Finding Good Partial Assignments during Restart-Based Branch and Bound Search

Hongbo Li[1] **and Jimmy H.M. Lee**[2]

[1]School of Information Science and Technology, Northeast Normal University, Changchun, China
[2]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
lihb905@nenu.edu.cn, jlee@cse.cuhk.edu.hk

Abstract

Restart-based Branch-and-Bound Search (BBS) is a standard algorithm for solving Constraint Optimization Problems (COPs). In this paper, we propose an approach to find good partial assignments to jumpstart search at each restart for general COPs, which are identified by comparing different best solutions found in different restart runs. We consider information extracted from historical solutions to evaluate the quality of the partial assignments. Thus the good partial assignments are dynamically updated as the current best solution evolves. Our approach makes restart-based BBS explore different promising sub-search-spaces to find high-quality solutions. Experiments on the MiniZinc benchmark suite show how our approach brings significant improvements to a black-box COP solver equipped with the state of the art search techniques. Our method finds better solutions and proves optimality for more instances.

Introduction

Constraint Programming (CP) is a powerful paradigm for solving discrete combinatorial optimization problems. Restart-based depth-first Branch-and-Bound Search (BBS) is a standard algorithm used in black-box constraint solvers to solve Constraint Optimization Problems (COPs). Due to the NP-hard characteristic of general COP, search heuristics (Michel and Van Hentenryck 2012; Palmieri and Perez 2018; Demirovic, Chu, and Stuckey 2018; Fages and Prud'Homme 2017) play an important role in BBS. An efficient search heuristic may find a high quality or even an optimal solution quickly. Besides search heuristics, exploring a good sub-search-space containing high quality solutions first will speed up the finding of an optimal solution. However, as far as we know, there exists no approach that can always find a sub-search-space containing optimal solutions for general COPs before solving the problems. To avoid search being trapped in a bad sub-search-space, the restart technique (Gomes, Selman, and Kautz 1998) helps BBS explore different sub-search-space and has been a necessary component in modern black-box COP solvers.

Finding good search entrance for general COPs is not easy. Restart techniques usually help BBS switch search entrances by utilizing the heuristic information accumulated in different restart runs (Boussemart et al. 2004; Michel and Van Hentenryck 2012; Li, Yin, and Li 2021). Recently, Frequent Pattern Mining-based Search (FPMS) (Li et al. 2020) is proposed to find good subtrees for solving COPs. It makes BBS directly zoom into a promising sub-search-space by running a pre-processing phase using frequent pattern mining on high quality sampled solutions. However, FPMS is suitable for only loosely constrained problems. In this paper, we propose a novel approach to find good initial partial assignments for general COPs. The approach discovers good partial assignments as search entrances that can be extended to high quality or even optimal solutions. The partial assignments are updated as the current best solution evolves, so that the approach is naturally incorporated with restart techniques to explore different sub-search-space specified by the partial assignments. Our idea is based on a simple premise. Given two feasible solutions S_1 and S_2, where S_1 is considered better than S_2. If a solution is a set of variable assignments, then $S_1 \setminus S_2$ contains the assignments that makes S_1 better. In our approach, the best solution at each restart run is always added to the front of a queue. The pairwise difference of adjacent solutions in the queue are extracted to form the basis of the good sub-search-space to jumpstart the next restart run. In case a run fails to find a solution within the restart limit, the partial assignment is cut by half and a larger sub-search-space will be explored. The procedure repeats until either a new solution is found or the partial assignment is empty and the default search strategy takes over. Extensive experimentation with the MiniZinc benchmark suite demonstrates that our approach significantly improves the performance of the state of the art search strategy for a black-box COP solver. Our method is able to find better solutions and prove optimality for more instances.

Background

Constraint Optimization Problem

A *Constraint Optimization Problem* (COP) $\mathcal{P}$ is a 4-tuple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{F} \rangle$, where $\mathcal{X}$ is a set of variables, the domain $dom(x) \in \mathcal{D}$ specifies the set of possible values for each $x \in \mathcal{X}$, $\mathcal{C}$ is a set of constraints and $\mathcal{F}$ is an objective function. Each constraint $c \in \mathcal{C}$ specifies the allowed combinations of values for a subset of variables. An *assignment* $\mathcal{A}$ to variables $X \subseteq \mathcal{X}$ is a set of *instantiations* of the form x/v,

one for each $x \in X$ to assign v to x where $v \in dom(x)$. If $X = \mathcal{X}$, then $\mathcal{A}$ is a *complete assignment*; otherwise a *partial assignment*. An assignment to a single variable is called a *singleton-assignment* or simply *s-assignment*. A complete assignment S that satisfies all constraints is a *feasible solution* to $\mathcal{P}$. The *search space* of $\mathcal{P}$ is the Cartesian product of the domains of all variables. Each partial assignment $\mathcal{A}$ specifies a sub-search-space. The *length* of a partial assignment is the number of s-assignments involved in it. The objective function $\mathcal{F}$ maps every complete assignment of $\mathcal{P}$ to a real number. Without loss of generality, we consider here minimization and a feasible solution S^* is an *optimal solution* if $\mathcal{F}(S^*) \leq \mathcal{F}(S)$ for any other feasible solution S to $\mathcal{P}$. In case of maximization, optimal solutions are defined simply with $\leq$ replaced by $\geq$. For simplicity, we consider minimization problems in this paper.

