X)$ respectively. An example is presented in Figure 1(c), where a box denotes a variable selection and an ellipse denotes that reduction of the domain by half. For simplicity sake, we focus on the restricted versions of binary and split branchings where the new variable is selected only after initializing the current variable.

III. Lazy Branching

Both k-way and binary branching are *eager* branching schemes whereby, based on some heuristic measure, a value, a, is selected and assigned to a selected variable, X. If the values assigned to a subset of variables are involved in a solution then the search is on the correct path. Otherwise, it can thrash too many times before refuting the decision $x = a$. When the refutation occurs in k-way branching a new untried value is assigned to the variable whereas in binary branching the same is done after propagating $x \neq a$. Split branching is less eager, since instead of assigning a value to a variable, its domain is split into two mutually exclusive subsets. Two branches corresponding to these two subsets are formed and the variable is instantiated when a subset contains only one value. Although split branching is less eager, it is not completely pessimistic since the domain is reduced to half in one shot.

We propose a *lazy* branching scheme. Instead of selecting and assigning a value to a variable, we select and remove a value from the variable's domain, thus instantiating variables lazily. For example, the assignment $X = a_1$, is equivalent to removing a_2, a_3, a_4 and a_5 progressively from the domain of X. Figure 1(d) is an example of a very simple lazy branching scheme. Each branch corresponds to an assignment of a variable which is done lazily. Instantiating a variable lazily may help in making better decisions, and may reduce the number of failures/mistakes. For example, we might use a value ordering heuristic to find a value that has the least chance of being part of a solution, remove it from the domain and propagate its effect. As search progresses, the value ordering heuristic measure may change and thus may help us make better decisions. We explain this with an example.

Figure 2 depicts part of the micro-structure of an instance of a CSP in which an edge represents a pair of values that are allowed by a constraint. Let us assume that only variable



(a) Before removing x_3. (b) After removing x_3.

Figure 2. A micro-structure of a part of a CSP.

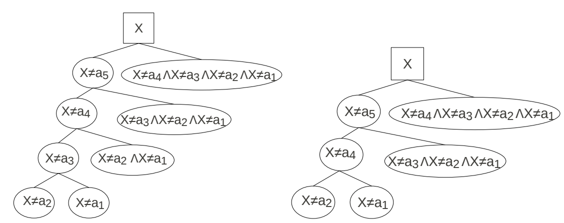
X is connected to the remaining part of the problem, which is not shown in Figure 2(a), and each value of X has the same number of conflicts with respect to the variables, which are also not shown in Figure 2(a). Further assume that the variable X is selected for instantiation and the heuristic *min-conflict* is used to select a value. If an eager branching scheme is used then X would be instantiated to x_1 since the sum of the min-conflicts with respect to variables U, Y, and Z (which are 1, 1 and 2 respectively) is minimum. Notice that when X is instantiated to x_1 there does not exist any satisfiable assignment for variables U, V and W. One can easily think of scenarios where realizing this can be very challenging for a systematic search algorithm. However, if a lazy branching scheme is used whereby the max-conflict value is selected and removed then the value x_3 would be removed and the result would be Figure 2(b). Notice that before removing x_3, the number of conflicts of x_1 and x_2 were 4 and 5 respectively, so x_1 is preferred over x_2. But after removing x_3 they are 3 and 2 respectively and therefore x_2 is preferred over x_1. If there exists a solution involving value x_2 then one can save the effort spent in proving that $X = x_1$ cannot lead to a solution. Assigning variables lazily might reduce the number of mistakes.

Another advantage of assigning variables lazily 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 when using the MAC algorithm. These dependencies can be exploited to reduce the number of decisions. For example, if 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 1(d). 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 1(d) can be avoided.

In Figure 1(d) there are many nodes that are common to different branches and so they can be shared among them. Below we show three different ways of factoring out the common nodes, which lead to lazy versions of k-way, binary and split branching schemes. By lazy k-way and lazy binary branching we mean that the instantiation of a variable is done lazily. By lazy split branching we mean that the domain of the variable is split lazily. In the following sections we describe these lazy branching schemes and assume that a reverse lexicographic ordering is used for removing the values from the selected variable.

