IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received June 10, 2018, accepted July 18, 2018, date of publication July 25, 2018, date of current version August 20, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2859618

A Novel Strategy of Combining Variable Ordering Heuristics for Constraint Satisfaction Problems

HONGBO LI[1] AND ZHANSHAN LI[2]
[1]School of Information Science and Technology, Northeast Normal University, Changchun 130117, China
[2]Key Laboratory for Symbol Computation and Knowledge Engineering of National Education Ministry, College of Computer Science and Technology, Jilin University, Changchun 130012, China

Corresponding author: Zhanshan Li (lizs@jlu.edu.cn)

This work was supported in part by the Fundamental Research Funds for the Central Universities under Grant 2412018JC009, in part by the Education Department of Jilin Province under Grant JJKH20170911KJ, in part by the Natural Science Foundation of Jilin Province of China under Grant 20180101043JC, and in part by the Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, under Grant 93K172017K06.

ABSTRACT Variable ordering heuristic plays a central role in solving constraint satisfaction problems. Many heuristics have been proposed and well-studied. In order to take advantage of the fact that many generic variable ordering heuristics work well for different problems, we propose a novel method in this paper, namely *ParetoHeu*, to combine variable ordering heuristics. At each node of the search tree, a set of candidate variables is generated by a new strategy based on Pareto optimality and a variable is selected from the set randomly. The method is easy to be implemented in constraint solvers. The experiments on various benchmark problems show that *ParetoHeu* is more efficient than both the participant heuristics which are popular in constraint solvers. It is also more robust than some classical strategies which have been used to combine variable ordering heuristics.

INDEX TERMS Constraint programming, constraint satisfaction problem, Pareto optimality, variable ordering heuristic.

I. INTRODUCTION

Constraint programming is a powerful paradigm for solving combinatorial search problems. It has been widely applied in Artificial Intelligence, Programming Languages, Databases and Operations Research. Many classic combinatorial problems have been successfully solved with constraint programming, such as Scheduling, Planning, Vehicle Routing, and so on. Constraint satisfaction problem (CSP) is the basis of constraint programming. In the most general form, constraint satisfaction problems are NP-Hard. Variable ordering heuristics are crucial for efficiently solving CSPs. A proper heuristic may solve a hard CSP instance in a second. Many efficient variable ordering heuristics have been proposed and well studied, such as *min-dom* [1], *max-degree* [2], *dom/deg* [3], *dom/wdeg* [4], impact-based search (IBS) [5], count-based search (CBS) [6], activity-based search (ABS) [7] and so on. *ABS*, *IBS*, *dom/wdeg* and *CBS* are the most popular searching strategies in constraint solvers.

Besides these variable ordering heuristics, some strategies of combining multiple heuristics are proposed. The *Brelaz* heuristic [8] combines *min-dom* and *max-degree*. It first selects variables according to *min-dom* and uses *max-degree* to break ties if there exists. The *dom/deg* heuristic is another strategy combining *min-dom* and *max-degree*. It uses domain size over variable degree as its heuristic score. The *dom/deg* heuristic was considered as one of the most efficient variable ordering heuristics. The recent *dom/wtdeg* heuristic adds constraint tightness in *dom/wdeg*. Its heuristic score combines summation of constraint tightness and the weighted degree of *dom/wdeg*. Moreover, hyper-heuristics are high-level heuristics designed to select an appropriate one from a number of low-level heuristics in the context of search. Many hyper-heuristics [9] have been proposed, such as the machine-learning-based methods [10], [11] and evolutionary-computation based methods [12], [13]. Some other methods also exist [14]–[17]. Unfortunately, these methods have not been implemented in the popular constraint solvers which are publicly available [18], [19]. In addition, the technique of algorithm portfolios [20], [21] runs multiple solving procedures simultaneously and it terminates as soon as one of the procedures terminates, so such technique is more efficient than running only one procedure. However, if none of the participant heuristics works on a problem, the portfolio strategy does not work either. Therefore, although the portfolio based methods are efficient, it is still important to develop new single-thread search strategies.

This paper proposes a novel method to combine different variable ordering heuristics, namely *ParetoHeu*. The aim is to generate a more robust single-thread search strategy, which is easy to be implemented in constraint solvers. We consider two participant heuristics in this paper. Most variable ordering heuristics generate a candidate variable set according to some heuristic scores and select one from the candidate set. We use $cand(heu)$ to denote the set of candidate variables selected by a variable ordering heuristic *heu*. The $cand(heu)$ is a set of variables because the heuristic *heu* may find more than one candidate variables which are equivalent under some evaluation criterion, such as they have the same heuristic score. *ParetoHeu* combines two participant heuristics, heu_1 and heu_2. If $cand(heu_1) \cap cand(heu_2)$ is not empty, then we use the intersection as the candidate set; otherwise, we generate a candidate set by a novel strategy combining two heuristics, which is based on Pareto optimality [22].