Restart-based Depth-First BBS

A COP $\mathcal{P}$ can be solved by Branch-and-Bound Search (BBS) augmented with constraint propagation (Bessiere 2006). This work focuses on search strategies. Thus we skip the details of constraint propagation and always adopt the solver's default propagation engine. BBS explores the search space with depth-first backtracking tree search resulting in a search tree. The main idea is to use the last best feasible solution found as an upper bound to help prune search space. When no more solutions can be found, optimality is proved and the last feasible solution found can be returned as the optimal solution of the COP. While building a search tree, BBS makes decisions with search heuristics, e.g., selecting an s-assignment. To avoid being trapped in a bad sub-search-space, modern constraint solvers usually employ restart techniques to switch different search entrances, e.g., the initial decisions of a search tree. Restart techniques set a *cutoff* to trigger restarts. Whenever the amount of a given resource reaches the cutoff during search, such as the number of failures or the number of search tree nodes visited, the search is restarted from the root node with a new search order. Each restart run will build a new search tree. To ensure the completeness of BBS, the popular restart strategies usually set an initial cutoff c and the cutoff is increased as the restart number r increases. For instance, the geometric restart (Gomes, Selman, and Kautz 1998; Walsh 1999) increases the cutoff with a growing factor ρ, e.g., $c \times \rho^r$. Luby restart uses the Luby sequence (Luby, Sinclair, and Zuckerman 1993) to set the cutoff, e.g, $c \times l_r$ where l_r is the number returned by Luby sequence at time r.

A high-level description of restart-based BBS is included in Algorithm 1 (please ignore the boxed part for the moment). Our approach is built on this framework. If the *fgpa* flag is set to false, i.e., the boxed parts are skipped, then the Algorithm 1 degenerates to the basic restart-based BBS. Whenever a new feasible solution S is found, S is recorded as the *current best solution* and $\mathcal{F}(S)$ is used to bound subsequent search to ensure the next feasible solution found must be strictly better than S at line 18. The algorithm terminates in three cases: (1) it returns the best solution found so far when a termination condition (such as a time limit) is reached; (2) it returns null if unsatisfiability is proved and no feasible solution is found; (3) it returns an optimal solution if unsatisfiability is proved and some feasible solutions are found. One possibility to define the restart condition is using the number of failures encountered during a search run. Whenever the number of failures reaches *restartCutoff* at line 6, the search is restarted at line 9.

Algorithm 1: Restart-based depth-first BBS

```
   Input: a COP P and a termination condition;
   Output: a solution S;
1  S ← null;
   if fgpa then
2      solQueue ← empty;                                   } while the
       newSolutionFound ← false;
   termination condition is not reached do
3      x/v ← MAKEDECISION(); // Algorithm 4
4      if propagating the s-assignment x=v fails then
5          if the restartCutoff is reached then
6              set next restartCutoff by the default
                restart strategy;
               if fgpa then
7                  PREPAREFORRESTART();
                       // Algorithm 2
8              restart the search;
9          else
10             cancel the decision and backtrack;
11             if unsatisfiability is proved then
12                 return S;
13     else
14         if all variables are assigned then
15             S ← current best solution;
               if fgpa then
16                 newSolutionFound ← true;
                   lastSolution ← S;
               add a constraint to ensure next solution
                is better than S;
17             cancel the last assignment and backtrack;
18             if unsatisfiability is proved then
19                 return S;
20 return S;
```

Our approach will work as a plugin that improves the default search strategy for COPs, so that there is a default restart strategy at line 7 in Algorithm 1. In the following pseudocodes, all the variables with the same names are global variables.

Finding Good Partial Assignments for COPs

We will first introduce the motivation of this work and the main workflow of the proposed approach. Then we introduce how to implement the workflow in Algorithm 1.

The Motivation

A partial assignment $\mathcal{A}$ is *optimal* if it can be extended to an optimal solution. If we know an optimal partial assignment $\mathcal{A}$, then we can find an optimal solution in a sub-search-space of size O($d^{n-|\mathcal{A}|}$) instead of the entire search-space of size O(d^n), where n is the variable number and d is the maximum domain size. However, as far as we know, no approach can always identify optimal partial assignments for general COPs before solving it. Thus the aim of our approach is to find promising partial assignments that have a higher possibility of being optimal, or can be extended to a high-quality solution whose objective is close to that of an optimal one.

Given a COP P and its feasible solutions S_1 and S_2 with different objectives. We consider each of the solutions as a set of s-assignments. $S_1 \cap S_2$ forms a partial assignment of P. Assuming that $\mathcal{F}(S_1) < \mathcal{F}(S_2)$, then in the sub-search-space specified by $S_1 \cap S_2$, the partial assignment $S_1 \setminus S_2$ is better than $S_2 \setminus S_1$, because $S_1 = (S_1 \cap S_2) \cup (S_1 \setminus S_2)$ and $S_2 = (S_1 \cap S_2) \cup (S_2 \setminus S_1)$. In a sense, $S_1 \setminus S_2$ is what makes S_1 a better solution. Intuitively, the current best solution evolves towards optimal solutions in BBS. The partial assignments extracted from the current best solution should evolve towards optimal partial assignments.