A. Lazy k-way Branching

In Figure 1(d), the first two assignments of X (or branches) have three nodes in common, the first three assignments have two nodes in common and the first four assignments have one node in common. Figure 3(a) is a result of sharing them among different branches. The node on the left branch corresponds to a negative decision $X \neq a_i$ and the node on the right branch is a decision of removing values which have already been tried. More precisely, the right branch is a set of negative decisions $X \neq a_j$ such that $j < i$, which results in the positive decision $X = a_i$. Each leaf node in Figure 3(a) corresponds to an assignment of X.



(a) Lazy k-way example. (b) Exploiting dependencies.

Figure 3. Lazy k-way branching.

An inherent feature of k-way branching is that whenever a decision, $X = a$, is proved to be false, $X \neq a$ is propagated in each unexplored branch emanating after selecting X. For example, when $X = a_1$ is refuted, $X \neq a_1$ is propagated in each subsequent branch emanating after selected variable X in Figure 1(a). Notice that this feature is also present in Figure 3(a). Therefore we call this *lazy k-way branching*.

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 work required for propagating the impact of removing a value is repeated. This repetition is reduced in lazy k-way branching since the nodes corresponding to some negative decisions are shared over different assignments. Another advantage is that some assignments can be avoided. For example, as explained in previous section if $X \neq a_3$ is removed after propagating

$X \neq a_4$ and $X \neq a_5$ then $X = a_3$ can be avoided. The resultant tree for lazy k-way branching is shown in Figure 3(b). The work of van Hoeve and Milano in [13] can be seen as a specific case of lazy k-way branching if the decisions are only postponed when there are ties among the values.

Algorithm 1 presents pseudo-code to incorporate lazy k-way branching in MAC algorithms. MAC$_{LK}$ requires CSP $\mathcal{P}$ and the current variable Y. If Y is null then a new variable is selected (Line 2). After the current variable, X, is determined, the domains are saved in D' (Line 5). A value v is selected and removed from $D(X)$, and AC is enforced (Line 5-6). If $\mathcal{P}$ is arc-consistent then the left branch is created (Line 7-9). If X is not instantiated then MAC$_{LK}$ is invoked with the current variable X. The right branch is created by restoring the domains to D' and by initializing X to v. As a result X will never be assigned any untried value that is removed as a result of enforcing arc consistency after making one or more negative decisions involving the variable X.

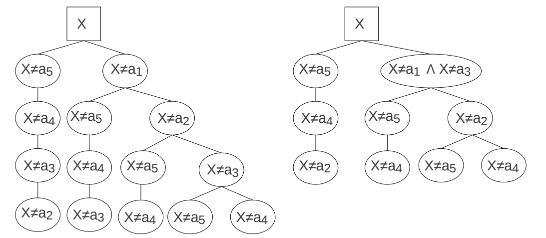
B. Lazy Binary Branching

An inherent feature of binary branching is that when a refutation occurs it is propagated before assigning any other value to the variable. Each subsequent refutation is shared over all the subsequent assignments. In Figure 1(d) if $X \neq a_1$ is shared in the last four branches, $X \neq a_2$ is shared in the last three branches, and $X \neq a_3$ is shared in the last two branches then the result is a search tree as shown in Figure 4(a). We call this lazy binary branching. As before if a_3 is removed from $D(X)$ after propagating $X \neq a_5$ and $x \neq a_4$ in the first branch then it can be removed before exploring the subsequent branches which is shown in Figure 4(b).

Algorithm 2 presents pseudo-code to incorporate lazy binary branching in MAC algorithms. Lines 5-7 repeatedly select and remove a value from the domain until the current variable is uninstantiated and there is no domain-wipeout. Each selected value is added to the set V. When the loop terminates, the set V contains all the values that were selected and removed and are not yet assigned to the variable X. Notice that V does not contain any value that was

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

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



(a) Lazy binary example. (b) Exploiting dependencies.