The participant heuristics considered in this paper are ABS and *dom/wdeg*. The experiments were run on 7 kinds of different benchmark problems. The results show that *ParetoHeu* is more robust than both ABS and *dom/wdeg*. It also outperforms some classic combining strategies. Among all the existing searching strategies we have tested in the experiments, the $A \times D$ strategy is the most efficient one. The average cpu time of *ParetoHeu* is about two thirds of that of the $A \times D$ strategy. Besides, *ParetoHeu* finishes the largest number of runs in given time durations. To our best knowledge, the participants used by the existing combining strategies are simple heuristics, such as *min-dom* and *max-degree*. Therefore, our contributions are as two fold: 1. we propose a novel strategy for combining variable ordering heuristics. 2. we report some experimental results of combining two heuristics which are the most popular searching strategies in constraint solvers.

This paper is organized as follows. Section 2 provides some technical background about CSP and variable ordering heuristics. The *ParetoHeu* is introduced in Section 3. The experimental results and the analysis are in Section 4. Finally, conclusion is in Section 5.

II. BACKGROUND

A constraint satisfaction problem (CSP) P is a triple $P = \langle X, D, C \rangle$, where X is a set of n variables $X = \{x_1, x_2 \ldots x_n\}$, D is a set of domains $D = \{dom(x_1), dom(x_2) \ldots dom(x_n)\}$, where $dom(x_i)$ is a finite set of possible values for variable x_i, and C is a set of e constraints $C = \{c_1, c_2 \ldots c_e\}$. Each constraint c consists of two parts, an ordered set of variables $scp(c) = \{x_{i1}, x_{i2} \ldots x_{ir}\}$ and a subset of the Cartesian product $dom(x_{i1}) \times dom(x_{i2}) \times \ldots \times dom(x_{ir})$ that specifies the allowed combinations of values for the variables $\{x_{i1}, x_{i2} \ldots x_{ir}\}$. A solution to a CSP is an assignment of a value to each variable such that all the constraints are satisfied.

For example, given a CSP, $X = \{x_1, x_2, x_3\}$, $dom(x_1) = \{1, 2, 3\}$, $dom(x_2) = \{1, 2\}$, $dom(x_3) = \{2, 3\}$, $C = \{c_1, c_2\}$ where c_1 is $x_1 > x_2$ and c_2 is $x_1 + x_2 = x_3$. A solution of the CSP is $\{x_1=2, x_2=1, x_3=3\}$.

Solving a CSP P involves either finding a solution of P or determining that P is unsatisfiable. Backtracking search performs a depth-first traversal of a search tree to solve a CSP. In the context of a search tree, each edge is associated with an assignment, a node at level k is associated with a set of k assignments which are attached to the path from the root to this node. During backtracking search, the instantiated variables are called past variables and the uninstantiated are called future variables. The set of all future variables is denoted by *FutVars*. At each node of the search tree, a future variable x_i is selected and a value in $dom(x_i)$ is assigned to x_i. Then a new node is generated and some propagation techniques, such as arc consistency [23] and bound consistency [24], are applied to remove those values that are no longer consistent. Besides these original propagation algorithms, some improved propagation algorithms have also been studied [25]–[29]. A dead-end is reached if the propagation fails, then one or more assignments must be canceled and backtracking occurs.

It is widely accepted that the order in which variables are instantiated is crucial to the efficiency of backtracking search. However, finding an optimal ordering is as difficult as solving a CSP. Thus, the ordering is usually determined by heuristics in practice. Variable ordering heuristics are used at each search tree node to select next future variable. A variable ordering heuristic usually computes a score for each variable and selects the next future variable according to the score. The following subsections recall the two participant variable ordering heuristics considered in this paper.

A. ACTIVITY-BASED SEARCH

ABS [7] considers the most active variables should be instantiated first. Given a CSP $P = \langle X, D, C \rangle$, the backtracking search applies a constraint propagation technique F after each assignment. Some inconsistent values are removed from the corresponding domains due to this propagation. A subset $X' \subseteq X$ records those affected variables whose domains were reduced by this propagation. The activity of x_i, denoted by $A(x_i)$, is updated at each search tree node by the following rules:

$$\begin{aligned} &\forall x_i \in X \;\; s.t. \;\; |dom(x_i)| > 1 : A[x_i] = A[x_i] \times \gamma \\ &\forall x_i \in X' : A[x_i] = A[x_i] + 1 \end{aligned} \tag{1}$$

where $\gamma \in [0, 1]$ is an aging decay parameter. The activity of an assignment at a search tree node is defined as:

$$A[x_i = a] = |X'| \tag{2}$$

which is the number of affected variables when applying F after the assignment. The activity of an assignment also can be estimated by the average of all observed activities of this assignment. The *ABS* heuristic selects the variable x_i with the largest ratio $\frac{A[x_i]}{|dom(x_i)|}$ which is its heuristic score.

