The Thirteenth International Symposium
on Combinatorial Search (SoCS 2020)

Decision Tree Learning-Inspired Dynamic Variable Ordering for the Weighted CSP

Hong Xu,* **Kexuan Sun, Sven Koenig, T. K. Satish Kumar**
University of Southern California, Los Angeles, CA 90089, United States of America
{hongx, kexuansu, skoenig}@usc.edu, tkskwork@gmail.com

Abstract

The *weighted constraint satisfaction problem (WCSP)* is a powerful mathematical framework for combinatorial optimization. The branch-and-bound search paradigm is very successful in solving the WCSP but critically depends on the ordering in which variables are instantiated. In this paper, we introduce a new framework for dynamic variable ordering for solving the WCSP. This framework is inspired by regression decision tree learning. Variables are ordered dynamically based on samples of random assignments of values to variables as well as their corresponding total weights. Within this framework, we propose four variable ordering heuristics (**sdr**, **inv-sdr**, **rr** and **inv-rr**). We compare them with many state-of-the-art dynamic variable ordering heuristics, and show that **sdr** and **rr** outperform them on many real-world and random benchmark instances.

Introduction

The *weighted constraint satisfaction problem (WCSP)* is a combinatorial optimization problem. It is a generalization of the constraint satisfaction problem (CSP) in which the constraints are no longer "hard." Instead, each tuple in a constraint—i.e., an assignment of values to all variables in that constraint—is associated with a non-negative weight (sometimes referred to as "cost"). The goal is to find a complete assignment of values to all variables from their respective domains such that the total weight is minimized (Bistarelli et al. 1999), called an optimal solution.

Formally, the WCSP is defined by a triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X} = \{X_1, X_2, \ldots, X_N\}$ is a set of N *variables*, $\mathcal{D} = \{\mathcal{D}(X_1), \mathcal{D}(X_2), \ldots, \mathcal{D}(X_N)\}$ is a set of N *domains* with discrete values, and $\mathcal{C} = \{C_1, C_2, \ldots, C_M\}$ is a set of M weighted *constraints*. Each variable $X_i \in \mathcal{X}$ can be assigned a value in its associated domain $\mathcal{D}(X_i) \in \mathcal{D}$. Each constraint $C_i \in \mathcal{C}$ is defined over a certain subset of the variables $S(C_i) \subseteq \mathcal{X}$, called the scope of C_i. C_i associates a non-negative weight with each possible assignment of values to the variables in $S(C_i)$. The goal is to find a complete assignment of values to all variables in $\mathcal{X}$ from their respective domains that minimizes the sum of the weights specified by each constraint in $\mathcal{C}$ (Bistarelli et al. 1999). This combinatorial task can equivalently be characterized by having to compute

$$\arg\min_{a \in A(\mathcal{X})} \left(E(a) \equiv \sum_{C_i \in \mathcal{C}} E_{C_i}(a|S(C_i)) \right), \qquad (1)$$

where $A(\mathcal{X})$ represents the set of all $|\mathcal{D}(X_1)| \times |\mathcal{D}(X_2)| \times \ldots \times |\mathcal{D}(X_N)|$ complete assignments to all variables in $\mathcal{X}$, $a|S(C_i)$ represents the projection of a complete assignment a onto the subset of variables in $S(C_i)$, and $E_{C_i}(\cdot)$ is a function that maps each $a|S(C_i)$ to its associated weight in C_i.

The WCSP can be used to model a wide range of useful combinatorial problems arising in a large number of real-world application domains. For example, in artificial intelligence, it can be used to model user preferences (Boutilier et al. 2004) and combinatorial auctions. In bioinformatics, it can be used to locate RNA motifs (Zytnicki, Gaspin, and Schiex 2008). In statistical physics, the energy minimization problem on the Potts model is equivalent to that on its corresponding pairwise Markov random field (Yedidia, Freeman, and Weiss 2003), which in turn can be modeled as the WCSP. In computer vision, it can be used for image restoration and panoramic image stitching (Boykov, Veksler, and Zabih 2001; Kolmogorov 2005).

While there exist other techniques such as those based on integer linear programming (Xu, Koenig, and Kumar 2017), satisfiability modulo theories (Bofill et al. 2014), and message passing (Kolmogorov 2006; Xu, Kumar, and Koenig 2017), branch-and-bound search is the cornerstone of many state-of-the-art modern WCSP solvers, such as `toulbar2` (Hurley et al. 2016) and `aolibWCSP` (Marinescu and Dechter 2007). In these search-based solvers, dynamic variable ordering (DVO) is an important ingredient, and how to determine the orderings is a critical question. Indeed, it has been shown that, for a given WCSP instance, different DVO heuristics can lead to dramatically different search efficiencies for various branch-and-bound search techniques, such as those based on local consistencies (Heras and Larrosa 2006; de Givry et al. 2005; Cooper et al. 2010), AND/OR search spaces (Marinescu and Dechter 2006), and dead-end elimination (de Givry, Prestwich, and O'Sullivan 2013). Therefore, studies on DVO heuristics are necessary.

*Now at International Business Machines (IBM) Corp.

In this paper, we derive DVO heuristics from recent advances in machine learning. In particular, we use regression decision tree learning (Breiman et al. 1984), an important machine learning algorithm. A decision tree is a tree where each leaf node is a label and each internal node, called a *decision node*, consists of different values of a feature that lead to different branches below that node. A regression decision tree has label values that are real numbers, while the values of the features can still be discrete. A regression decision tree learning algorithm takes some training samples as its input, where each training sample consists of the values of some features and a label. The goal of this algorithm is to build a decision tree such that, given test samples in which the labels are missing, the prediction of these labels following the decision tree is accurate.

