Finding Good Subtrees for Constraint Optimization Problems Using Frequent Pattern Mining

Hongbo Li,[1] **Jimmy H.M. Lee,**[2*] **He Mi,**[1] **Minghao Yin**[1*]
[1]School of Information Science and Technology, Northeast Normal University, Changchun, China
{lihb905, mih221, ymh}@nenu.edu.cn
[2]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
jlee@cse.cuhk.edu.hk

Abstract

Making good decisions at the top of a search tree is important for finding good solutions early in constraint optimization. In this paper, we propose a method employing frequent pattern mining (FPM), a classic datamining technique, to find good subtrees for solving constraint optimization problems. We demonstrate that applying FPM in a small number of random high-quality feasible solutions enables us to identify subtrees containing optimal solutions in more than 55% of problem instances for four real world benchmark problems. The method works as a plugin that can be combined with any search strategy for branch-and-bound search. Exploring the identified subtrees first, the method brings substantial improvements for four efficient search strategies in both total runtime and runtime of finding optimal solutions.

Introduction

Constraint programming (CP) is a powerful paradigm for solving discrete combinatorial optimization problems. Depth-first branch-and-bound search (BBS) is a complete method for solving constraint optimization problems (COPs). BBS can be divided into two phases. The first phase finds an optimal solution and the second phase proves the optimality. If a better solution is found earlier, BBS could prune the search space more efficiently. Thus, much work has been done on devising search strategies to speed up the first phase. For instance, Chu and Stuckey use machine learning to design search strategies (Chu and Stuckey 2015). Counting-Based Search (Pesant 2016) uses cost-based solution density to guide search. Objective-Based Selector (OBS) (Palmieri and Perez 2018) uses objective variables as a feature for decisions within search strategies. To find good solutions early, the Bound-Impact Value Selector (BIVS) (Fages and Prud'Homme 2017) chooses the value of a variable that would lead to the best objective bound after a propagation.

We note also the multitude of recent work in integrating CP and datamining/machine learning. On the one hand, the CP-based approaches offer a trade-off between generality and efficiency for some classic datamining problems, such as clustering (Dao, Duong, and Vrain 2017) and frequent pattern mining (Guns, Nijssen, and Raedt 2011; Lazaar et al. 2016; Schaus, Aoga, and Guns 2017; Guns et al. 2017). On the other hand, a number of studies have shown the benefits of applying machine learning in constraint solving (Epstein and Petrovic 2007; Xu, Stern, and Samulowitz 2009; Loth et al. 2013; Hurley et al. 2014; Chu and Stuckey 2015; Balafrej, Bessiere, and Paparrizou 2015; Bachiri et al. 2015; Xia and Yap 2018; Cappart et al. 2019). In addition, some datamining techniques have been applied in heuristic methods for solving some combinatorial problems (Samorani and Laguna 2012; Zhou, Hao, and Duval 2017).

This work proposes to employ *frequent pattern mining* (FPM), a classic datamining technique, to speed up the first phase of BBS for solving COPs. The intuition behind the proposed method is that the frequent patterns extracted from high-quality solutions identify promising regions in the search space that are worthy of being explored first. The method runs a FPM algorithm to mine useful patterns from a set of high-quality feasible solutions, called *probes*, which are generated by a random probing procedure. After the mining, the best patterns are selected according to a measurement considering the qualities of the probes containing the pattern, the support number and pattern length, and the top patterns are combined to generate a partial assignment. Then we run BBS exploring first the subtree under this partial assignment before the rest of the complete search tree. The method is not a search heuristic per se, and can be readily combined with any search strategies for BBS. It identifies and zooms directly into a subtree with high possibility of containing optimal solutions. We expect the method to work well on loosely constrained optimization problems, where feasible solutions are easy to find by probing.

We have tested the method on the *Uncapacitated Warehouse Location Problem* (UncapWLP), the *Multidimensional-Knapsack Problem* (MKnap), the *Traveling Tournament Problem with Predefined Venues* (TTPPV) and the *Traveling Salesman Problem* (TSP). The results show that the method finds the subtrees involving optimal solutions in over 55% of benchmark instances.

[*]Corresponding authors

Improvements in both total runtime and runtime of finding optimal solutions are demonstrated when combining our method with four efficient search strategies which are different combinations of OBS, BIVS and some other popular heuristics.