Figure 4. Lazy binary branching.

removed as a result of enforcing arc consistency in the left branch. If the right branch is explored then $D(X)$ is set to V.

Algorithm 2 MAC$_{LB}(\mathcal{P}, Y)$

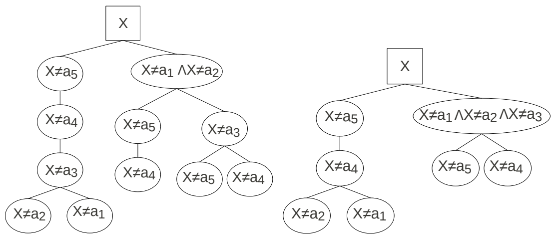
Require: $\mathcal{P}$: input CSP $(\mathcal{X}, \mathcal{C}, D)$; Y: current variable
1: **if** $\mathcal{X} = \emptyset$ **then** solution found and stop search
2: **if** $Y = \texttt{null}$ **then** select and remove any variable X from $\mathcal{X}$
3: **else** $X \leftarrow Y$
4: $V \leftarrow \emptyset$;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

C. Lazy Split Branching

Another way of sharing nodes among different branches of Figure 1(d) is to share the first three decision nodes amongst the first two branches and last two decision nodes amongst the last three branches. The result is shown in Figure 5(a). This is called lazy split branching. Although in split branching the assignment of a variable is also done lazily: a subset of values are determined and removed eagerly. In lazy split branching both the subset of values that is removed from the domain and its cardinality are determined lazily.

Pseudo-code for implementing lazy split branching in MAC is shown in Algorithm 3. 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 7). 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 5-7). When the loop is terminated and if no domain is empty then the left branch is created (Line 8-9). The right branch is created by restoring the domains to D' (Line 10) and setting $D(X)$ to the set V, which is the set of values that were removed earlier.

For example for Figure 5(a) the loop (in Line 5) is terminated after the node $X \neq a_3$ (in the left branch). When the algorithm backtracks it first removes all those values that



(a) Lazy split example. (b) Exploiting dependencies.

Figure 5. Lazy split branching.

are already tried as assignments to the current variable, e.g., in Figure 5(a) $X \neq a_1 \wedge X \neq a_2$ is enforced on the right branch. The algorithm also removes all those values of X that were removed while enforcing arc-consistency in the left branch. For example, if a_3 is removed when arc consistency is enforced after the decision $X \neq a_4$ then $X \neq a_3$ is also enforced in the right branch node as shown in Figure 5(b). This is done in Line 11 of MAC_{LS} when $D(X)$ is set to V, which in this case contains only a_4 and a_5.

Algorithm 3 $\text{MAC}_{LS}(\mathcal{P}, Y)$

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

IV. Value Ordering for Lazy Branching

Two well-known value ordering heuristics for eager branching schemes that are based on the notion of support are *min-conflict* [2] and *promise* [3]. The heuristic *min-conflict* associates with each value of each variable the sum of the number of values in the domains of the other variables that are *not supported*. The values are then considered in the increasing order of this count. The heuristic *promise* associates with each value the product of the number of *supported* values in the domain of each variable. The value with the highest product is chosen subsequently. For a eager branching scheme the value that has the highest chance of leading search to a solution is selected and assigned to the variable. But in a lazy branching scheme we want to select and remove the value from a variable that has the least chance of leading search to a solution. An adaptation of min-conflict and promise for lazy branching schemes would be max-conflict and anti-promise, respectively.