A branch-and-bound search tree for a WCSP instance is quite similar to a regression decision tree: A variable is analogous to a feature, and a total weight is analogous to a label. Each internal node in a search tree assigns different values to a single variable leading to different branches below it. Each leaf node is associated with a total weight corresponding to a complete assignment, indicated by the path from the root node to it. Due to these similarities, introducing techniques from regression decision tree learning can be beneficial to branch-and-bound search. However, despite the existence of dedicated research on DVO heuristics (Heras and Larrosa 2006), to the best of our knowledge, none of it has ever mentioned techniques originating from regression decision tree learning.

In this paper, we introduce a new framework of DVO heuristics for the WCSP inspired by regression decision tree learning. This framework orders variables dynamically based on samples of random assignments of values to variables and their total weights. Within this framework, we propose four variable ordering heuristics (**sdr**, **inv-sdr**, **rr** and **inv-rr**). We compare them with many state-of-the-art DVO heuristics, and show that **sdr** and **rr** outperform them on many real-world and random benchmark instances.

Solving the WCSP with Branch-and-Bound Search

The WCSP can be solved with branch-and-bound search, which explores a search tree with each node representing an assignment of values to variables (Larrosa and Schiex 2004). In a search tree, internal nodes represent partial assignments, whereas leaf nodes represent complete assignments. At any point in time during the search, the currently known best solution $a^\dagger$, which we refer to as the *currently best solution*, and its total weight $w^\dagger$ are maintained. At each node, the search algorithm computes the total weight w_a corresponding to the assignment of that node. If $w_a \geq w^\dagger$, the subtree below this node is pruned. The details of this algorithm are provided in Algorithm 1.

The procedure of enforcing local consistencies can be integrated with branch-and-bound search (Larrosa and Schiex 2004). This is done via associating each node in the search tree with a smaller WCSP instance, which we refer to as a *WCSP subinstance*. A WCSP subinstance $P' = \langle \mathcal{X}' = \mathcal{X} \setminus \{X\}, \mathcal{D}', \mathcal{C}' \rangle$ of a WCSP instance $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ with respect to an assignment $X = x$ to a single variable $X \in \mathcal{X}$ is a WCSP instance in which the total weight of any assignment a of values to variables in $\mathcal{X}'$ plus $E_{C_X}(\{X = x\})$ equals that of $a \cup \{X = x\}$ in P, i.e.,

$$\forall a \in A(\mathcal{X}') : \sum_{C' \in \mathcal{C}'} E_{C'}(a|S(C')) + E_{C_X}(\{X = x\}) = \sum_{C \in \mathcal{C}} E_C(a \cup \{X = x\}|S(C)), \quad (2)$$

where C_X is the unary constraint whose scope is $\{X\}$. For each node, its associated WCSP subinstance can be constructed from the WCSP subinstance associated with its parent node by reducing binary constraints that involve the newly assigned variable to unary constraints on other variables. The details of this procedure are provided in Algorithm 2. The construction of WCSP subinstances enables the search algorithm to enforce local consistency on the WCSP instance associated with each node in the search tree before proceeding to the next node.

Integrating local consistency enforcement with branch-and-bound search yields multiple benefits. For example, it may detect global inconsistency before the search algorithm reaches a leaf node and therefore reduces the search space. Even if global inconsistency is not detected, it may reduce the domain sizes of variables, which also helps to reduce the search space. Algorithm 3 shows the basic framework of local consistency enforcement. In this paper, since our focus is on DVO heuristics, we only enforce weighted arc consistency (WAC) using the WAC-3 algorithm (Larrosa and Schiex 2004). The reason for our choice is that WAC is a fundamental type of local consistency for the WCSP, and using it as a common local consistency allows us to focus on the difference between DVO heuristics.

Known DVO Heuristics

In this section, we review different DVO heuristics from the literature. Intuitively, they all try to first instantiate the variables that are the most constrained, as defined by each heuristic's own measurement.

Domain Size and Degree-Based Heuristics

We consider the following well-known DVO heuristics that are based on the domain sizes and *degrees of variables*. The degree of a variable is defined as the number of constraints whose scopes include the variable.

- *Domain size* (**dom**) (Heras and Larrosa 2006): Choose the variable with the smallest domain size after enforcing local consistency at each search node.
- *Degree* (**deg**) (Heras and Larrosa 2006): Choose the variable with the largest degree after enforcing local consistency at each search node.
- *Weighted degree* (**wdeg**) (Boussemart et al. 2004): Let $wdeg(C)$ be the "wdeg"[1] of each constraint $C \in \mathcal{C}$, and

[1]This is referred to as "weight" in (Boussemart et al. 2004), but we refer to it as "wdeg" to avoid confusion with the WCSP terminology.

Algorithm 1: Solve the WCSP using branch-and-bound search.

```
1  Function SolveWCSP(P)
       Input: P: A WCSP instance.
       Output: The optimal solution of P and its total weight.
2      return BranchAndBound(P, ∅, 0, ∅, +∞);
3  Function BranchAndBound(P = ⟨𝒳, 𝒟, 𝒞⟩, a, w_a, a†, w†)
       Input: P = ⟨𝒳, 𝒟, 𝒞⟩: A WCSP instance.
       Input: a: A partial or complete assignment of values to variables.
       Input: w_a: The total weight associated with a.
       Input: a†: The currently best solution.
       Input: w†: The weight of the currently best solution.
       Output: Updated currently best solution and its total weight.
4      if 𝒳 = ∅ then
5          if w_a < w† then
6              return a, w_a;
7      else
8          (P' = ⟨𝒳', 𝒟', 𝒞'⟩), global_consist := EnforceLocalConsistency(P, w† − w_a);
9          if ¬global_consist then
10             return a†, w†;
11         X := ChooseVariable(P');
12         D := OrderDomain(X, P');
13         foreach x ∈ D do
14             a' := a ∪ {X = x};
15             w_a' := w_a + E_{C'_X}({X = x});
16             P'' := ConstructWCSPSubInstance(X, x, P');
17             a†, w† := BranchAndBound(P'', a', w_a', a†, w†);
18     return a†, w†;
```