B. DOM/WDEG

dom/wdeg [4] has been well studied for more than 10 years. It selects the variables which are most likely to lead to a failure. A $weight[c]$ is associated with each constraint c, whenever a domain wipeout is reached, the weight of the constraint that leads to the wipeout is incremented. The weighted degree of a variable x_i is defined as:

$$wdeg[x_i] = \sum_{c \in C} weight[c] \Big| x_i \in scp(c) \wedge |FutScp(c)| > 1 \tag{3}$$

where $|futScp(c)|$ denotes the number of future variables in $scp(c)$. Combining the weighted degree with domain size, *dom/wdeg* uses the ratio of the current domain size to $wdeg[x_i]$, $\frac{|dom(x_i)|}{wdeg(x_i)}$, as its heuristic score and selects the variable with the smallest score.

C. CLASSICAL COMBINING STRATEGIES

The strategy of *Brelaz* [8] heuristic can be used to combine two heuristics, heu_1 and heu_2. It first selects candidate variables by heu_1. Then it uses heu_2 to select a variable from $cand(heu_1)$. We denote this strategy as $\textsc{add}(heu_1, heu_2)$. It is obvious that heu_1 is much more important than heu_2. If heu_1 makes bad decisions, heu_2 can not help. The problem is that we usually do not know which heuristic works better on a problem before solving it, so it is not easy to decide which heuristic is heu_1.

The strategy of *dom/deg* [3] has been shown to be successful. The *dom/deg* heuristic is no longer as popular as before, because its participants, *min-dom* and *max-degree*, are simple. But its combining strategy is still useful. It uses multiplication (or division) to combine the heuristic scores. In this way, the two heuristic scores contribute equally to the final score. We denote this strategy as $\textsc{mul}(heu_1, heu_2)$.

III. *PARETOHEU*: A METHOD FOR COMBINING TWO VARIABLE ORDERING HEURISTICS

When we are combining heuristic scores, the aim is to find the variable which maximizes both scores simultaneously. If such variables do not exist, we should balance the scores to make the selection of the next variable, e. g., the scores should contribute equally to the decision. We employ the notion of Pareto optimality which is the well-studied tradeoff strategy from multi-objective optimization problem [22]. Given two variable ordering heuristics heu_1 and heu_2, $sc_1[x_i]$ and $sc_2[x_i]$ are the heuristic scores of variable x_i calculated by heu_1 and heu_2 respectively.

Definition 1: Given two future variables x_i and x_j, we say that x_i dominates x_j with respect to heu_1 and heu_2 iff either

$$sc_1[x_i] \geq sc_1[x_j] \quad and \quad sc_2[x_i] > sc_2[x_j], \quad or$$
$$sc_2[x_i] \geq sc_2[x_j] \quad and \quad sc_1[x_i] > sc_1[x_j].$$

A future variable x_i is called a Pareto candidate iff there is no variable $x_j \in FutVars$ such that x_j dominates x_i. The set of all Pareto candidates is called a Pareto front, denoted by $cand(\textsc{pareto}(heu_1, heu_2))$.

We are going to use a novel strategy, namely *ParetoHeu*, to select the next future variable. It first generates the Pareto front $cand(\textsc{pareto}(heu_1, heu_2))$, then randomly select a variable from the set. It is obvious that $cand(\textsc{pareto}(heu_1, heu_2))$ and $cand(\textsc{pareto}(heu_2, heu_1))$ are the same set. In this strategy, heu_1 and heu_2 contribute equally. The following Algorithm 1 generates a Pareto front.

Algorithm 1 Pareto($FutVars$, sc_1, sc_2)

```
1:  $ParetoFront \leftarrow \emptyset$;
2:  for each variable $x_i$ in $FutVars$ do
3:    $isPareto \leftarrow$ true;
4:    for each variable $x_j$ in $ParetoFront$ do
5:      if $x_i$ dominates $x_j$ then
6:        $ParetoFront \leftarrow ParetoFront \setminus \{x_j\}$;
7:      else
8:        if $x_j$ dominates $x_i$ then
9:          $isPareto \leftarrow$ false;
10:       end if
11:       break;
12:     end if
13:   end for
14:   if $isPareto$ then
15:     $ParetoFront \leftarrow ParetoFront \cup \{x_i\}$;
16:   end if
17: end for
18: return $ParetoFront$;
```

The algorithm compares every variable x_i in $FutVars$ with the variables x_j in $ParetoFront$. If x_i is not dominated by any x_j, it is added into $ParetoFront$, otherwise, it will not be added into $ParetoFront$. All the variables x_j that are dominated by x_i will be removed from the set. If x_i and x_j have same scores of both heuristics, both them are Pareto candidates and x_i is added into $ParetoFront$. It is straightforward to see that the worst case time complexity of Algorithm 1 is $O(kp)$, where k is the size of $FutVars$ and p is the size of $ParetoFront$. Because the outer loop at line 2 costs $O(k)$ time and the inner loop at line 4 costs $O(p)$ time.

