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, \( P \), is a triple \((X, C, D)\) where \( X \) is a set of variables and \( C \) is a set of constraints. Each variable \( X \in X \) 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...
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. 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 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 \( D(\mathcal{X}) \) respectively. An example is presented in Figure 1(b), 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

![Figure 1](image-url)
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 x₁ 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 x₁ 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 x₃ would be removed and the result would be Figure 2(b). Notice that before removing x₃, the number of conflicts of x₁ and x₂ were 4 and 5 respectively, so x₁ is preferred over x₂. But after removing x₃ they are 3 and 2 respectively and therefore x₂ is preferred over x₁. If there exists a solution involving value x₂ then one can save the effort spent in proving that X = x₁ 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 a₃ is removed from D(X) when arc consistency is enforced after taking the negative decisions X ≠ a₅ and X ≠ a₄ in the first branch of Figure 1(d). One can infer the following implication: X ≠ a₅ ∧ X ≠ a₄ → X ≠ a₃.

This effectively means that there does not exist any solution in the resulting subproblem after selecting variable X where X = a₃. Therefore, if the decision of instantiating X to a₃ 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 ≠ a₁ 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 ≠ a₃ such that j < i, which results in the positive decision X = a₁. Each leaf node in Figure 3(a) corresponds to an assignment of X.

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 ≠ a is propagated in each unexplored branch emanating after selecting X. For example, when X = a₁ is refuted, X ≠ a₁ 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 ≠ a₃ is removed after propagating
Algorithm 1 MAC_{LK} (P, Y) 

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

Algorithm 2 MAC_{LB}(P, Y) 

Require: \( P \) : input CSP \((X, C, D)\); \( Y \) : current variable 
1: if \( X = \emptyset \) then solution found and stop search
2: if \( Y = null \) then select and remove any variable \( X \) from \( X' \)
3: else \( X \leftarrow Y \)
4: \( V \leftarrow \emptyset \), \( D' \leftarrow D \)
5: select and remove any value \( v \) from \( \mathcal{D}(X) \)
6: \( P \leftarrow \mathcal{AC}(\mathcal{P}) \)
7: if \( \mathcal{P} \neq \emptyset \) then 
8: if \( |\mathcal{D}(X)| = 1 \) then MAC_{LK}(\mathcal{P},null)
9: else MAC_{LK}(\mathcal{P},X)
10: \( D \leftarrow D' \), \( \mathcal{D}(X) \leftarrow \{v\} \), \( P \leftarrow \mathcal{AC}(\mathcal{P}) \)
11: if \( \mathcal{P} \neq \emptyset \) then MAC_{LK}(\mathcal{P},null)
12: \( D \leftarrow D' \)
are already tried as assignments to the current variable, e.g., in Figure 5(a) \( X \neq a_1 \land X \neq a_2 \) 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 MAC_{LS}(P, Y)**

```
Require: \( P \): input CSP \((X, C, D); Y \): current variable
1: if \( X = \emptyset \) then solution found and stop search
2: if \( Y = \text{null} \) then select and remove any variable \( X \) from \( X \)
3: else \( X \leftarrow Y \)
4: \( V \leftarrow \emptyset; D' \leftarrow D \)
5: while \( P \neq \perp \land |V| < |D(X)| \) do
6: \( V \leftarrow V \cup \{v\}; P \leftarrow \text{ac}(P) \)
7: \( X \leftarrow X \setminus \{v\} \)
8: if \( |D(X)| = 1 \) then MAC_{LS}(\( P, \text{null} \)) else MAC_{LS}(\( P, X \))
9: \( D \leftarrow D' ; D(X) \leftarrow V; P \leftarrow \text{ac}(P) \)
10: if \( P \neq \perp \) then
11: \( \sum_{X \in E} \text{unique}(X, Y, a) = |D(X)| |\{a \in C_{XY} : a \neq b \}| \) be
12: MAC_{LS}(\( P, \text{null} \)) else MAC_{LS}(\( P, X \))
```

**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.

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 details in 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 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.

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

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.

<table>
<thead>
<tr>
<th>Problem class</th>
<th>number of instances</th>
<th>k-way</th>
<th>binary</th>
<th>split</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>eager</td>
<td>lazy</td>
<td>oracle</td>
</tr>
<tr>
<td>h1</td>
<td>20</td>
<td>1460.02</td>
<td>57.54</td>
<td>56.92</td>
</tr>
<tr>
<td>h2</td>
<td>20</td>
<td>3881.81</td>
<td>335.16</td>
<td>335.16</td>
</tr>
<tr>
<td>qcp-25 (easy)</td>
<td>20</td>
<td>2992.42</td>
<td>512.26</td>
<td>2498.71</td>
</tr>
<tr>
<td>qcp-25 (hard)</td>
<td>11</td>
<td>959.38</td>
<td>872.64</td>
<td>193.33</td>
</tr>
<tr>
<td>qcp-20 (easy)</td>
<td>5</td>
<td>3843.81</td>
<td>934.81</td>
<td>9235.24</td>
</tr>
<tr>
<td>qcp-20 (hard)</td>
<td>10</td>
<td>3872.20</td>
<td>211.01</td>
<td>1438.88</td>
</tr>
<tr>
<td>rllpModScens (unsat)</td>
<td>5</td>
<td>2.03</td>
<td>1.86</td>
<td>1.82</td>
</tr>
<tr>
<td>rllpModScens (sat)</td>
<td>5</td>
<td>0.25</td>
<td>0.29</td>
<td>0.25</td>
</tr>
<tr>
<td>rllpModKrgs (unsat)</td>
<td>6</td>
<td>0.36</td>
<td>3.92</td>
<td>0.36</td>
</tr>
<tr>
<td>rllpModKrgs (sat)</td>
<td>6</td>
<td>1.47</td>
<td>6.06</td>
<td>0.36</td>
</tr>
<tr>
<td>frb40-19</td>
<td>5</td>
<td>389.83</td>
<td>311.14</td>
<td>36.48</td>
</tr>
<tr>
<td>frb45-21</td>
<td>5</td>
<td>254.42</td>
<td>273.50</td>
<td>206.08</td>
</tr>
<tr>
<td>frb50-23</td>
<td>5</td>
<td>8550.87</td>
<td>9499.98</td>
<td>8375.69</td>
</tr>
<tr>
<td>qsa</td>
<td>3</td>
<td>5062.56</td>
<td>4858.12</td>
<td>4858.12</td>
</tr>
</tbody>
</table>

**Note:** The table shows the performance of the single best branching scheme for each problem class. The combined column shows the performance of the Oracle, which selects the best of all six branching schemes. The overall table shows the cumulative time for all six branching schemes. The eager and lazy branching schemes are combined into a single portfolio. However, there is a dramatic difference between the eager and lazy branching schemes. 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 scheme for all instances and choosing the same one. 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 scheme for all instances and choosing the same one.
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