Algorithm 2: Construct a WCSP subinstance.

```
1  Function ConstructWCSPSubInstance(X, x, P = ⟨𝒳, 𝒟, 𝒞⟩)
       Input: X, x: The variable and a value in its domain (X = x) to construct a WCSP instance with respect to.
       Input: P = ⟨𝒳, 𝒟, 𝒞⟩: The WCSP instance to construct a subinstance of.
       Output: A WCSP subinstance of P with respect to X = x.
2      P' := ⟨𝒳' = 𝒳, 𝒟' = 𝒟, 𝒞' = 𝒞 \ {C_X}⟩;
3      foreach Y ∈ {Y ∈ 𝒳' | C_XY ∈ 𝒞'} do
           // C_XY is the constraint whose scope is {X, Y}.
4          foreach y ∈ 𝒟'(Y) do
5              E_{C_Y}({Y = y}) := E_{C_Y}({Y = y}) + E_{C_XY}({X = x, Y = y});
6          𝒞' := 𝒞' \ {C_XY};
7      𝒳' := 𝒳' \ {X};
8      return P';
```

Algorithm 3: Enforce local consistency on a WCSP instance P.

```
1  Function EnforceLocalConsistency(P, w)
       Input: P: The WCSP instance to enforce local consistency on.
       Input: w: An upper bound on the total weight of an optimal solution of P.
       Output: Whether global inconsistency is detected.
2      Enforce local consistency on P with w as the upper bound (we denote the output WCSP instance as P');
3      if global inconsistency is detected then
4          return P', FALSE;
5      else
6          return P', TRUE;
```

$wdeg(X) \equiv \sum_{C \in \{C \mid X \in S(C)\}} wdeg(C)$ be the wdeg of each variable $X \in \mathcal{X}$. For each constraint $C \in \mathcal{C}$, initialize $wdeg(C)$ to one. During the enforcement of local consistency (as per Algorithm 1), every time the domain of a variable X becomes $\emptyset$, increase the last visited constraint's wdeg by 1. Choose the variable X with the largest wdeg.

- *Domain size/weighted degree* (**dom/wdeg**) (Boussemart et al. 2004): Choose the variable with the smallest domain size/wdeg ratio.

Cost-Based Heuristics

Cost-based heuristics select variables based on costs. One approach (**suc**) (Heras and Larrosa 2006) is to always choose the variable X with the smallest sum of unary costs $\sum_{x \in \mathcal{D}(X)} E_{C_X}(x)$.

Activity-Based Heuristics

Activity-based heuristics (**abs**) (Michel and Van Hentenryck 2012) select the variable whose domain was most often reduced when enforcing local consistency. Let $activity(X)$ be the activity of a variable $X \in \mathcal{X}$ and initialize it to $|\mathcal{D}(X)|$ before search. When enforcing local consistency, if the domain of a variable X is reduced, $activity(X) := \lambda \cdot activity(X) + 1$, where $0 \leq \lambda \leq 1$ is a decay parameter that favors recent activities. Choose a variable with the smallest $|\mathcal{D}(X)|/activity(X)$. At a search node k where $X = x$ is instantiated, the activity $activity_k(X = x)$ is defined as the number of variables that have their domain sizes reduced when enforcing local consistency. The activity $activity(X = x)$ of $\{X = x\}$ is updated us-

Algorithm 4: Sampling.

Function `Sample(`P, N_s`)`
  Input: $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$: The given WCSP instance.
  Input: N_s: Number of samples.
  Output: N_s samples.
  $J \coloneqq \emptyset$;
  for $i \leftarrow 1$ **to** N_s **do**
    $(P' = \langle \mathcal{X}', \mathcal{D}', \mathcal{C}' \rangle) \coloneqq P$;
    $J_i \coloneqq$ an empty associative array of size $|\mathcal{X}|$;
    while $\mathcal{X}' \neq \emptyset$ **do**
      Randomly select a variable $X \in \mathcal{X}'$;
      Select a value $x \in \mathcal{D}'(X)$ with probability $\propto \frac{1}{E_{C'_X}(x)}$;
      $J_i[X] \coloneqq x$;
      $P' \coloneqq$ `ConstructWCSPSubInstance(`X, x, P'`)`;
    $J \coloneqq J \cup \{\langle J_i, E(J_i) \rangle\}$;
  return J;

ing $activity(X = x) \coloneqq (activity(X = x) \cdot (\alpha - 1) + activity_k(X = x))/\alpha$, where $\alpha \geq 1$ is a parameter. When ordering domain values, choose a value with the largest activity. All activities are initialized by sampling paths according to certain distributions in the search tree.

Impact-Based Heuristics

Impact-based heuristics (**ibs**) (Refalo 2004) select the variable with the highest expected impact. The impact $impact(X = x)$ of an assignment of a value x to a variable X indicates the potential reduction of the search space after instantiating X. Similarly to the activity of an assignment in **abs**, it is updated using $impact(X = x) \coloneqq (impact(X = x) \cdot (\alpha - 1) + impact_k(X = x))/\alpha$. At a search node k, the impact $impact_k(X)$ of a variable X is defined as the sum of all impacts of assignments of values to X minus its domain size at that search node, i.e., $impact_k(X) = \sum_{x \in \mathcal{D}(X)} impact(X = x) - |\mathcal{D}(X)|$. Choose a variable with the largest impact and a value with the smallest impact.