The main workflow of our approach is shown in Figure 1. The variable $solQueue$ is a FIFO queue with a maximum size m (a parameter) storing some representative solutions. A new solution will be added to the head of $solQueue$ and the earliest added one will be removed when the queue is full. The variable $paList$ is a list storing all the s-assignments extracted from $solQueue$. The variable $subtree$ is a list storing the partial assignment to be explored in next restart. The details of the algorithm will be introduced in the following subsections.

Generating Good Partial Assignments

To generate good partial assignments, we collect the last and best solution found by BBS at each restart run. The collection is done at lines 3-5 of Algorithm 2 which is called at line 8 of Algorithm 1 before each restart. We select only the last solution as the representative solution of the current search tree and store these solutions in the queue $solQueue$. The good partial assignments will be extracted from the stored solutions. In BBS, a new solution is always better than those found before it. We always add a new representative solution to the head of $solQueue$. Thus the solutions in $solQueue$ are naturally sorted by their objectives and the best one is at the head. Other parts of Algorithm 2 are related to utilizing good partial assignments, and we will explain those later.

We track the evolution of the solutions to extract good partial assignments. Algorithm 3 is the procedure of Generating Partial Assignments (GPA), which synthesizes a list $paList$ storing all the extracted s-assignment. Whenever a new solution is added into $solQueue$, a new $paList$ will be created. The GPA procedure iterates over $solQueue$ from the head and extracts a partial assignment $S_i \setminus S_{i+1}$ from each pair of adjacent solutions S_i and S_{i+1}. It always puts the partial assignments extracted from better solutions at earlier locations in $paList$, e.g., the s-assignments in $S_i \setminus S_{i+1}$ are always stored before those in $S_j \setminus S_{j+1}$ ($j > i$), where i and j are the indexes of the solutions in $solQueue$.

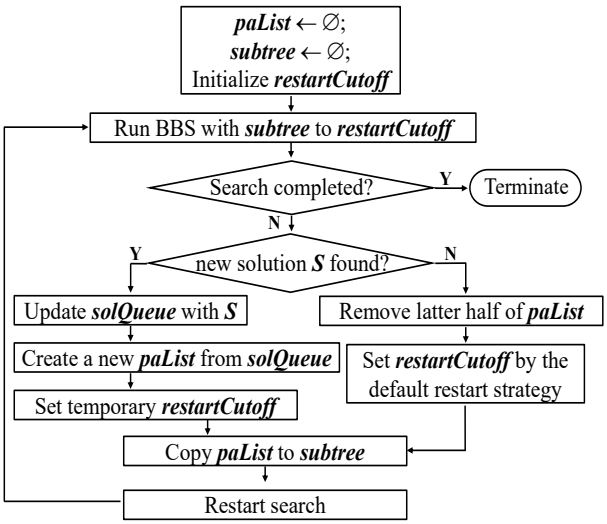

Figure 1: The main workflow of our approach.

Algorithm 2: PREPAREFORRESTART

```
1  restartNum ← restartNum + 1;
2  if newSolutionFound then
3      add lastSolution to the head of solQueue ;
4      if solQueue.size > m then
5          remove the last one from solQueue ;
6      newSolutionFound ← false;
7      GPA();
8      defCut ← the cutoff returned by the default
         restart sequence;
9      set restartCutoff to restartNum×defCut;
10     restartNum ← 1;
11 PREPARESUBTREE(); // Algorithm 5
```

Algorithm 3: GPA

```
1  paList ← empty;
2  for i = 1 to solQueue.size - 1 do
3      for each s-assignment x_i/v_i in S_i \ S_{i+1} do
4          if x_i/v_i is not in paList then
5              add x_i/v_i at the end of paList;
```

Utilizing Good Partial Assignments in BBS

The variable $subtree$ is essentially a copy of $paList$ (to be explained in Algorithm 5). We use a simple strategy to invoke BBS to first explore the sub-search-space specified by $subtree$ (a list storing the partial assignment to be used) in Algorithm 4. This is a normal depth-first BBS except that the initial assignments are specified. At each step, the first s-assignment x/v in $subtree$ is selected and removed from the list. If v is still in the domain of x, it will be used to make

Algorithm 4: MAKEDECISION

```
1 while subtree is not empty AND fgpa do
2     x/v ← select and remove the first s-assignment
        from subtree;
3     if v is still in dom(x) then
4         return x/v;
5 return the decision made by the default heuristic;
```

Algorithm 5: PREPARESUBTREE

```
1 subtree ← empty;
2 copy paList to subtree;
3 remove the s-assignments indexed from
    paList.size/2 + 1 to paList.size from paList;
```

a decision. After $subtree$ is exhausted, the default search strategy takes over. Thus the search will not be limited in the sub-search-space and the rest of the search-space will be explored.

The list $subtree$ is generated from $paList$ in Algorithm 5 which is called at line 11 of Algorithm 2. In each call of PREPARESUBTREE procedure, it copies all the s-assignments from $paList$ to $subtree$ with the initial ordering and removes the latter half from $paList$. The PREPARESUBTREE procedure is called before each restart, so that if a better solution is not found in current run, only the first half of $paList$ will be copied to the $subtree$ for next restart run. The reason for the operation (line 3) is as follows. A longer partial assignment results in a smaller sub-search-space. We would prefer longer partial assignments that can be extended to an optimal solution. However, we do not know which of the s-assignments in $paList$ are involved in an optimal solution. Therefore, we explore the smallest sub-search-space when a new $paList$ is generated. After that, if no solution is found in the current run, we remove the bottom half of the partial assignment and explore a larger sub-search-space in the next restart run. The procedure repeats until either a new solution is found, or the partial assignment is empty. In the latter case, the default search strategy takes over.