In k-way and binary branching when the min-conflict value ordering heuristic is used, the number of conflicts associated with a value indicates how many values will be removed from the other domains, which is not necessarily true for lazy branching schemes. The reason is that the values of other domains that are in conflict with the removed value might be supported by the other values of the same variable. For example, let us assume that $D(X) = \{a_1, \ldots, a_{10}\}$ and a_1 and a_2 have 5 and 7 conflicts in $D(Y)$ respectively. It is possible that removing a_1 from $D(X)$ may remove more values than removing a_2 from $D(X)$. This would happen when a_1 is supporting more values of $D(Y)$ for which it is unique in $D(X)$. Based on this idea we propose a value ordering heuristic for lazy branching schemes. Let $\texttt{unique}(X, Y, a) =_{def} |\{b : |\{(a, b) \in C_{XY}\}| = 1\}|$ be the number of values in $D(y)$ for which $a \in D(X)$ is the only support in $D(x)$. We select the value $a \in D(X)$ whose $\sum_{C_{XY} \in \mathcal{C}} \texttt{unique}(X, Y, a)$ is minimal. We call this the *min-removal* value ordering heuristic.

V. Experimental Results

Our experimental evaluation is two fold. First in Section V-A we compare the performance of lazy and eager branching schemes. The primary objective of these experiments is to investigate whether lazy branching schemes dominate, or are dominated by, eager schemes. We find that both lazy and eager schemes can significantly out-perform each other. Therefore, a second objective of our experiments is to determine the extent to which lazy and eager schemes are complementary. We find that if the branching scheme corresponding to the best runtime for a given instance can be chosen, the overall reduction in search effort can be significant.

In Section V-B we develop a number of portfolios of branching schemes that try to select the best branching scheme for each instance. These experiments show that the complementary nature of eager and lazy branching schemes can be exploited in practice. Specifically, several portfolios have performance comparable with selecting the best possible branching strategy for a given problem instance.

A. Comparing Eager and Lazy Branching

Throughout our experiments and branching schemes we use dom/wdeg [1] as a variable ordering heuristic. We use the min-conflict value ordering heuristic for eager branching schemes and the min-removal value ordering heuristic with max-conflict as a tie-breaker for lazy branching schemes, as discussed earlier in this paper. Search effort was measured in terms of visited nodes, failures and the solution time in seconds. All algorithms were implemented 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 the binary CSPs, however, the principle of lazy branching is not limited to them.

Random Binary CSPs. We first experimented with Model B random binary problems [4], where a random CSP instance is characterised by (n, d, p_1, p_2), where n is the number of variables, d is the uniform domain size of each variable, p_1 is the density of the graph, and p_2 is the tightness of a constraint. For each combination of $(n, d) \in \{(20, 80), (30, 70), (40, 40)\ (40, 80), (50, 50)\}$ and $p_2 \in \{0.65, 0.7, 0.75, 0.8, 0.85\}$ we computed the critical value of p_1 and generated 20 instances. These experiments clearly show that each scheme can be out-performed by orders of magnitude by another. We do not present these results in detail for space reasons.

We also generated a class of instances by merging a Model B random instance with the structure depicted in Figure 2. Thereby we created a set of instances in which an early mistake during search would make search very challenging. The results are shown in the first two rows of Table I, where hc_1 and hc_2 denote that $(40, 10, 0.93, 0.11)$ and $(50, 10, 0.81, 0.10)$ Model B random classes were used to merge with the structure described before. Notice that lazy branching can significantly out-perform eager branching.

Non-Random CSPs. We also performed experiments on instances of balanced quasi-group with holes (qwh), quasi-completion (qcp), modified radio-link frequency assignment (rlfap), forced random binary (frb), queens attacking (qa), geometric (geo) and dual encoding of 3-SAT problems (ehi) that were used as benchmarks in the first CP solver competition.[1] Some results on which lazy branching performs better than eager branching are shown in Table I.

Table II demonstrates the complementary nature of eager and lazy branching. In this table, for each problem class, we present the cumulative runtime required to solve all instances in the class; the number of instances is given for each class. For each scheme (k-way, binary and split) we present both cumulative time for eager and lazy branching, along with the cumulative time associated with picking the best runtime per instance, in the oracle columns, corresponding to a choice of the best branching scheme in each case. We highlight the oracle time in bold when it is better than the best of the two branching schemes, which demonstrates that they complement each other.