Regression Decision Tree Learning-Inspired Heuristics

We create a new framework of DVO heuristics based on the idea of learning from randomly sampled assignments, which is inspired by regression decision tree learning algorithms. In this framework, the DVO heuristics are based on some measurements of samples of randomly selected assignments and their corresponding total weights. (The details of the sampling method are provided in Algorithm 4.) The details of the DVO heuristics to replace the function `ChooseVariable()` in Algorithm 1 are provided in Algorithm 5. It chooses a variable with the largest measurement. Here, we consider the following measurements.

Algorithm 5: Choose a variable based on sampling.

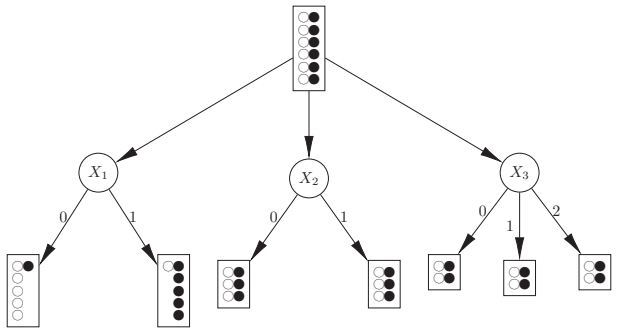
Function `ChooseVariableSampling(`P, N_s`)`
  Input: $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$: A WCSP instance.
  Input: N_s: Number of samples.
  Output: The next variable to assign a value to.
  $J \coloneqq$ `Sample(`P, N_s`)`;
  foreach $X \in \mathcal{X}$ **do**
    Compute $Measurement(X, J)$ of X in J;
  return $\arg\max_{X \in \mathcal{X}} Measurement(X, J)$;

X_1 \ X_2	0	1
0	1	2
1	102	3

(a) Constraint C_1

X_2 \ X_3	0	1	2
0	1	2	3
1	102	3	101

(b) Constraint C_2



(c) Search tree

Figure 1: Illustrates the intuition behind **sdr**. The example WCSP instance has three variables X_1, X_2 and X_3. It has two constraints C_1 and C_2, illustrated in (a) and (b), respectively. (c) illustrates the first step of building the search tree, i.e., choosing the first variable and its domain value. Each rectangle represents a search node. The circles in each rectangle illustrate the distribution of total weights of all assignments of values to variables in that search node. Each empty/filled circle in these rectangles represents a low-cost (with a total weight less than 10)/high-cost (with a total weight higher than 100) assignment. Choosing X_1, which has the largest SDR, is the best option among the three, since the branch of $X_1 = 0$ offers the highest probability of reaching a low-weight leaf, which increases the probability of pruning other parts of the search tree later during the search.

- *Standard deviation reduction (SDR)* (**sdr**): The measurement is the SDR of the samples J with respect to each variable X. It is defined as

$$\mathrm{SDR}(X, J) = \mathrm{SD}(J) - \frac{1}{|J|} \sum_{x \in \mathcal{D}(X)} |J[X = x]| \cdot \mathrm{SD}(J[X = x]), \tag{3}$$

  where $\mathrm{SD}(J)$ is the standard deviation of all total weights in J, $|J|$ is the number of samples in J, and $J[X = x]$ is

the subset of J in which $X = x$. In the actual implementation of this algorithm, the computation of $\mathrm{SD}(J)$ is not required. However, we use it to relate our measurement to the concept of the SDR. This measurement is inspired by the regression decision tree learning algorithm (Breiman et al. 1984). Figure 1 illustrates the intuition behind **sdr**. Algorithm 4 samples with a bias towards smaller weights. The reason for having this bias is that search nodes corresponding to assignments with high weights are in general less likely to be reached, and therefore samples corresponding to these assignments are less useful.

- *Inverse SDR* (**inv-sdr**): Similar to **sdr**, but negated SDRs are used as the measurements instead of SDRs. The intuition behind this approach is that this can lead to more pruning within the subtree due to a larger variance of the total weights in the subtree.
- *Range reduction (RR)* (**rr**): The measurement is the RR of the samples J with respect to each variable X. The definition of RR is similar to that of SDR, with all standard deviations in Equation (3) replaced by ranges, i.e., the maximum total weight minus the minimum total weight in the samples. It is a variant of **sdr** and shares similar intuition with it.
- *Inverse RR* (**inv-rr**): Similar to **rr**, but negated RRs are used as the measurements instead of RRs. It is a variant of **inv-sdr**.

Under this framework, `OrderDomain()` orders each value x in the domain of a variable X in ascending order with respect to the average of the weights over all samples with x assigned to X. The intuition is that, by first focusing the search along a branch that is more likely to produce an assignment with a small total weight, the more likely it is to prune other regions of the search space. We also note that, in practice, the total weights of the samples *per se* also provide bounds that could be useful for pruning the search tree.

Discussion: The Number of Samples

In Algorithm 5, the number of samples N_s is a parameter. However, its choice poses a dilemma: If N_s is too small, then Algorithm 5 may not have enough samples to provide useful guidance for the search; if N_s is too large, then it may take too much time to compute the DVO heuristic at a search node and thus the entire procedure is self-defeating.

Here, to strike a balance, we propose the number of samples at each search node k to be the total number of domain values of all variables in the WCSP instance $P_k = \langle \mathcal{X}_k, \mathcal{D}_k, \mathcal{C}_k \rangle$ of k, i.e., $N_s = \sum_{X \in \mathcal{X}_k} |\mathcal{D}_k(X)|$. Although the true efficiency must be proven by experiments, our rationale for the proposal is as follows.