We design a strategy to give more resource to the next run after a new $paList$ is generated, because we expect to find a new solution in the smallest sub-search-space specified by the new partial assignment. The idea is to set a temporary restart cutoff for the next run, which is usually larger than the default cutoff. The temporary restart cutoff is set to *defCut*×*restartNum* at line 9 of Algorithm 2, where *defCut* is the cutoff returned by the default restart sequence and *restartNum* is the number of restarts before the last representative solution was found, which is accumulated in PREPAREFORRESTART procedure. The temporary cutoff will overwrite the cutoff set at line 7 of Algorithm 1. The variable *restartNum* is initialized to 1 and reset to 1 at line 10 of Algorithm 2 after a new solution is found. Whenever a restart run fails to find a new solution, *restartNum* will be incremented by 1. It is used to measure the difficulty of finding a new solution. The intuition is that the partial assignments extracted from a hard to find solution may contain some good s-assignments that are hard to find, so that we should give more resource to explore the corresponding sub-search-space. Thus the more difficult to find the current solution, the larger the temporary cutoff is set. If the first run with a temporary restart cutoff fails to find a solution, the default restart strategy will take over until the next feasible solution is found.

Refining Procedure GPA

Algorithm 3 may put some good s-assignments at the latter locations of $paList$, because some good ones may be extracted from a solution found in an earlier run. Thus, we propose a refined strategy utilizing the average contribution of each s-assignment to sort all the s-assignments in $paList$. The Refined Generating Partial Assignment (RGPA) procedure is presented in Algorithm 6.

The good s-assignments will be extracted from only the most recent k solutions where k is set to $\frac{solQueue.size}{2}$. Assuming that $\mathcal{F}(S_i) < \mathcal{F}(S_j)$, and given an s-assignment, if it appears in $set = S_i \setminus S_j$, it is given an *objScore* = $\frac{\mathcal{F}(S_j)-\mathcal{F}(S_i)}{|set|}$ which is the average contribution to the improvement of objective value between S_i and S_j. We also design a strategy to give higher weight to the s-assignments extracted from a better solution (line 4). The *weight* is measured by the distance from a solution S_i to the current best solution S' (the first one in $solQueue$), e.g., $\frac{k+1-i}{k}$, where i is the index of the solution S_i in $solQueue$. We compare each S_i with the solutions from S_{i+1} to S_{i+k} and add *score* = *weight*×*objScore* considering both the *weight* and the contribution into the $scoreList$ of each s-assignment in $S_i \setminus S_j$ ($j = i+1$ to $i+k$) (lines 9-13). The $scoreList$ of an s-assignment is a list recording all the scores of the

Algorithm 6: RGPA

```
1  visited ← ∅;
2  k ← solQueue.size/2;
3  for i = 1 to k do
4      weight ← (k+1−i)/k;
5      for j = 1 to k do
6          set ← S_i \ S_{i+j};
7          objScore ← (F(S_{i+j}) − F(S_i))/|set|;
8          score ← objScore×weight;
9          for each s-assignment x/v in set do
10             if x/v ∉ visited then
11                 visited ← visited ∪ {x/v};
12                 scoreList(x/v) ← empty;
13             add score into scoreList(x/v);
14 paList ← empty;
15 sort all s-assignments in visited in descending order
     of the average of their recorded scores;
16 add the sorted s-assignments in visited into paList;
```

s-assignment and it may contain duplicate scores coming from different extractions for the s-assignment. All the s-assignments are measured and sorted by the descending order of the average of scores recorded in their $scoreList$ (lines 14-16).

Proposition 1. *The procedure* GPA *costs* $O(mn)$ *time and the procedure* RGPA *costs* $O(m^2n + (\log_2 nd)nd)$ *time, where* m *is the maximum size of* $solQueue$*,* n *is the number of variables and* d *is the maximum domain size.*

Proof. The GPA procedure compares at most m-1 pairs of solutions to extract partial assignments. It costs $O(n)$ time to generate each $S_i \setminus S_j$, so it costs $O(mn)$ time.

The RGPA procedure extracts partial assignment from at most $\frac{m}{2}$ solutions (the loop at line 3). Each of these solutions will be compared with at most $\frac{m}{2}$ solutions (the loop at line 5). So it costs $O(m^2n)$ time to collect the scores of the s-assignments. There are at most nd s-assignments in the problem. So, sorting all the s-assignment in $visited$ costs $O((\log_2 nd)nd)$ time. Thus, the RGPA procedure costs $O(m^2n + \log_2 nd)nd$ time. □

Proposition 2. *Both the* GPA *procedure and* RGPA *procedure require* $O(mn + nd)$ *space.*

Proof. The $solQueue$ costs $O(mn)$ space to store the solutions. The $paList$ contains at most nd s-assignments, so the GPA procedure needs $O(mn + nd)$ space.