We also present in the three right-most columns the oracles for each of the eager, lazy and overall strategies. These correspond to the cumulative time associated with the best choice of the three eager strategies, the three lazy strategies, or all six strategies, respectively. These results clearly motivate the value, and complementarity, of both eager and lazy branching schemes, and demonstrates that if one could select amongst the six branching schemes presented in this paper to each problem instance that significant performance improvements would be observed. We tackle this question in the next section.

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

Table I
EXAMPLES OF INSTANCES ON WHICH LAZY BRANCHING OUTPERFORMS EAGER BRANCHING.

		k-way		binary		split	
problem instance	metric	eager	lazy	eager	lazy	eager	lazy
	fails	1079410	38865	1214869	38055	1208956	35862
hc_1	time	73.00	2.88	87.23	2.95	86.48	2.71
	nodes	2015179	77779	2429754	81966	2417933	76386
	fails	23M	182395	24M	171425	24M	157596
hc_2	time	1940.79	16.76	2175.54	15.91	2163.81	14.44
	nodes	44M	364845	49M	360033	49M	329590
	fails	5718055	2159983	2226450	2159983	2226450	2159983
qwh-o25-h235-b27	time	255.03	91.73	96.54	94.10	97.929	93.09
	nodes	10949267	4320004	4452933	4551590	4452933	4537049
	fails	90M	7M	91M	7M	91M	7M
qwh-o25-h235-b993	time	5022.39	392.25	5058.77	402.64	5071.86	391.30
	nodes	178M	13M	183M	14M	183M	14M
	fails	45M	248696	56M	248696	56M	248696
qcp-o20-h187-b217	time	1737.26	12.91	2149.39	13.30	2146.02	12.08
	nodes	87M	497418	112M	509371	112M	509079
	fails	897M	224M	1146M	224M	1146M	224M
qcp-o20-h187-b6	time	31594.86	7576.98	39243.08	7839.63	39334.80	7590.79
	nodes	1756M	449M	2293M	473M	2293M	471M
	fails	35425	18434	28340	9243	25609	10427
scen2_f25	time	1.13	0.99	1.30	1.12	1.18	0.94
	nodes	41279	36866	56678	52938	51216	34456
	fails	3047953	1774349	2558887	920263	2341643	1016562
scen11_f7	time	73.00	61.27	88.36	88.30	79.34	68.17
	nodes	3643215	3548696	5117772	5357476	4683284	3266799
	fails	13020	156769	173442	54649	225840	162809
graph2_f25	time	0.27	4.44	4.31	2.42	5.72	5.84
	nodes	16501	313536	346882	220284	451678	446554
	fails	21955	7524	20320	5694	19595	5720
graph9_f10	time	0.96	0.43	1.04	0.56	1.03	0.48
	nodes	29154	15046	40638	26833	39188	16461
	fails	108088	57564	104753	73214	105026	54856
frb40-19-5-bis	time	6.14	3.72	6.65	5.63	6.76	3.99
	nodes	174924	115166	209517	187345	210065	128149
	fails	2577022	107078	2493696	102227	2505655	106549
frb50-23-5-bis	time	182.79	8.52	199.38	9.76	201.95	9.60
	nodes	4181106	214210	4987408	265615	5011331	249779
	fails	855640	479928	747821	557990	733060	519875
qa_6	time	20.32	11.83	16.98	16.24	17.23	14.22
	nodes	1326441	959903	1495651	1637222	1466135	1269118
	fails	160M	135M	126M	131M	122M	129888725
qa_7	time	5042.21	4846.26	4037.86	5345.11	3938.97	4792.47
	nodes	245M	271M	253M	381M	245M	315M
	fails	35498	1237	34054	767	34893	1205
geo50.20.d4.75.8	time	2.07	0.08	2.22	0.06	2.30	0.09
	nodes	56685	2532	68126	2118	69812	2892
	fails	27570	11534	26625	11839	26573	11238
geo50.20.d4.75.90	time	1.89	0.93	2.12	1.147	2.14	1.03
	nodes	43075	23116	53264	31148	53168	26406
	fails	1385	645	1714	1041	1432	1043
dual_ehi-85-297-95	time	1.88	0.75	1.99	1.35	1.87	1.22
	nodes	2518	1288	3426	2471	2862	2366
	fails	1020	458	1040	385	1014	386
dual_ehi-90-315-86	time	1.40	0.34	1.23	0.33	1.19	0.32
	nodes	1832	914	2078	1004	2026	912