(a) The number of samples is not too large:

- The time complexity of the search algorithm is similar to that of those search algorithms that spend a constant time at each search node. In our search framework, at a search node k of depth d_k in the search tree, the time complexity to order the variables is $\mathcal{O}\left(\hat{D} \cdot (|\mathcal{X}| - d_k)^3\right)$, where $\hat{D} = \max_{X \in \mathcal{X}} |\mathcal{D}(X)|$ is the maximum domain size of all variables. Each sample requires $\mathcal{O}\left((|\mathcal{X}| - d_k)^2\right)$ time to generate and the number of samples is $\mathcal{O}\left(\hat{D} \cdot (|\mathcal{X}| - d_k)\right)$. (In practice, if samples in previous search nodes are reused, this complexity can be further reduced.) Therefore, the time complexity of our DVO heuristics at all search nodes is

$$\mathcal{O}\left(\sum_{d_k=0}^{|\mathcal{X}|-1} \left[\hat{D} \cdot (|\mathcal{X}| - d_k)^3 \cdot \hat{D}^{d_k}\right]\right). \quad (4)$$

  For either a bounded $\hat{D}$ or $|\mathcal{X}|$, this can be simplified to $\mathcal{O}(\hat{D}^{|\mathcal{X}|})$, which is the same time complexity as that of DVO heuristics that spend constant time at each search node.
- The probability of producing two identical samples at a search node is low. We assume that all assignments of values to variables are sampled uniformly at random. At search node k, the probability P_k that at least two samples are identical is

$$P_k = 1 - \frac{\prod_{i=0}^{\sum_{X \in \mathcal{X}_k} |\mathcal{D}_k(X)|-1} [|\mathcal{A}(\mathcal{X}_k)| - i]}{|\mathcal{A}(\mathcal{X}_k)|^{\sum_{X \in \mathcal{X}_k} |\mathcal{D}_k(X)|}}. \quad (5)$$

  If the number of variables approaches infinity and all domain sizes remain finite, P_k approaches 0.

(b) The number of samples is not too small:

- In our algorithm, at each search node, it is best to use the actual standard deviation of all assignments of values to uninstantiated variables when computing SDR. In reality, as shown in Algorithm 5, we can only use the standard deviation of samples to estimate the actual standard deviation. Such an estimate, in statistical terminology, is called a *point estimate*. The quantity used to do the estimate, the standard deviation of the samples in our case, is referred to as the *estimator*. The quality of an estimator can be assessed using its standard deviation, related to the *efficiency* of the estimator—the lower the standard deviation, the higher the efficiency is.

  We now argue that the efficiency of our estimator decreases slowly as the problem size increases. We assume that the weights of all constraints independently follow a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$. Then, at search node k, the total weight of random assignments follows $\mathcal{N}(|\mathcal{C}_k| \cdot \mu, |\mathcal{C}_k| \cdot \sigma^2)$. By definition, the variance $\mathrm{SD}(J)^2$ of N_s samples J is

$$\mathrm{SD}(J)^2 = \left(\sum_{\langle J_i, E(J_i)\rangle \in J} \left(E(J_i) - \bar{E}(J)\right)^2\right) / N_s, \quad (6)$$

  where $\bar{E}(J) = \left[\sum_{\langle J_i, E(J_i)\rangle \in J} E(J_i)\right] / N_s$. With $|\mathcal{C}_k| \cdot \mu$ being the estimate of $\bar{E}(J)$, $N_s \cdot \mathrm{SD}(J)^2 / \left(|\mathcal{C}_k| \cdot \sigma^2\right)$, seen as a random variable, is the sum of the squares of N_s independent standard normal random variables, and thus follows a χ^2-distribution with a degree of freedom N_s and

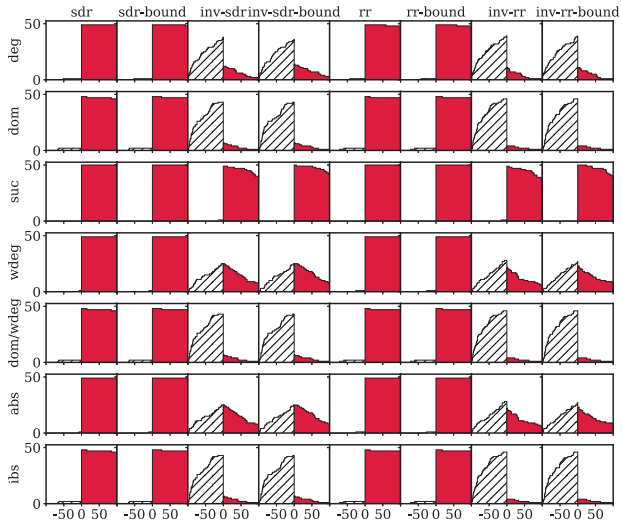


Figure 2: Compares the numbers of visited search nodes between our DVO heuristics and the existing DVO heuristics on 50 random benchmark instances with $n = 20$. Each subplot compares one of our DVO heuristics and one existing DVO heuristic. The x-axes indicate the relative numbers of visited search nodes, i.e., (number of visited search nodes of the existing DVO heuristic - number of visited search nodes of our DVO heuristic)/number of visited search nodes of our DVO heuristic. The y-axes indicate the numbers of benchmark instances. Curves in the left halves (with x-values smaller than zero) are cumulative histograms, and curves in the right halves (with x-values greater than zero) are reverse cumulative histograms. Larger areas under the curves in the right halves and smaller areas under the curves in the left halves indicate smaller relative numbers of visited search nodes (i.e., higher search node efficiency) of our DVO heuristics.

thus has a standard deviation of $\sqrt{2N_s}$. This implies that $\mathrm{SD}(J)^2$ itself (seen as a random variable) has a standard deviation of $|\mathcal{C}_k| \cdot \sigma^2\sqrt{2/N_s}$. In our algorithm, N_s increases linearly with the number of variables (assuming that $\hat{D}$ is bounded). This implies that the standard deviation of $\mathrm{SD}(J)^2$ increases sublinearly with respect to $|\mathcal{C}_k|$. The increment is even smaller if we consider $\mathrm{SD}(J)$ instead of $\mathrm{SD}(J)^2$.