Proposition 1: $cand(\textsc{pareto}(heu_1, heu_2))$ is a superset of both $cand(\textsc{add}(heu_1, heu_2))$ and $cand(\textsc{mul}(heu_1, heu_2))$.

Proof: Each variable x_i in $cand(\textsc{add}(heu_1, heu_2))$ is a Pareto candidate, because if there exists another variable x_j dominating x_i, then x_j is in $cand(\textsc{add}(heu_1, heu_2)$ and x_i is not. It is the same reason for $cand(\textsc{mul}(heu_1, heu_2))$. ■

It is well known that a single heuristic has the risk be trapped in heavy-tailed phenomena. The randomization based searching strategy randomly selects a variable from a set of top-ranked variables [30]. The searching equipped with randomization may not be stuck to the top-ranked variable, because it always randomly selects a variable from the set of top-ranked variables. The *ParetoHeu* has a nice feature that it selects a variable from a set in most cases. This is because the Pareto front usually contains more than one candidate variables. Thus the risk of being trapped in heavy-tailed phenomena is reduced. The only case where the Pareto front contains one variable is that the best variables selected by the participant heuristics are the same one. In this case, we should select the unique variable.

IV. EXPERIMENTS

The experiments were run in Choco [18], an open source constraint solver written in Java. The environment is JDK8 under CentOS 6.4 with Intel Xeon CPU E7-4820@2.00GHz processor and 58 GB RAM.

A. PARTICIPANT HEURISTICS

We choose ABS and *dom/wdeg* (DW) as the participant heuristics, because they have been the most popular heuristics used in constraint solvers and it is cheap to calculate the heuristic scores. We did not select *dom/wtdeg* [31], count-based search (CBS) [6] and the Explanation-Based Weighted Degree [32], because it is heavy to calculate the heuristic scores. Moreover, they have not been implemented in publicly available solvers. We choose the solver-built-in heuristics makes the comparison fair and justified.

In order to make both participants select the variables with the highest scores, we use $\frac{wdeg(x_i)}{|dom(x_i)|}$ as the score of *dom/wdeg*. Besides the participant heuristics, we compared the following classic combining strategies with *ParetoHeu*.

- *A+D*: $\text{ADD}(ABS, DW)$, the ABS heuristic with DW breaking ties.
- *D+A*: $\text{ADD}(DW, ABS)$, the DW heuristic with ABS breaking ties.
- *A×D*: $\text{MUL}(DW, ABS)$, it uses ABS×DW as the heuristic score.

B. IMPLEMENTATION DETAILS

The binary branching strategy is used by all the heuristics [33]. The value heuristic of *dom/wdeg* is lexicographical. For the other strategies involving ABS, the value heuristic is the value selector of ABS, which selects the value with the least activity. The initialization procedure of ABS is performed for all the strategies involving ABS. The parameters of ABS are set as $\alpha = 8$, $\gamma = 0.999$ and $\delta = 0.2$, which are the setting used in the ABS paper [7]. This is also the default settings in Choco.

We also tested the heuristics with two geometric restart strategies [30], [34]. With a geometric restart strategy, every time the searching reaches a failure limit, it restarts from the root. After each restart, the limit is increased by an increasing factor ρ, e. g., $limit_{after} = \rho \times limit_{before}$. One of restart strategies we have tested is a fast-restart strategy with increasing factor $\rho = 1.1$. The other one is a slow-restart strategy with increasing factor $\rho = 2$. The initial failure limit of them is set to $3 \times |X|$. All remaining ties are broken randomly.

C. DATASETS

In order to examine the robustness of the proposed strategy, we tested 7 different kinds of benchmark problems in the experiments:

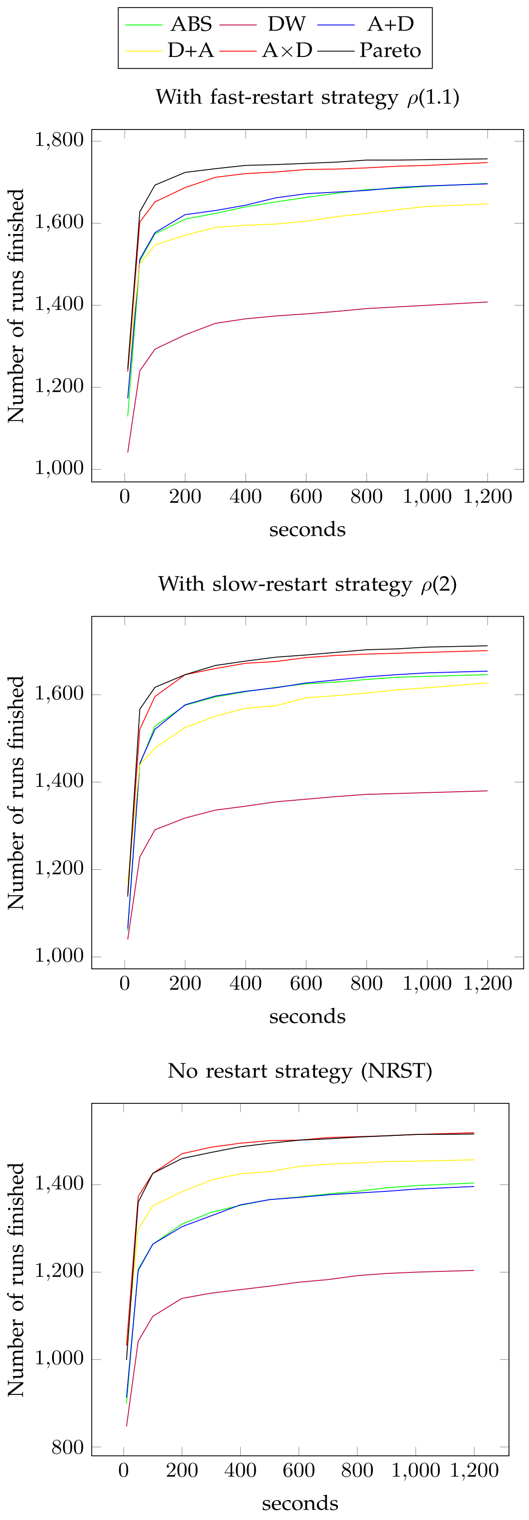

FIGURE 1. Number of runs finished in different time periods. The x-axis of the first spots is 10 seconds.

- *NumberPartitioning*: This problem consists in finding a partition of numbers 1, 2, . . ., N into two sets A and B such that: (1) A and B have the same cardinality, (2) The

TABLE 1. Average of cpu time on different problem classes.

		ABS	*DW*	*A+W*	*W+A*	*A×W*	*Pareto*
Partition (9)	$\rho(1.1)$	46.00	570.25	56.79	60.25	24.27	**9.46**
	$\rho(2)$	335.79	671.50	317.73	380.76	333.12	**304.82**
	NRST	**927.36**	1111.74	942.40	1017.59	1086.80	1041.88
Job Shop e0ddr (20)	$\rho(1.1)$	3.81	15.37	3.75	**3.44**	4.92	4.60
	$\rho(2)$	6.22	74.88	**5.66**	17.18	25.67	22.48
	NRST	160.28	408.06	160.22	**110.63**	150.98	133.94
SchurLemma (8)	$\rho(1.1)$	3.53	5.43	**3.13**	39.08	29.18	4.58
	$\rho(2)$	**3.01**	9.67	4.61	5.15	7.09	6.50
	NRST	7.58	58.61	7.75	10.95	9.12	**5.69**
SocialGolfer (14)	$\rho(1.1)$	126.29	239.67	119.96	522.27	48.19	**28.71**
	$\rho(2)$	133.52	230.93	123.92	437.06	61.12	**34.66**
	NRST	260.34	351.52	284.65	421.43	54.22	**30.32**
QWH20 (10)	$\rho(1.1)$	192.15	132.92	197.32	49.59	53.96	**49.21**
	$\rho(2)$	163.52	62.76	170.29	41.33	43.42	**40.80**
	NRST	161.32	63.67	179.50	**37.82**	37.93	48.30
BIBD (18)	$\rho(1.1)$	5.61	170.33	4.11	43.39	3.82	**3.46**
	$\rho(2)$	4.64	157.66	4.41	38.18	4.15	**3.55**
	NRST	3.98	59.73	**3.36**	10.18	3.39	3.51
MagicSquare (9)	$\rho(1.1)$	332.76	1191.14	295.21	**44.26**	157.19	100.43
	$\rho(2)$	342.98	1192.52	349.36	**76.11**	140.70	153.03
	NRST	873.80	1188.88	828.75	304.17	**299.77**	439.48
Average (88)	$\rho(1.1)$	83.00	272.20	79.48	112.62	36.91	**23.57**
	$\rho(2)$	111.88	284.66	109.90	133.14	70.44	**63.40**
	NRST	281.89	408.74	284.63	234.74	**190.58**	193.49

TABLE 2. Detailed results of some representative instances.