The RGPA procedure records a $scoreList$ for each s-assignment. To calculate the average score for each s-assignment, we can use an implementation costing O(1) space to record the number of scores and the intermediate average result. So the RGPA procedure also needs $O(mn + nd)$ space. □

Note that both GPA and RGPA may create a $paList$ containing more than one s-assignment for a variable. We keep multiple s-assignments x/v_1 and x/v_2 for x for two reasons. Assuming that x/v_1 is before x/v_2. Firstly, if x/v_1 is used to make a decision, then x/v_2 will be filtered by propagation, so the lines 3-4 of Algorithm 4 guarantees that x/v_2 will not take effect. Secondly, the s-assignments in $paList$ are from different solutions, so they may be inconsistent. If x/v_1 is inconsistent with those s-assignments before it, it will be filtered by propagation. In this case, x/v_2 can be used as an alternate decision, which should be better than a decision made by the default strategy.

Example. The following is an example of the different $paList$s generated by GPA and RGPA. Table 1 presents the assignments of 4 solutions recorded in $solQueue$. S_4 is the current best solution located at the head of $solQueue$ and S_1 is the best solution from the first restart run and it is located at the end of $solQueue$.

• The GPA procedure does not consider the objectives. It extracts $x_1/2$, $x_5/4$, $x_6/4$ and $x_7/4$ from $S_4 \setminus S_3$, $x_1/3$, $x_3/3$ and $x_4/3$ from $S_3 \setminus S_2$, $x_1/2$ and $x_2/2$ from $S_2 \setminus S_1$. Thus it creates a $paList$: $x_1/2$, $x_5/4$, $x_6/4$, $x_7/4$, $x_1/3$, $x_3/3$, $x_4/3$, $x_2/2$.

• The RGPA procedure considers more information. There are 4 solutions in $solQueue$, so k is set to 2 here. When finding good partial assignments from S_4, the $weight$ is 1. The procedure first extracts $x_1/2$, $x_5/4$, $x_6/4$ and $x_7/4$ from $S_4 \setminus S_3$, and then it adds $\frac{(60-54)\times 1}{4} = 1.5$ into the $scoreList$ of each of the four s-assignments. Secondly, it extracts $x_3/3$, $x_4/3$, $x_5/4$, $x_6/4$ and $x_7/4$ from $S_4 \setminus S_2$, and adds 7.2 into the $scoreList$ of each of the five s-assignments. When extracting good partial assignments from S_3, the $weight$ is 0.5. It first extracts $x_1/3$, $x_3/3$, $x_4/3$ from $S_3 \setminus S_2$, and adds $\frac{(90-60)\times 0.5}{3} = 5$ into the $scoreList$ of each of the three s-assignments. Then it extracts $x_1/3$, $x_2/2$, $x_3/3$ and $x_4/3$ from $S_3 \setminus S_1$, and adds 5 into the $scoreList$ of each of the four s-assignment. Finally, the $scoreList$ of $x_5/4$, $x_6/4$ and $x_7/4$ contain 1.5 and 7.2, so the average score is 4.35. The $scoreList$ of $x_3/3$ and $x_4/3$ contain 7.2, 5 and 5, so the average score is 5.73. The $scoreList$ of $x_1/2$ contains 1.5. The $scoreList$ of $x_1/3$ contains 5 and 5, so the average score is 5. The $scoreList$ of $x_2/2$ contains 5. After sorting the s-assignments by the descending order of the average scores, it creates a $paList$: $x_3/3$, $x_4/3$, $x_1/3$, $x_2/2$, $x_5/4$, $x_6/4$, $x_7/4$, $x_1/2$.

i	*sol*	*obj*	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
1	S_4	54	2	2	3	3	4	4	4	1
2	S_3	60	3	2	3	3	1	1	1	1
3	S_2	90	2	2	1	1	1	1	1	1
4	S_1	100	1	1	1	1	1	1	1	1

The i column is the index of the solutions in $solQueue$.

Table 1: The solutions in $solQueue$ of the example

Discussion and Related Work

Our approach explores the sub-search-spaces specified by different partial assignments. The procedure seems to simulate Large Neighbourhood Search (LNS) (Shaw 1998) that fixes the assignment of some variables and searches in a sub-search-space to find better solutions in an iterative manner. LNS selects a set of variables and fix them to their values in the current best solution. Both GPA and RGPA provide a strategy to generate a partial assignment extracted from different solutions. When utilizing the partial assignment, the variables involved in the partial assignment are not fixed. They are instantiated one by one at the top of the search tree, so that the corresponding sub-search-space will be explored first and search is not limited in the sub-search-space. If the sub-search-space contains no better solution and the restart limit is not reached, the search will continue to explore the rest of the search space upon backtracking. While the partial assignment fixed by LNS is consistent, the partial assignment generated by our approach may be inconsistent. This is not a problem when utilizing the partial assignment, because only one s-assignments is used at each step and lines 3-4 in Algorithm 4 will skip the inconsistent s-assignments.