Experimental Evaluation

In this section, we present an experimental evaluation of **sdr**, **inv-sdr**, **rr**, and **inv-rr**. We compare them with **dom**, **deg**, **wdeg**, **dom/wdeg**, **suc**, **abs**, and **ibs**. Since the total weights of the samples also provide bounds that can be useful for pruning the search tree, we also experimented with variants of **sdr**, **inv-sdr**, **rr**, and **inv-rr**, in which the smallest total weights of existing samples are also used as bounds at each search node during the search. We refer to these variants as **sdr-bound**, **inv-sdr-bound**, **rr-bound**, and **inv-rr-bound**, respectively. In order to make our implementation

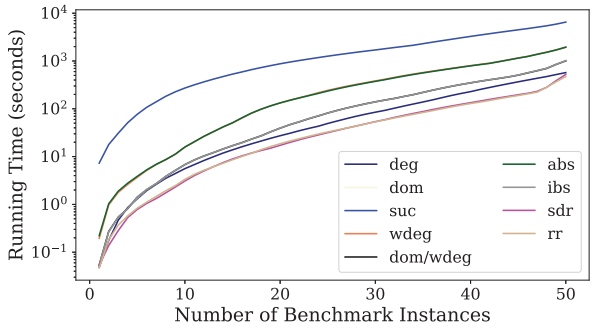


Figure 3: Shows the runtime distribution of all DVO heuristics for $n = 20$. Each curve represents a DVO heuristic and shows the runtimes for various numbers of benchmark instances for this DVO heuristic. The runtimes of all 50 random benchmark instances are sorted for each DVO heuristic in ascending order. This allows us to consider the best cases for each number of benchmark instances for each DVO heuristic. The y-axis is in logarithmic scale, which means that a slight gap between two curves may represent orders of magnitude of difference. (This figure should be viewed in color.)

more efficient (in practice), each time we need to sample (Algorithm 4) in a regression decision tree-inspired heuristic, we cache the samples and reuse as many previous applicable samples as possible.

In our experiments, all heuristics are implemented in C++, compiled by GCC 6.3.0 with the "-O3" option. All real-world benchmark instances are run on a GNU/Linux workstation (Ubuntu 16.04) with Intel Xeon Processor E5-2676 v3 (30MB Cache, 2.40GHz) and 990MB RAM. All random benchmark instances were run on a GNU/Linux workstation (Debian 9) with Intel Xeon Processor E3-1240 v3 (8MB Cache, 3.4GHz) and 16GB RAM.

The real-world benchmark instances are from (Hurley et al. 2016)[2]. These benchmark instances include the Probabilistic Inference Challenge 2011, the Computer Vision and Pattern Recognition OpenGM2 benchmark, the Weighted Partial MaxSAT Evaluation 2013, the MaxCSP 2008 Competition, the MiniZinc Challenge 2012 & 2013 and the CFLib (a library of cost function networks). Since the purpose of the experiments is to compare DVO heuristics, other parts of the branch-and-bound search algorithm are not algorithmically tailored to them to avoid introducing unnecessary bias in the experiments. Although this also limits the sizes of benchmark instances on which we could perform experiments within a reasonable amount of runtime, we still select 6 benchmark instances based on the number of variables (no more than 25) and domain sizes (no more than 6). We set the runtime limit for each benchmark instance to 48 hours. Table 1 lists all such benchmark instances and shows the experimental results on them. With the exception

[2]http://genoweb.toulouse.inra.fr/~degivry/evalgm/

Table 1: Comparison results for real-world benchmark instances (nodes/time)

	Name	ff1	j4*	l4*	q5*	q3*	q4*
Instance	$\lvert\mathcal{X}\rvert$	2	28	8	25	25	25
	$\lvert\mathcal{C}\rvert$	3	196	32	185	185	185
	$\hat{D}$	5	2	6	5	3	4
Algorithm	**sdr**	$31/3 \cdot 10^{-4}$s	833/0.27s	101/0.05s	391,065/4042s	-/48h	-/48h
	sdr-bound	$31/3 \cdot 10^{-4}$s	637/1.60s	11/0.04s	6/0.94s	-/48h	-/48h
	inv-sdr	$31/2 \cdot 10^{-4}$s	5491/1.64s	179/0.05s	429,005/4984s	-/48h	-/48h
	inv-sdr-bound	$31/2 \cdot 10^{-4}$s	667/1.80s	8/0.08s	6/0.94s	-/48h	-/48h
	rr	$31/3 \cdot 10^{-4}$s	801/2.16s	109/0.16s	1100/9.95s	-/48h	-/48h
	rr-bound	$31/1 \cdot 10^{-2}$s	665/1.71s	11/0.08s	6/0.97s	-/48h	-/48h
	inv-rr	$31/2 \cdot 10^{-4}$s	5943/11.97s	429/0.29s	14,677/44.78s	-/48h	-/48h
	inv-rr-bound	$31/2 \cdot 10^{-4}$s	659/1.58s	10/0.08s	6/0.94s	-/48h	-/48h
	deg	$31/1 \cdot 10^{-4}$s	3225/1.26s	187/0.04s	27,834,834/48,163s	-/48h	-/48h
	dom	$31/9 \cdot 10^{-5}$s	8623/5.24s	331/0.08s	-/48h	-/48h	-/48h
	suc	$31/9 \cdot 10^{-5}$s	3491/1.72s	606/0.12s	7,718,377/8867s	-/48h	-/48h
	wdeg	$31/9 \cdot 10^{-5}$s	8623/5.37s	203/0.15s	-/48h	-/48h	-/48h
	dom/wdeg	$31/9 \cdot 10^{-5}$s	8623/5.29s	331/0.08s	-/48h	-/48h	-/48h
	abs	$31/2 \cdot 10^{-4}$s	3173/2.73s	404/0.33s	1,814,781/911s	-/48h	-/48h
	ibs	$31/1 \cdot 10^{-4}$s	7045/4.53s	236/0.08s	-/48h	-/48h	-/48h

* The benchmark instance names are johnson8-2-4, langford-2-4, queens-5-5-5, queens-5-5-3, and queens-5-5-4, respectively.

of 2 benchmark instances that could not be solved within the runtime limit, our DVO heuristics mostly outperformed the other DVO heuristics on all other non-trivial benchmark instances. In particular, on queens-5-5-5, the empirically hardest solvable benchmark instance in our experiments, there were orders of magnitude of difference in both the number of visited search nodes and runtime between our DVO heuristics and other heuristics. In addition, using sample results as bound during the search process decreases the runtime of our algorithms.