		ABS	*DW*	*A+W*	*W+A*	*A×W*	*Pareto*
Partition-132	μ(cpu)	117.67	790.96	127.78	78.86	131.52	**31.25**
	σ(cpu)	331.52	513.23	365.25	268.59	366.03	81.23
	μ(nodes)	-	-	-	-	-	**261,638**
e0ddr-0-3	μ(cpu)	59.44	240.85	60.39	**55.0**	85.04	78.33
	σ(cpu)	199.04	492.03	212.90	173.87	245.97	159.74
	μ(nodes)	488,328	-	488,328	**309,866**	544,455	550,074
SchurLemma (44-4)	μ(cpu)	18.32	32.53	**16.62**	305.29	225.29	28.64
	σ(cpu)	24.02	36.81	15.31	356.39	369.26	36.07
	μ(nodes)	37,689	47,960	**31,322**	569,637	436,924	48,978
SocialGolfer (5-8-4)	μ(cpu)	327.38	541.14	250.66	855.09	44.27	**26.40**
	σ(cpu)	516.06	611.44	453.31	430.48	56.30	14.18
	μ(nodes)	-	-	-	-	34,266	**29,360**
QWH20-3	μ(cpu)	118.82	30.40	122.03	**11.99**	29.80	14.88
	σ(cpu)	209.40	23.74	200.58	9.54	49.84	16.41
	μ(nodes)	267,410	62,404	267,410	**23,243**	54,354	33,359
BIBD (7-98-42-3-14)	μ(cpu)	10.27	365.52	8.91	68.07	11.99	**7.57**
	σ(cpu)	1.65	514.22	2.97	266.46	16.71	1.17
	μ(nodes)	64,298	-	67,348	-	74,835	**64,278**
MagicSquare-15	μ(cpu)	272.27	1200	265.89	**24.64**	151.60	59.33
	σ(cpu)	301.93	0	366.63	20.01	215.34	56.60
	μ(nodes)	2.27M	-	-	**0.32**M	1.57M	0.73M

sum of numbers in A equals to that in B, (3) The sum of squares of numbers in A equals to that in B. There is no solution for $N<8$ and from $N\geq 8$, there is no solution if N is not a multiple of 4. So we tested some solvable instances with N from 100 to 132.

- *SocialGolfer*: This is a problem from Operations Research. The problem is to schedule g groups of s golfers over w weeks, such that no golfer plays in the same group as any other golfer twice. A SocialGolfer instance is specified by its parameters (w-s-g).
- *MagicSquare*: An order n magic square is a n by n matrix containing the numbers 1 to n^2, with each row, column and main diagonal equal the same sum.
- *Balanced Incomplete Block Designs (BIBD)*: This is a standard combinatorial problem from design theory. A BIBD is defined as an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks. A BIBD is therefore specified by its parameters (v, b, r, k, λ).

- *SchurLemma*: The problem is to put n balls labelled 1, ..., n into k boxes so that for any triple of balls (x, y, z) with $x+y=z$, not all are in the same box. A *SchurLemma* instance is specified by its parameters (n, k).
- *Job-Shop-e0ddr*: This is the Job Shop Scheduling problem studied in [35]. The instances are presented with binary extensional constraints. We have tested the e0ddr1 and e0ddr2 series.
- *Quasi-group With Holes (QWH20)*: This is a variant of the *LatinSquare* problem. A *Latinsquare* is an $n\times n$ array filled with n different symbols, each occurring exactly once in each row and exactly once in each column. Given a partial *Latinsquare* where some of the grids have already been filled, the task is to extend it to a complete *LatinSquare*. The partial *LatinSquares* are generated in such a way that they are guaranteed to be extended to complete ones. The instances were used in a CSP solver competition.

The *Job-Shop* and *QWH* problems are from http://www.cril.univ-artois.fr/~lecoutre/#/benchmarks and the others are from http://www.csplib.org/. More details about the benchmarks can be found from the web sites.

D. METRICS

The performance of searching for the first solution or proving unsatisfiable are measured by cpu time in seconds and search tree nodes. Timeout is set to 1200 seconds. For each instances, we run 20 times with different random seeds (1-20) and use the average of the 20 runs as the result of the instance. The time cost of a timeout run is count as 1200 seconds. If all 20 runs are timeout, the heuristic for this instance is considered as timeout. We have eliminated those instances for which all the tested heuristics are timeout. There are 88 remaining instances, so we have 1760 runs in total.

Firstly, we present the number of runs finished in given time limits in Figure 1. From the Figure, we can see that the fast-restart strategy is the best restart strategy for these problems. With different restart strategies, the performance of *ABS* and $A+D$ are close, while $D+A$ performs much better than *DW*. $A\times D$ and *ParetoHeu* always perform best. With both the fast and the slow restart strategies, *ParetoHeu* performs better than $A\times D$. The *ParetoHeu* with fast-restart strategy has the largest number of runs, 1757, finished in 1200s.

Secondly, we present the average time cost of each type of problems in Table 1. The best one of each row is in bold. The integer in the brackets under each problem name is the number of instances of the problem. From the last group, the average of all the 88 instances, we can see that *ParetoHeu* has the best performance. Although it loses a little when no restart strategy is used, it dominates the other searching strategies when equipped with restart strategies. For each type of problems, we have 3 rows of results, so we have 21 rows in total. In the 21 rows, we can see that *ParetoHeu* performs best in 10 rows, which is clearly more than that of other searching strategies. Despite the $W+A$ gets the best performance in *e0ddr* and *MagicSquare* problems, its performance is poor in *SocialGolfer* and *BIBD* problems, so its average performance is much worse than $A\times W$ and *ParetoHeu*.