Solution-Based Phase Saving (SBPS) (Demirovic, Chu, and Stuckey 2018) is an efficient value-selection strategy for COPs. It gives priority to the value v for each variable x if x/v is in the current best solution. If v is still in $dom(x)$, it selects x/v; otherwise, it makes the selection by an underlying value heuristic. SBPS also simulates LNS. The phase to

be saved in the next feasible solution is usually determined by variable heuristics (Michel and Van Hentenryck 2012; Palmieri and Perez 2018), because the variables selected earlier are placed at higher level of the search tree, so that they have a larger chance to keep their current values in next feasible solution. Our approach finds good partial assignment first, and then place them at the top of the search tree. The s-assignments in the partial assignments have larger chance to be kept in the next feasible solution.

Bound-Impact Value Selector (BIVS) (Fages and Prud'Homme 2017) chooses the value of a variable leading to the best objective bound after a propagation. For each value v in $dom(x)$, BIVS applies a decision x/v in a copy of the COP and calculates the reduction effects on the objective through propagation, and then selects the value leading to the best bound of the objective. SBPS employing BIVS as the underlying value heuristic has been used as the default value-selection heuristic in some black-box COP solvers, such as Choco (Prud'homme, Fages, and Lorca 2017). Our approach works as a heuristic to make decisions at the top levels of the search trees. It can be combined with any search strategy including SBPS and BIVS.

Frequent Pattern Mining-based Search (FPMS) (Li et al. 2020) finds good subtrees containing high-quality solutions for loosely constrained optimization problems. It employs a random search procedure to generate sample solutions to represent the feature of the solution space and uses frequent pattern mining to find a good partial assignment. If the problem is tightly constrained, then the sampling procedure of FPMS will be costly. Thus, it is not suitable for tightly constrained problems. Besides, it finds only one good search entrance and has not been incorporated with restart techniques. Our approach analyses the intermediate solutions found by BBS to find good partial assignments and does not need a sampling procedure. Thus our approach is suitable for all kinds of constraint optimization problems. Besides, our approach updates the good partial assignments as the current best solution evolves, so that it provides multiple search entrances to incorporate with restart techniques.

Our approach enjoys benefits of both Evolutionary Computation (EC) (Back, Fogel, and Michalewicz 1997) and BBS. People customize different strategies in EC to solve specific optimization problems. We adapt the idea of maintaining a solution set to extract useful information to find good partial assignments for general constraint optimization problems. Whenever a new solution is found, it will be used to update the partial assignment for the next restart run. In our approach, some components of EC work as a part of the search strategy in BBS for completely solving COPs.

Experiments

In the following, GPA and RGPA denote our approach with GPA and RGPA procedures generating partial assignments respectively. We incorporate GPA and RGPA into state of the art search strategy (baseline) to examine how our approach improves the baseline black-box COP solver.

The Environment. The environment is JDK8 under CentOS 6.4 with Intel Xeon CPU E7-4820@2.00GHz processor and 58 GB RAM. The experiments were run in Choco solver (version 4.10.8) (Prud'homme, Fages, and Lorca 2017). [1] We have used a unique random seed 0 throughout the experiments and the timeout is set to 2 hours.

Benchmark. The experiments were run with the MiniZinc benchmark suite from https://github.com/MiniZinc/minizinc-benchmarks. The instances are flattened offline to FlatZinc format to use the global constraints provided by Choco. After eliminating those large instances that cannot be flattened in 1 hour, we have 71 MiniZinc models of 1292 instances. To balance the effect of instance set size, we randomly pick 10 instances for each MiniZinc model. If a model contains less than 10 instances, we select them all.

Metrics. The performance of the compared search strategies are measured by solution quality and the number of instances where search completes. It is not easy to define a unique measurement for solution qualities, because the objectives of the instances vary greatly and there are minimization and maximization problems. We compare the strategies in pairs to see which one finds better solutions. In the following tables, the row #C presents the numbers of instances where search completes. In each cell containing two numbers separated by a colon, we present the numbers of instances where the approach with the row id finds better solutions versus that of the approach with the column id. The instances where the two compared strategies find solutions with same objective are not counted. The last column presents the number of wins for each strategy with a row id.

Baseline. We have used SBPS (Demirovic, Chu, and Stuckey 2018) as the value heuristic and BIVS (Fages and Prud'Homme 2017) as the underlying heuristic for SBPS. The Luby sequence (Luby, Sinclair, and Zuckerman 1993) based restart strategy which achieves a good balance between frequent and extended restarts is employed. The initial restart cutoff is set to n, the number of variables. The default nogood recording (Lecoutre et al. 2007) technique in Choco is also used. With the above settings, we have tested three variable-selection heuristics including $dom/wdeg$ (the default variable heuristic in Choco) (Boussemart et al. 2004), Activity-Based Search (ABS, the variable heuristic which is recommend to be combined with SBPS) and ABSO (the variable heuristic combining ABS with an objective-based function, which is designed for constraint optimization problems) (Palmieri and Perez 2018). Table 2 presents the results got in 2 hours of 577 random selected instances. The cells containing two numbers separated by a colon present the comparison between two methods, e.g., the cell containing 174:54 means the search strategy with ABS finds a solution better than that found by the strategy with $dom/wdeg$ in 174 instances, and the latter finds better solutions in 54 instances. Thus ABS wins. It is shown that ABS performs better than the others in both completing search and solution quality. Thus we have used ABS as the variable-selection strategy. All the above techniques put together is a state of the art black-box COP solver containing ABS, SBPS, BIVS, Luby restart and nogood recording, which is used as the baseline (marked by Base in the results) in the experiments.