Background

This work crosses two disciplines. We employ the method of *data science* to help *constraint solving and optimization*. In the following, we give basic definitions of Constraint Programming and Frequent Pattern Mining.

Constraint Programming

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 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*, or otherwise a *partial assignment*. An assignment to a single variable is called a singleton-assignment, denoted by s-assignment. The objective $\mathcal{F}$ maps every complete assignment to a real number. A complete assignment $\mathcal{S}$ that satisfies all constraints is a *feasible solution* to $\mathcal{P}$. Without loss of generality, we consider here minimization and a feasible solution $\mathcal{S}^*$ is an *optimal solution* if $\mathcal{F}(\mathcal{S}^*) \leq \mathcal{F}(\mathcal{S})$, where $\mathcal{S}$ is any other feasible solution to $\mathcal{P}$. In case of maximization, optimal solutions are defined simply with $\leq$ replaced by $\geq$. The *solution space* of $\mathcal{P}$ contains all complete assignments. Given a partial assignment $\mathcal{A}$. The *sub-space* of $\mathcal{A}$ is the set of all complete assignments that can be extended from $\mathcal{A}$.

A COP $\mathcal{P}$ can be solved by BBS augmented with constraint propagation. BBS explores the solution space with depth-first backtracking tree search resulting in a search tree. Whenever a new feasible solution $\mathcal{S}$ is found, $\mathcal{S}$ is recorded as the *best solution found so far* and $\mathcal{F}(\mathcal{S})$ is used to bound subsequent search to ensure the next feasible solution found must be better than $\mathcal{S}$. Search continues until no more feasible solutions are found after exhausting the search space. The last solution found is an optimal solution of $\mathcal{P}$. Given a partial assignment $\mathcal{A}$. The *subtree under* $\mathcal{A}$ is the result of exploring the sub-space of $\mathcal{A}$ using BBS. An *optimal subtree* is any subtree of $\mathcal{P}$'s search tree containing an optimal solution.

Frequent Pattern Mining

FPM was first proposed for mining transaction databases to analyze market baskets (Agrawal, Imielinski, and Swami 1993). Let $\mathcal{I}$ be the set of all items. A set $I \subseteq \mathcal{I}$ is an *itemset*. The *length* of I is $|I|$. A *l-itemset* (or *l-pattern*) has length l. A *transaction* is an itemset. Let $\mathcal{T}$ be a transaction database (a set of transactions), where each $T_i \in \mathcal{T}$ is an itemset over $\mathcal{I}$. A transaction T *supports* an itemset I if $I \subseteq T$. The set of all transactions supporting I is denoted by $cover(I)$, and $|cover(I)|$ is the support number of I. An itemset I is a *frequent pattern* if its support number is at least a specified *support threshold* sup.

The FPM problem aims to find all the frequent patterns with a specified support threshold in a given transaction database. There are three basic FPM methodologies: *Apriori* (Agrawal and Srikant 1994), *FP-growth* (Han, Pei, and Yin 2000) and *Eclat* (Zaki 2000).

We will apply frequent pattern mining method in finding good subtrees. Thus, each item is a s-assignment. An itemset is either a partial assignment or a complete assignment. A transaction is a feasible solution. A transaction database is a set of feasible solutions. In this work, mining is not the major challenge, so any FPM algorithm can be employed. We adopt *Eclat* in the method since the transactions information will be used to select the best frequent patterns and it is easy for *Eclat* to track the transactions supporting the mined frequent patterns.

The Motivation

Our work is based on the intuition that a sub-space containing an optimal solution usually involves some high-quality solutions. On the contrary, if we find a sub-space involving a number of high-quality solutions, it may involve an optimal solution. Even the sub-space does not contain an optimal solution, it may contain some other high-quality solutions. Thus, the frequent patterns extracted from high-quality solutions identify promising regions in the search space that are worthy of being explored first.

To check whether the intuition is true at least on some problems, we did an experiment on an instance of the *Multidimensional-Knapsack Problem* with 20 items and 422601 feasible solutions, which is the fourth instance in mknap1 in the OR-Library[1]. This is the largest instance of mknap1, for which we can generate all feasible solutions within 4 hours. We first find all feasible solutions of the instance, and consider the top 42260 solutions (top 10% according to the objectives of the solutions) as high-quality solutions and do frequent pattern mining in the 42260 solutions with support threshold 21130. After finding all the frequent patterns, the one with largest $basic(I) \times quality(I)$ is selected, and it is an optimal subtree. So the intuition seems to work on at least some problems.