Finally, we present the detailed results of some representative instances in Table 2, where μ(cpu) is the average cpu time of the 20 runs, σ(cpu) is the standard deviation of the cpu time and μ(nodes) is the average search tree nodes. If any of the 20 runs is timeout, then the μ(nodes) is present as -. It has been shown that the fast-restart strategy is the most efficient one in the experiments, so the results in the table are the detailed results with the fast-restart strategy. From the table, we can see that *ParetoHeu* gets the best performance on the *Partition*, *SocialGolfer* and *BIBD* instances. It gets no worst performance. While $W+A$ gets the best performance on the *e0ddr*, *QWH* and *MagicSquare* instances, it gets poor performance on the *SchurLemma* and *SocialGolfer* instances. Note that *e0ddr* and *QWH* are binary problems and the heu_1 of $W+A$ is the state of the art heuristic for binary problems, so $W+A$ gets the best performances on them. From the previous results, we know that $A\times W$ and *ParetoHeu* are the best strategies. On these representative instances, $A\times W$ is clearly outperformed by *ParetoHeu*.

In summary, fast-restart is the best restart strategy on most of the tested problems. Generally speaking, the combining strategy of breaking ties, $A+D$ and $D+A$, are outperformed by the other two combining strategies, $A\times D$ and *ParetoHeu*. It is shown that *ParetoHeu* is more efficient than $A\times D$, so it is the most robust one in general.

V. CONCLUSION AND FUTURE WORK

We propose a method *ParetoHeu* combining two variable ordering heuristics for constraint satisfaction problems. A novel strategy based on Pareto optimality of multi-objective optimization problem is employed to combine the scores. The proposed method has a nice feature that it is equipped with randomized search naturally, because its candidate variable set, the Pareto front, usually contains more than one candidate variables. The experiments were run with the popular constraint solver Choco. The results of 7 well-known benchmarks show that *ParetoHeu* outperforms the participant heuristics and it is in general more robust than some classical strategies combining variable ordering heuristics. Its average cpu time is one third less than that of the best of the existing combining strategies.

To generate more robust searching strategies, we will investigate using Pareto optimality to combine more heuristic scores in the future.

REFERENCES

[1] R. M. Haralick and G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," *Artif. Intell.*, vol. 14, no. 3, pp. 263–313, 1980.

[2] R. Dechter and I. Meiri, "Experimental evaluation of preprocessing algorithms for constraint satisfaction problems," *Artif. Intell.*, vol. 68, no. 2, pp. 211–241, 1994.

[3] C. Bessière and J.-C. Régin, "MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems," in *Proc. Int. Conf. Princ. Pract. Constraint Program.* Berlin, Germany: Springer, 1996, pp. 61–75.

[4] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *Proc. ECAI*, vol. 16, 2004, pp. 146–150.

[5] P. Refalo, "Impact-based search strategies for constraint programming," in *Proc. Int. Conf. Princ. Pract. Constraint Program.* Berlin, Germany: Springer, 2004, pp. 557–571.

[6] G. Pesant, C.-G. Quimper, and A. Zanarini, "Counting-based search: Branching heuristics for constraint satisfaction problems," *J. Artif. Intell. Res.*, vol. 43, pp. 173–210, Feb. 2012.

[7] L. Michel and P. Van Hentenryck, "Activity-based search for black-box constraint programming solvers," in *Proc. CPAIOR*. Berlin, Germany: Springer, 2012, pp. 228–243.

[8] D. Brélaz, "New methods to color the vertices of a graph," *Commun. ACM*, vol. 22, no. 4, pp. 251–256, 1979.

[9] E. K. Burke *et al.*, "Hyper-heuristics: A survey of the state of the art," *J. Oper. Res. Soc.*, vol. 64, no. 12, pp. 1695–1724, 2013.

[10] A. Arbelaez, Y. Hamadi, and M. Sebag, "Online heuristic selection in constraint programming," in *Proc. Int. Symp. Combinat. Search*, 2009. [Online]. Available: https://hal.inria.fr/inria-00392752/document

[11] J. C. Ortiz-Bayliss, H. Terashima-Marín, and S. E. Conant-Pablos, "A supervised learning approach to construct hyper-heuristics for constraint satisfaction," in *Proc. Mexican Conf. Pattern Recognit.* Berlin, Germany: Springer, 2013, pp. 284–293.

[12] H. Terashima-Marín, J. C. Ortiz-Bayliss, P. Ross, and M. Valenzuela-Rendón, "Hyper-heuristics for the dynamic variable ordering in constraint satisfaction problems," in *Proc. 10th Annu. Conf. Genetic Evol. Comput.*, 2008, pp. 571–578.