[1] The source code of RGPA implemented in Choco 4.10.8 is available at https://github.com/lihb905/rgpa.

	ABS	$\frac{dom}{wdeg}$	ABSO	win
ABS	-	**174**:54	**126**:91	2
$\frac{dom}{wdeg}$	54:**174**	-	98:**163**	0
ABSO	91:**126**	**163**:98	-	1
#C	**321**	244	248	-

Table 2: The baseline search strategies

Results. Our approach is designed for the optimization procedure. It starts to work after the second feasible solution is found. So we eliminated the infeasible instances and those instances where the baseline cannot find the second feasible solution in 2 hours. The tables contain the results of 480 instances randomly selected from the remaining instances.

Firstly, we investigate the influence of the parameter m in RGPA, which is the maximum number of solutions in $solQueue$. The performance of RGPA with m set to 10, 20, 30, 50 and unlimited (marked by U) is investigated. Table 3 presents the results. The cells in the table present the same results as those in Table 2, e.g., the cell containing 41:38 means the RGPA with m = 20 finds a solution better than that found by the RGPA with m = 10 in 41 instances, and the latter finds better solutions in 38 instances. Thus RGPA with m = 20 wins. It is shown that the parameter 20 outperforms all the others, 30 is outperformed by 20 only, 10 is outperformed by all the others and unlimited is not a good choice either. The results indicate that the parameter should not be too small or too large. A small number of solutions may not provide sufficient information and a large number of solutions may keep the information of the earlier found solutions, which should be discarded. We believe that there is no best parameter for all problems, and 20 is recommended here for a black-box solver. It is shown the performance of different parameters are close in completing search. Although 20 is not the one completing search in the largest number of instances, it is good at finding better solutions. Thus, we set m to 20 in the following experiments.

Secondly, we compare GPA and RGPA with the baseline. The results are grouped by different time limits in seconds in Table 4. The OBJ rows present the same comparison as in Table 3. Note that in each cell we can compute the number u of instances where the compared strategies finds the same quality of solution since the sum of u and the two presented numbers is 480. From the comparison of solution qualities, we can see that RGPA performs better than the other two in all time limits and GPA also outperforms the baseline in all time limits. From the #C rows, we can see that the baseline strategy completes search in the largest number of instances in the short time limit of 600 seconds, but GPA completes search in 276 instances before timeout (7200 seconds). RGPA completes search in the largest number of instances in the other time limits. RGPA loses in 1 instance in completing search and outperforms the others in finding good solutions, and gets the best overall performance. We also present the result of comparing the number of better instances found in different time limits in a histogram in Figure 2. We can see a clear trend that as the time limit increases, the gap between RGPA and the baseline widens.

m	10	20	30	50	U	win
10	-	38:**41**	34:**43**	37:**43**	39:**42**	0
20	**41**:38	-	**30**:24	**32**:20	**32**:19	4
30	**43**:34	24:**30**	-	**25**:19	**29**:15	3
50	**43**:37	20:**32**	19:**25**	-	**18**:10	2
U	**42**:29	19:**32**	15:**29**	10:**18**	-	1
#C	**276**	275	**276**	274	274	-

Table 3: Experiments on parameter m of RGPA

Time Limit			Base	GPA	RGPA	win
600	OBJ	Base	-	66:**76**	58:**73**	0
		GPA	**76**:66	-	58:**69**	1
		RGPA	**73**:58	**69**:58	-	2
	#C		**215**	206	210	-
1200	OBJ	Base	-	62:**78**	56:**74**	0
		GPA	**78**:62	-	59:**69**	1
		RGPA	**74**:56	**69**:59	-	2
	#C		224	224	**226**	-
1800	OBJ	Base	-	60:**77**	52:**75**	0
		GPA	**77**:60	-	58:**75**	1
		RGPA	**75**:52	**75**:58	-	2
	#C		230	228	**234**	-
3600	OBJ	Base	-	50:**82**	46:**78**	0
		GPA	**82**:50	-	54:**71**	1
		RGPA	**78**:46	**71**:54	-	2
	#C		242	244	**245**	-
5400	OBJ	Base	-	45:**85**	43:**82**	0
		GPA	**85**:45	-	53:**62**	1
		RGPA	**82**:43	**62**:53	-	2
	#C		251	**253**	**253**	-
7200	OBJ	Base	-	43:**84**	40:**84**	0
		GPA	**84**:43	-	52:**61**	1
		RGPA	**84**:40	**61**:52	-	2
	#C		269	**276**	275	-

Table 4: The overall results

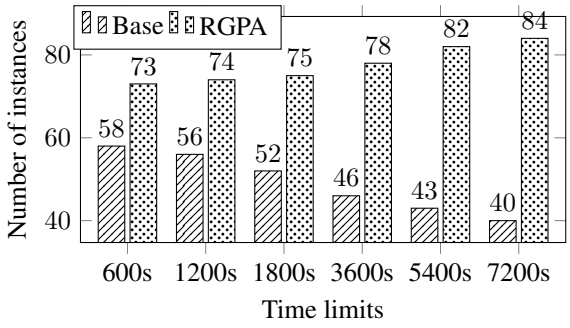

Figure 2: Comparing RGPA with the baseline.