B. A Portfolio Approach to Branching Strategy Selection

With the current success of portfolio approaches in a plethora of distinct fields [16], [6], [9], [11], there has been a flux of research into how to most effectively decide the best solver or approach to use for the instance at hand. There are a number of different approaches. The three most prevalent ones are as follows. Algorithm selection can be treated as a classification problem, where the label to predict is the solver to use. Alternatively, the training instances can be

Table II

CUMULATIVE TIME PER PROBLEM CLASS FOR EACH BRANCHING SCHEME. IN EACH CASE OF K-WAY, BINARY AND SPLIT BRANCHING WE SHOW THE CUMULATIVE TIME ASSOCIATED WITH PICKING THE BEST BRANCHING STRATEGY ON AN INSTANCE LEVEL (ORACLE). WE ALSO PRESENT THE CUMULATIVE GLOBAL ORACLE TIME WHICH ASSUMES THAT THE BEST OF ALL SIX BRANCHING STRATEGIES WAS CHOSEN.

problem class	number of instances	k-way			binary			split			overall oracles		
		eager	lazy	oracle	eager	lazy	oracle	eager	lazy	oracle	eager	lazy	overall
hc_1	20	1460.02	57.54	**56.92**	1744.56	58.93	58.93	1729.53	54.13	54.13	1458.75	51.55	**50.99**
hc_2	20	38815.81	335.16	335.16	43510.86	318.23	318.23	43276.16	288.82	288.82	38438.70	257.29	**257.29**
qwh-25 (easy)	5	307.24	211.11	**143.88**	209.59	217.93	**184.17**	210.71	213.17	**181.31**	148.74	211.11	143.88
qwh-25 (hard)	11	29922.42	51226.26	**24209.71**	28780.83	52739.09	**23110.31**	28806.08	51380.68	**23019.07**	28266.51	51219.75	**22923.39**
qcp-20 (easy)	5	595.38	8725.64	**193.53**	976.66	9059.05	**212.17**	987.40	8773.19	**203.55**	592.07	8725.64	**192.92**
qcp-20 (hard)	10	35413.83	9341.84	**9235.24**	43715.12	9663.34	**9505.99**	43793.27	9387.81	**9253.63**	35359.13	9339.96	**9216.46**
rlfapModScens (unsat)	9	2.03	1.86	**1.82**	2.27	2.55	**2.07**	2.04	1.89	**1.80**	1.98	1.79	**1.75**
rlfapModScens (sat)	5	0.25	0.29	0.25	0.27	0.39	0.27	0.28	0.33	0.28	0.25	0.29	0.25
rlfapModGraphs (unsat)	6	0.36	3.92	0.36	0.80	0.35	**0.28**	0.41	0.81	**0.28**	0.36	0.35	**0.27**
rlfapModGraphs (sat)	6	1.47	6.06	**0.94**	5.71	4.47	**3.33**	6.98	57.16	**6.42**	1.46	4.04	**0.93**
frb40-19	5	28.98	31.14	**26.48**	31.44	42.13	**30.42**	30.89	38.30	**28.05**	28.98	31.04	26.48
frb45-21	5	255.42	273.50	**206.08**	279.88	291.90	**222.24**	280.75	317.42	**262.74**	255.42	230.58	**202.56**
frb50-23	5	8550.87	9499.98	**8375.69**	9249.62	10937.17	**9060.00**	8758.83	9328.67	**8257.28**	8550.87	8973.38	**8200.52**
qa	3	5062.56	4858.12	**4858.11**	4054.88	5361.40	4054.14	3956.24	4806.71	**3953.22**	3955.98	4804.32	**3950.82**
geom (unsat)	8	91.61	107.37	91.61	105.02	128.54	105.02	104.91	119.46	104.91	91.61	107.37	91.61
geom (sat)	92	40.09	45.48	**35.92**	45.28	48.31	**39.68**	44.70	48.08	**40.10**	40.04	43.26	**35.04**
ehi-85	100	38.61	48.03	**30.08**	40.32	52.38	**32.41**	40.05	48.19	**32.55**	32.36	40.19	**26.35**
ehi-90	100	40.24	51.49	**34.50**	42.68	53.45	**33.35**	40.94	52.12	**33.77**	34.25	43.57	**28.68**
bqwh15-106	100	1.29	1.18	**0.82**	1.38	1.24	**0.86**	1.38	1.19	**0.85**	1.15	1.18	**0.77**
bqwh18-141	100	23.37	22.33	**15.64**	24.78	23.14	**17.30**	24.67	22.45	**17.07**	20.00	22.30	**14.49**