[13] J. C. Ortiz-Bayliss, H. Terashima-Marín, and S. E. Conant-Pablos, "Combine and conquer: An evolutionary hyper-heuristic approach for solving constraint satisfaction problems," *Artif. Intell. Rev.*, vol. 46, no. 3, pp. 327–349, 2016.

[14] M. Streeter, D. Golovin, and S. F. Smith, "Combining multiple heuristics online," in *Proc. AAAI*, 2007, pp. 1197–1203.

[15] N. R. Sabar, M. Ayob, R. Qu, and G. Kendall, "A graph coloring constructive hyper-heuristic for examination timetabling problems," *Appl. Intell.*, vol. 37, no. 1, pp. 1–11, 2012.

[16] M. Loth, M. Sebag, Y. Hamadi, and M. Schoenauer, "Bandit-based search for constraint programming," in *Proc. Int. Conf. Princ. Pract. Constraint Program.* Springer, 2013, pp. 464–480.

[17] W. Xia and R. H. C. Yap, "Learning robust search strategies using a bandit-based approach," in *Proc. AAAI*, 2018, pp. 431–437.

[18] C. Prud'homme, J.-G. Fages, and X. Lorca. (2016). *Choco Documentation, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.* [Online]. Available: http://www.choco-solver.org

[19] C. Schulte, G. Tack, and M. Z. Lagerkvist. (2018). *Modeling and Programming With Gecode*. [Online]. Available: http://www.gecode.org/documentation.html

[20] C. P. Gomes and B. Selman, "Algorithm portfolios," *Artif. Intell.*, vol. 126, nos. 1–2, pp. 43–62, 2001.

[21] R. Amadini, M. Gabbrielli, and J. Mauro, "A multicore tool for constraint solving," in *Proc. IJCAI*, 2015, pp. 232–238.

[22] K. Miettinen, *Nonlinear Multiobjective Optimization*, vol. 12. New York, NY, USA: Springer, 2012.

[23] A. K. Mackworth, "Consistency in networks of relations," *Artif. Intell.*, vol. 8, no. 1, pp. 99–118, 1977.

[24] H. Collavizza, F. Delobel, and M. Rueher, "A note on partial consistencies over continuous domains," in *Proc. Int. Conf. Princ. Pract. Constraint Program.* Berlin, Germany: Springer, 1998, pp. 147–161.

[25] C. Lecoutre and F. Hemery, "A study of residual supports in arc consistency," in *Proc. IJCAI*, 2007, pp. 125–130.

[26] C. Lecoutre, "STR2: Optimized simple tabular reduction for table constraints," *Constraints*, vol. 16, no. 4, pp. 341–371, 2011.

[27] H. Li, Y. Liang, J. Guo, and Z. Li, "Making simple tabular reductionworks on negative table constraints," in *Proc. AAAI*, 2013, pp. 1629–1630.

[28] H. Li, "Narrowing support searching range in maintaining arc consistency for solving constraint satisfaction problems," *IEEE Access*, vol. 5, pp. 5798–5803, 2017.

[29] H. Li, R. Li, and M. Yin, "Saving constraint checks in maintaining coarse-grained generalized arc consistency," *Neural Comput. Appl.*, pp. 1–10, May 2017, doi: 10.1007/s00521-017-3015-7.

[30] C. P. Gomes, B. Selman, and H. Kautz, "Boosting combinatorial search through randomization," in *Proc. AAAI*, 1998, pp. 431–437.

[31] H. Li, Y. Liang, N. Zhang, J. Guo, D. Xu, and Z. Li, "Improving degree-based variable ordering heuristics for solving constraint satisfaction problems," *J. Heuristics*, vol. 22, no. 2, pp. 125–145, 2016.

[32] E. Hebrard and M. Siala, "Explanation-based weighted degree," in *Proc. CPAIOR*. Cham, Switzerland: Springer, 2017, pp. 167–175.

[33] J. Hwang and D. G. Mitchell, "2-way vs. D-way branching for CSP," in *Proc. Int. Conf. Princ. Pract. Constraint Program.* Berlin, Germany: Springer, 2005, pp. 343–357.

[34] T. Walsh, "Search in a small world," in *Proc. IJCAI*, 1999, pp. 1172–1177.

[35] N. Sadeh and M. S. Fox, "Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem," *Artif. Intell.*, vol. 86, no. 1, pp. 1–41, 1996.

HONGBO LI received the Ph.D. degree in computer science from Jilin University, China. He is currently a Post-Doctoral Researcher with the School of Information Science and Technology, Northeast Normal University, Changchun, China. His research interests include constraint programming, local consistencies, and heuristics for solving constraint satisfaction problems.

ZHANSHAN LI is currently a Professor of computer science with Jilin University. His main research interests include constraint reasoning and machine learning.