The *basic* score and *quality* score of each frequent pattern I are defined as follows:

$$basic(I) = |cover(I)| \times |I| \tag{1}$$

A frequent pattern with more supports and larger length is usually preferred, because a frequent pattern with more support is more promising and a larger frequent pattern leads the search to a smaller search space. However, a frequent pattern with larger length usually has a smaller number of supports. We want these conflicting quantities to be both large, so that the *basic* score tries to balance support number and pattern length, where the product is the larger the better. It is an estimation of the effectiveness of a subtree.

[1]http://people.brunel.ac.uk/~mastjjb/jeb/orlib/mknapinfo.html

$d\%$	0.01%	0.02%	0.03%	0.04%	0.05%	0.06%	0.07%	0.08%	0.09%
#samples	42	84	126	169	211	253	295	338	380
opt rates	33%	35%	44%	51%	62%	71%	76%	80%	90%
$d\%$	0.1%	0.2%	0.3%	0.4%	0.5%	0.6%	0.7%	0.8%	0.9%
#samples	422	845	1267	1690	2113	2535	2958	3380	3803
opt rates	86%	98%	100%	99%	100%	100%	100%	100%	100%

Table 1: Results of mining in samples of all solutions

$$quality(I) = \sqrt{aveSol(I) \times bestSol(I)} \qquad (2)$$

The *quality* score considers the average quality of the solutions ($aveSol(I)$) and the quality of the best solution ($bestSol(I)$) supporting the pattern I. The square root in the formula is to make the number smaller. The quality score gives a raw estimation of the quality of a subtree, and should be the smaller the better for minimization problems and the larger the better for maximization.

From a statistical perspective, if we have sufficient random samples of all solutions, they can reflect properties of the whole feasible solution space. So the second question is whether we can find such structures by analyzing a set of samples of all the feasible solutions. Then we performed another test. We randomly select $d\%$ samples (#samples is the number of samples) from all the 422601 feasible solutions and do the mining on the top 10% of the samples which are considered as high-quality solutions. The results are shown in Table 1. For each $d\%$, we run 100 independent tests that generate 100 different sample sets with random seed from 0 to 99. The "opt rates" row of the table presents the percentages of the best frequent patterns that are optimal subtrees. For example, with $d\% = 0.3\%$, we have 1267 sample solutions, the mining is done in the top 126 samples, and the best frequent pattern found is an optimal subtree for each of the 100 different sample sets generated with different seeds. The results show that, for this instance, we can do analysis in about 1000 random sample solutions to find the structure of the solution space containing 422601 feasible solutions. Therefore, on some problems, we can find an optimal subtree by mining in a set of high-quality solutions which are selected from a set of random samples of all the feasible solutions.

Frequent Pattern Mining for Solving COPs

We propose *Frequent Pattern Mining based Search (FPMS)* for solving COPs. The method applies frequent pattern mining in a set of high-quality random solutions to find a good subtree. BBS explores the good subtree first, and then continues exploring the rest of the entire search tree. The framework of FPMS is presented in Algorithm 1.

Generating Transaction Database

It is impractical to find all feasible solutions and select some random ones, so we achieve random sampling using a random probing procedure. The PROBING procedure at line 1 of Algorithm 1 generates t distinct feasible solutions with a simple backtracking procedure by selecting a random

Algorithm 1: FPMS

```
Input: a COP
Output: an optimal solution
1 $\mathcal{T}_{r\%} \leftarrow$ PROBING($t$, $r\%$);
2 $topFPs \leftarrow$ MINING($\mathcal{T}_{r\%}, sup, min, max, num$);
3 $subtree \leftarrow$ GENERATESUBTREE($topFPs$);
4 start BBS with $subtree$ as initial assignment;
```

variable x and assigning it with a random value in $dom(x)$ followed by constraint propagation at each search node. The random search procedure backtracks whenever a failure is detected and is restarted every time a feasible solution is found. Every new solution is recorded. Note that no constraints are added at restart to forbid a solution being found more than once during probing, but such duplicate solutions will be recorded only once. Then it returns the top $r\%$ good solutions with the best objective values as the transaction database $\mathcal{T}_{r\%}$.