Due to the lack of relatively small real-world benchmark instances, we also generate additional random benchmark instances. Here, we focus on sparse random benchmark instances. We use the Erdős-Rényi (ER) model (Erdős and Rényi 1959), a basic random graphical model, to generate these benchmark instances. The ER model has two parameters: n (the number of vertices) and p (the probability of connecting each pair of vertices). We create each random benchmark instance by first (a) using the ER model to generate a random graph, and then (b) generating a WCSP benchmark instance from it. In each WCSP benchmark instance, each variable corresponds to a vertex in the random graph and each constraint corresponds to an edge. The variables in each constraint C correspond to the two endpoint vertices of the edge corresponding to C. The weights of constraints are integers chosen uniformly at random between 1 and 100. We set n to values ranging from 12 to 20 and p to 0.1 as parameters of the ER model to generate sparse graphs, which in turn generate sparse benchmark instances.

Figure 2 compares the numbers of visited search nodes between our DVO heuristics and the existing DVO heuristics for $n = 20$. **sdr**, **sdr-bound**, **rr** and **rr-bound** clearly outperform all other heuristics, while **inv-sdr**, **inv-sdr-bound**, **inv-rr**, and **inv-rr-bound** do not perform well. Although the original regression decision tree learning algorithm uses SDR as the main criterion for branching, our results show that, for regression decision tree learning-inspired DVO heuristics, RR can achieve similar numbers of visited search nodes as SDR, but is computationally much less expensive.

We also compare the runtimes of all DVO heuristics. Figure 3 shows the runtime distribution of all DVO heuristics for $n = 20$. The curves of **sdr** and **rr** lie below the other curves, indicating that, within the same amount of runtime, our DVO heuristics solved more benchmark instances than the existing DVO heuristics, which means that our DVO heuristics are more efficient.

Since we could not perform experiments on large benchmark instances, we analyze the trend of the runtime and the number of visited search nodes as the number of variables increases to estimate the efficiency of our DVO heuristics on large instances. Figure 4 demonstrates how the average runtime $\bar{T}$ and the average number of visited search nodes $\bar{K}$ vary with the number of variables $n \in [12, 20]$ for each DVO heuristic. Visually, the computational cost of our DVO heuristics increases less significantly than that of the existing DVO heuristics in terms of both $\bar{T}$ and $\bar{K}$. To analyze this increment quantitatively, we fit our experimental results of each DVO heuristic h using $\log_{10} \bar{T} = a^h_{\bar{T}} n + b^h_{\bar{T}}$ and $\log_{10} \bar{K} = a^h_{\bar{K}} n + b^h_{\bar{K}}$. The smaller $a^h_{\bar{T}}$ and $a^h_{\bar{K}}$ are, the higher the efficiency of DVO heuristic h is expected to be for large instances. As shown in Figure 4, $a^{\mathbf{sdr}}_{\bar{T}}$, $a^{\mathbf{sdr}}_{\bar{K}}$, $a^{\mathbf{rr}}_{\bar{T}}$ and $a^{\mathbf{rr}}_{\bar{K}}$ are smaller than those of the existing DVO heuristics. Since we applied logarithms to $\bar{T}$ and $\bar{K}$ when fitting, small differences in $a^h_{\bar{T}}$ and $a^h_{\bar{K}}$ may represent orders of magnitude of differences in efficiency for large instances. This means that our DVO heuristics are expected to outperform the existing

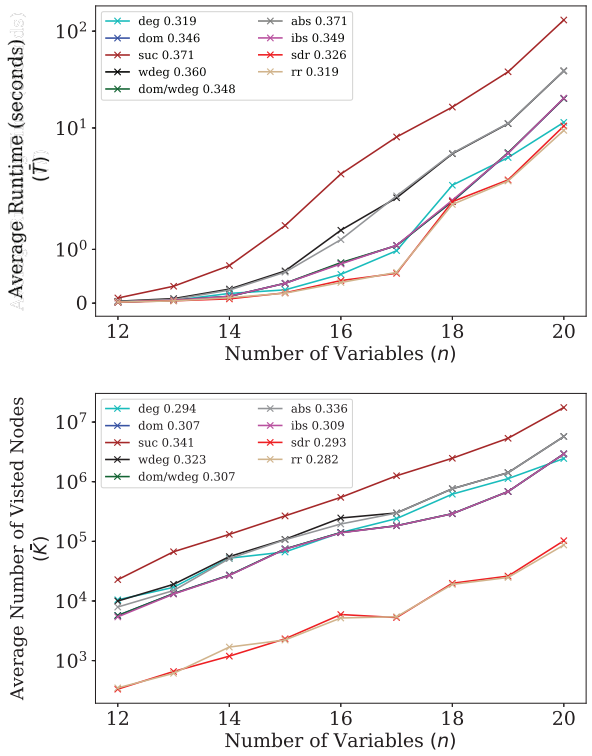


Figure 4: Shows how the average runtime $\bar{T}$ (upper panel) and the average number of visited search nodes $\bar{K}$ (lower panel) vary with the number of variables $n \in [12, 20]$. Curves of different color represent different DVO heuristics. The values of $a^h_{\bar{T}}$ (upper panel) and $a^h_{\bar{K}}$ (lower panel) are shown in the legends. Similar to Figure 3, the y-axes are in logarithmic scale, which means that a slight difference in $a^h_{\bar{T}}$ and $a^h_{\bar{K}}$ may represent orders of magnitutde of difference in the amount that $\bar{T}$ and $\bar{K}$ increase as n increases. (This figure should be viewed in color.)