clustered and the best solver chosen for each cluster. The third approach is to predict the runtime of each solver on a particular instance using a regression model and choose the solver with the lowest predicted runtime.

We investigate all three approaches. We converted each instance of our non-random problem set into SAT using the direct encoding [14]. In each case, we use the same 54 base SAT features as those used in SATzilla [15]. These features take into account aspects of the problem such as the number of variables, number of clauses, number of variables per clause, the number of clauses a variable typically appears in, the percentage of positive to negative literals per clause, and various other measures. We evaluate the performance of each approach on the three possible portfolios – just the eager branching schemes, just the lazy branching schemes and both combined – using ten-fold cross-validation. The entire set of instances is split into ten sets of roughly equal size. The combination of nine of these sets is used for training and the remaining one for testing. This process is repeated until every subset has been used for testing. For branching scheme selection as a classification problem we use the `AdaBoost`, `BayesNet`, `DecisionTable`, `IBk` with 1, 3, 5 and 10 neighbours, `J48`, `JRip`, `MultilayerPerceptron`, `OneR`, `PART`, `RandomForest`, `RandomTree` and `SVM` with radial basis function and sigmoid function kernels machine learning algorithms. For clustering, we use ISAC [7]. For the regression-based approaches, we use the algorithms `AdditiveRegression`, `GaussianProcesses`, `LinearRegression`, `M5P`, `M5Rules`, `REPTree`, `SMOreg` and `SVM` with ε and ν kernels. For all algorithms except ISAC, we use the Weka [5] implementations.

Table III shows the results for the three types of different portfolios. We compare each portfolio on PAR10 score, based on a timeout of 3600 seconds. We compute the mean average over the PAR10 scores of all instances. For each problem instance, we record the time it takes to solve it using the chosen solver, or record ten times the timeout if it was not solved. Also presented in the first two lines of this table is the performance of the single best branching strategy, and the performance of the oracle which selects the best possible strategy on an instance-by-instance basis.

For both the eager and lazy portfolios, there is practically no gap between the best single solver and the oracle. In these cases, we are better off just sticking to only one of the heuristics, as the potential benefits of choosing one dynamically are very small. If the eager and lazy schemes are combined into a single portfolio however, there is a dramatic difference between selecting the same branching scheme for all instances and choosing the best one. It is clearly beneficial to attempt to dynamically select a branching strategy. The 'combined' column shows the performance of different machine learning methods for selecting amongst both eager and lazy branching schemes.

Table III
RESULTS ACROSS THE DIFFERENT PORTFOLIO APPROACHES. THE PAR10 SCORES WHERE THE COMBINED PORTFOLIO BEATS BOTH THE EAGER AND LAZY PORTFOLIOS ARE SHOWN IN **BOLD**.