Next, we examine the effect of different settings of the restart strategy. The latest Choco solver (version 4.10.8) provides a strategy that resets the restart sequence after a solution is found. Whether resetting the sequence or not may

	RGPA(IM)	RGPA(D)	RGPA(U)	Base(D)	Base(U)	win
RGPA(IM)	-	**70**:50	**72**:48	**84**:40	**81**:46	4
RGPA(D)	50:**70**	-	**67**:53	**77**:48	**66**:60	3
RGPA(U)	48:**72**	53:**67**	-	**81**:54	**62**:55	2
Base(D)	40:**84**	48:**77**	54:**81**	-	57:**71**	0
Base(U)	46:**81**	60:**66**	55:**62**	**71**:57	-	1
#C	275	275	**277**	269	269	-

Table 5: Experiments on different settings of restart

Problems		RGPA	FPMS
OpenStacks (48 instances)	Average Objective	**10.5**	11.73
	# Better	**48**	26
	Average Time	**31**	481
	#C	**29**	19
Trucking (15 instances)	Average Objective	**470**	1771
	#Better	**15**	5
	Average Time	**206**	3202
	#C	**5**	**5**

Table 6: RGPA verse FPMS

affect the performance of all the search strategies. So in Table 5, we compare our approach with the baseline with resetting (the default setting, marked by (D)) and unresetting (marked by (U)) the sequence. Only the strategy RGPA(IM) uses our strategy that sets the temporary restart cutoff. It is shown that RPGA always outperforms the baseline strategy in both resetting and unresetting the sequence. Adding our strategy that uses the temporary restart cutoff into RGPA(D), RGPA(IM) outperforms all the others in finding better solutions. Although RGPA(IM) completes search in less instances than RGPA(U), it outperfroms the baseline.

RGPA is known to be more superior than FPMS as discussed in the last section. In particular, FPMS is applicable only to loosely constrained problems. As such, we selected 2 among the 71 MiniZinc benchmarks in our experiment, with which FPMS can be used successfully, for comparison between RGPA and FPMS. The results are shown in Table 6. The #Better rows show the number of instances where the corresponding method finds better solutions in 7200 seconds and the Average Time rows show the time cost of finding the best solution. It is shown that RGPA costs significantly less time to find better solutions than FPMS. RGPA also completes search in more instances in the OpenStacks problem.

Conclusion

In this paper, we propose an approach to find good partial assignments for general constraint optimization problems. Our approach extracts good partial assignments from different intermediate solutions found during branch-and-bound search. Exploring the sub-search-space specified by the good partial assignments, our approach helps BBS finds better solutions and complete search in more instances. It significantly improves the state of the art search strategy for general constraint optimization problems.

Acknowledgements

We thank the anonymous referees for their constructive comments. This work is supported by the National Natural Science Foundation of China under Grant 62276060 and Natural Science Foundation of Jilin Province under Grant 20210101470JC. In addition, we acknowledge the financial support of a General Research Fund (RGC Ref. No. CUHK 14206321) by the University Grants Committee, Hong Kong.

References

Back, T.; Fogel, D. B.; and Michalewicz, Z. 1997. *Handbook of Evolutionary Computation*. GBR: IOP Publishing Ltd., 1st edition. ISBN 0750303921.

Bessiere, C. 2006. Constraint Propagation. In Rossi, F.; van Beek, P.; and Walsh, T., eds., *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, 29–83. Elsevier.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting Systematic Search by Weighting Constraints. In *Proc. ECAI'04*, volume 16, 146–150.

Demirovic, E.; Chu, G.; and Stuckey, P. J. 2018. Solution-based phase saving and large neighbourhood search. In *Proc. CP'18*, 99–108. Springer.

Fages, J. G.; and Prud'Homme, C. 2017. Making the First Solution Good. In *Proc. ICTAI'17*. IEEE.

Gomes, C. P.; Selman, B.; and Kautz, H. 1998. Boosting Combinatorial Search Through Randomization. In *Proc. AAAI'98*, 431–437.

Lecoutre, C.; Sais, L.; Tabary, S.; and Vidal, V. 2007. Recording and Minimizing Nogoods from Restarts. *Journal on Satisfiability, Boolean Modeling and Computation*, 1: 147–167.

Li, H.; Lee, J. H.; Mi, H.; and Yin, M. 2020. Finding Good Subtrees for Constraint Optimization Problems Using Frequent Pattern Mining. In *Proc. AAAI'20*, 1577–1584.

Li, H.; Yin, M.; and Li, Z. 2021. Failure Based Variable Ordering Heuristics for Solving CSPs. In *Proc. CP'21*, 9:1–9:10.

Luby, M.; Sinclair, A.; and Zuckerman, D. 1993. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4): 173–180.

Michel, L.; and Van Hentenryck, P. 2012. Activity-based Search for Black-box Constraint Programming Solvers. In *Proc. CPAIOR'12*, 228–243. Springer.

Palmieri, A.; and Perez, G. 2018. Objective as a Feature for Robust Search Strategies. In *Proc. CP'18*, 328–344. Springer.

Prud'homme, C.; Fages, J.-G.; and Lorca, X. 2017. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S.

Shaw, P. 1998. Using constraint programming and local search methods to solve vehicle routing problems. In *Proc. CP'98*, 417–431. Springer.

Walsh, T. 1999. Search in a Small World. In *Proc. IJCAI'99*, 1172–1177.