DVO heuristics significantly for large instances as well. Indeed, as shown in Table 1 before, our experimental results on the relatively large real-world benchmark instance queens-5-5-5 already display huge differences in efficiency between our DVO heuristics and the existing DVO heuristics.

Conclusion and Future Work

In this paper, we introduced a new extensible framework of DVO heuristics for the WCSP inspired by regression decision tree learning. We proposed four DVO heuristics, namely **sdr**, **inv-sdr**, **rr** and **inv-rr**. We discussed details of the intuition behind **sdr** and **rr**. We also carefully chose the number of samples at each search node, and discussed the rationales behind this choice. Finally, we compared them with many other state-of-the-art DVO heuristics, and showed that our DVO heuristics outperformed them on both real-world and random benchmark instances.

Our work has a wide range of potential future extensions. One potential extension is to improve the efficiency in producing samples. Our current approach to producing a sample has a time complexity that is linear in the number of constraints. However, a Markov chain Monte Carlo sampling method such as the Hasting-Metropolis algorithm might significantly improve the efficiency. Another possible extension is to combine our new DVO heuristics with other DVO heuristics. Yet another extension is to adapt them to more specific combinatorial optimization problems such as the weighted MAX-SAT and integer linear programming problems.

Acknowledgments

The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1724392, 1409987, 1817189, 1837779, and 1935712. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies or the U.S. government.

References

Bistarelli, S.; Montanari, U.; Rossi, F.; Schiex, T.; Verfaillie, G.; and Fargier, H. 1999. Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. *Constraints* 4(3):199–240.

Bofill, M.; Palahí, M.; Suy, J.; and Villaret, M. 2014. Solving intensional weighted CSPs by incremental optimization with BDDs. In *the International Conference on Principles and Practice of Constraint Programming*, 207–223.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *the European Conference on Artificial Intelligence*, 146–150.

Boutilier, C.; Brafman, R. I.; Domshlak, C.; Hoos, H. H.; and Poole, D. 2004. CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *Journal of Artifical Intelligence Research* 21:135–191.

Boykov, Y.; Veksler, O.; and Zabih, R. 2001. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23(11):1222–1239.

Breiman, L.; Friedman, J.; Stone, C. J.; and Olshen, R. A. 1984. *Classification and Regression Trees*. Chapman and Hall/CRC.

Cooper, M.; de Givry, S.; Sanchez, M.; Schiex, T.; Zytnicki, M.; and Werner, T. 2010. Soft arc consistency revisited. *Artificial Intelligence* 174(7):449–478.

de Givry, S.; Heras, F.; Zytnicki, M.; and Larrosa, J. 2005. Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *the International Joint Conference on Artificial Intelligence*, 84–89.

de Givry, S.; Prestwich, S. D.; and O'Sullivan, B. 2013. Dead-end elimination for weighted CSP. In *the International Conference on Principles and Practice of Constraint Programming*, 263–272.

Erdős, P., and Rényi, A. 1959. On random graphs I. *Publicationes Mathematicae* 6:290–297.

Heras, F., and Larrosa, J. 2006. Intelligent variable orderings and re-orderings in DAC-based solvers for WCSP. *Journal of Heuristics* 12(4):287–306.

Hurley, B.; O'Sullivan, B.; Allouche, D.; Katsirelos, G.; Schiex, T.; Zytnicki, M.; and de Givry, S. 2016. Multi-language evaluation of exact solvers in graphical model discrete optimization. *Constraints* 21(3):413–434.

Kolmogorov, V. 2005. Primal-dual algorithm for convex Markov random fields. Technical Report MSR-TR-2005-117, Microsoft Research.

Kolmogorov, V. 2006. Convergent tree-reweighted message passing for energy minimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28(10):1568–1583.

Larrosa, J., and Schiex, T. 2004. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence* 159(1):1–26.

Marinescu, R., and Dechter, R. 2006. Dynamic orderings for AND/OR branch-and-bound search in graphical models. In *the European Conference on Artificial Intelligence*, 138–142.

Marinescu, R., and Dechter, R. 2007. Best-first AND/OR search for graphical models. In *the AAAI Conference on Artificial Intelligence*, 1171–1176.

Michel, L., and Van Hentenryck, P. 2012. Activity-based search for black-box constraint programming solvers. In *the International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, 228–243.

Refalo, P. 2004. Impact-based search strategies for constraint programming. In *the International Conference on Principles and Practice of Constraint Programming*, 557–571.

Xu, H.; Koenig, S.; and Kumar, T. K. S. 2017. A constraint composite graph-based ILP encoding of the Boolean weighted CSP. In *the International Conference on Principles and Practice of Constraint Programming*, 630–638.

Xu, H.; Kumar, T. K. S.; and Koenig, S. 2017. The Nemhauser-Trotter reduction and lifted message passing for the weighted CSP. In *the International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, 387–402.

Yedidia, J. S.; Freeman, W. T.; and Weiss, Y. 2003. Understanding belief propagation and its generalizations. *Exploring Artificial Intelligence in the New Millennium* 8:239–269.

Zytnicki, M.; Gaspin, C.; and Schiex, T. 2008. DARN! A weighted constraint solver for RNA motif localization. *Constraints* 13(1):91–109.