	PAR10 Score		
	eager	lazy	combined
single best strategy	431	426.3	431
oracle	428.5	425.2	303.2
AdaBoost	431	425.6	431.2
AdditiveRegression	430.6	425.8	**311.4**
BayesNet	430.4	425.7	**364.4**
DecisionTable	430.8	425.8	**424.1**
GaussianProcesses	431.5	426	**368.4**
IBk 1	430.5	425.9	**306.5**
IBk 3	431.6	426.2	**363.7**
IBk 5	431.6	425.6	**362.5**
IBk 10	430.1	425.7	**420.9**
J48	431.7	426.1	**366.4**
JRip	430.9	426.1	427.1
LinearRegression	432.8	426.2	**425.8**
M5P	432	426	429.1
M5Rules	433	426	**425.6**
MultilayerPerceptron	430.3	425.7	**367.6**
OneR	432.1	425.7	**307.1**
PART	431	426.3	**365.5**
RandomForest	430.6	426.1	**364.5**
RandomTree	431.1	426.2	**362.9**
REPTree	431.2	425.8	**365.1**
SMOreg	431.4	425.8	426.1
SVM rbf	431.3	426.1	**369.4**
SVM ν	431	425.6	425.6
SVM sf	431	426.3	431.7
ISAC	432.8	426	**370.4**
SVM ε	431	425.6	425.6

The performance data of the actual portfolios shows that in the vast majority of cases, we are able to exploit the difference between the single best heuristic and the oracle in practice. There are a few cases where the combined portfolio performs worse than just eager or just lazy, but keep in mind that our approaches have not been tuned and the performance could be improved. The reason for presenting many machine learning algorithms is to demonstrate that this good behavior is robust across a variety of learning methods. In general, we can easily train a model to select algorithms from the combined portfolio by applying existing techniques in a more or less straightforward fashion.

VI. CONCLUSIONS AND FUTURE WORK

We have proposed lazy forms of branching for solving CSPs. Our experiments demonstrate the complementary nature of eager and lazy branching, motivating the study of machine learning-based approaches to selecting a strategy for a given instance. We also explored a variety of machine learning-based portfolios for selecting a branching strategy for a given instance and showed that it is possible to significantly out-perform the single best branching strategy. This paper presents, for the first time, the practicality of using lazy branching schemes when satisfying constraint satisfaction problems. In this paper we have studied extreme forms of branching schemes where either we remove 1 value (lazy branching) or $k-1$ number of values (eager branching). In future, we would like to investigate more general forms of branching schemes where it is possible to adapt the removal of number of values between 0 and k.

REFERENCES

[1] F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *Procs of ECAI'2004*, 2004.

[2] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Procs. of the IJCAI'95*, pages 572–578, 1995.

[3] P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of ECAI'92*, 1992.

[4] I.P. Gent, E. MacIntyre, P. Prosser, B.M. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. *Journal of Constraints*, 6(4):345–372, 2001.

[5] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[6] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. CP, pages 454–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac – instance-specific algorithm configuration. *ECAI*, pages 751–756, 2010.

[8] Deepak Mehta, Barry O'Sullivan, and Luis Quesada. Value ordering for finding all solutions: Interactions with adaptive variable ordering. In *Procs. of CP'11*, pages 606–620, 2011.

[9] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *AICS*, 2008.

[10] Vincent Park. An empirical study of different branching strategies for constraint satisfaction problems, 2004.

[11] Luca Pulina and Armando Tacchella. A multi-engine solver for quantified boolean formulas. CP, pages 574–589, Berlin, Heidelberg, 2007. Springer-Verlag.

[12] D. Sabin and E.C. Freuder. Understanding and improving the MAC algorithm. In *Procs. of CP'97*, pages 167–181, 1997.

[13] Willem Jan van Hoeve and Michela Milano. Postponing branching decisions. In *ECAI*, 2004.

[14] Toby Walsh. Sat v csp. In Rina Dechter, editor, *CP*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000.

[15] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *JAIR*, 32(1):565–606, 2008.

[16] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. volume 32, pages 565–606, June 2008.