Finding A Good Subtree

At line 2 of Algorithm 1, the MINING procedure adopts and adapts the classic ECLAT algorithm to find all frequent patterns with support numbers greater than or equal to sup and lengths between min and max.

A subtree specified by an itemset I has a higher chance of being an optimal subtree than a subtree specified by an itemset I' which is a superset of I, since the latter subtree is involved in the former one. So the longer frequent patterns have higher risks of not being optimal subtrees. On the contrary, too short a frequent pattern may not be helpful to the search. From the analysis in previous section, we know that with the same length a frequent pattern with more support is preferred, therefore, the MINING procedure sets a starting support threshold sup to a relative large number and runs the ECLAT algorithm. If sup is too large for finding frequent patterns, sup is recursively reduced by half. The procedure repeats until either frequent patterns are found or sup is less than a minimum threshold. If no frequent patterns are found eventually, we run a normal BBS.

After finding all frequent patterns, the top num ones are selected by considering the *basic* score and *quality* score introduced in last section. We assume the objective value $\mathcal{F}(S) \geq 0$. In solving a minimization problem, we would like to select a frequent pattern I that has smaller $quality(I)$ and larger $basic(I)$. So the MINING procedure returns the top num ones with the smallest $\frac{quality(I)}{basic(I)}$ for minimization problems. In solving maximization problems, we want I with both larger $quality(I)$ and larger $basic(I)$. So it returns the top num ones with largest $quality(I) \times basic(I)$.

```
Algorithm 2: GENERATESUBTREE($topFPs$)
  Input: a set of frequent patterns.
  Output: a FIFO queue $subtree$
1 $subtree \leftarrow$ an empty queue;
2 for each s-assignment appears in $topFPs$ do
3     count the number of its appearance in $topFPs$;
4 for each frequent pattern $FP$ in $topFPs$ do
5     sort the s-assignments in $FP$ with descending
        ordering of their appearances;
6     for each s-assignment $x/v$ in $FP$ do
7         if $subtree$ contains no s-assignment of $x$ then
8             add $x/v$ at the end of $subtree$;
9 return $subtree$;
```

Finally, the GENERATESUBTREE procedure combines the top num frequent patterns to generate a subtree. It first counts the number of each assignment appeared in $topFPs$. Then at line 4 of Algorithm 2, it visits the frequent patterns with the ascending (descending) order of the measurement for minimization (maximization) problems. For each FP, it visits the s-assignments with the descending order of the numbers of their appearances in $topFPs$, because we want the most promising s-assignment to be assigned first.

Here is an example. Given $topFPs$ involving three frequent patterns $FP_1 = \{x_1/a, x_2/b, x_3/c\}$, $FP_2 = \{x_2/b, x_3/c, x_4/d_1\}$ and $FP_3 = \{x_2/b, x_4/d_2, x_5/e\}$, where FP_1 is better than FP_2 which is better than FP_3 with respect to the measurement. The appearances of the s-assignments x_1/a, x_2/b, x_3/c, x_4/d_1, x_4/d_2, x_5/e are 1, 3, 2, 1, 1, 1 respectively. Note that x_4 takes different values in FP_2 and FP_3. The algorithm will visit FP_1 first. After sorting, the s-assignments in FP_1 are x_2/b, x_3/c, x_1/a. So x_2/b is the first s-assignment added into $subtree$. x_3/c is the second and x_1/a is the third. Then FP_2 will be visited. The s-assignments in FP_2 are x_2/b, x_3/c, x_4/d_1. And x_2/b will be tried first, but the variable x_2 is already contained in $subtree$, so it will not be added into $subtree$ again (line 7 of Algorithm 2), neither is x_3/c. The fourth one to add into $subtree$ is x_4/d_1. Finally, FP_3 will be visited. After sorting, the s-assignments in FP_3 are x_2/b, x_4/d_2, x_5/e, but x_2/b and x_4/d_2 will not be added into $subtree$, because their variables are already contained in $subtree$ (line 7 of Algorithm 2). And x_5/e is the fifth one added into $subtree$. So the GENERATESUBTREE procedure returns a queue with x_2/b, x_3/c, x_1/a, x_4/d_1, x_5/e. Assuming that x_1/a is not contained in any optimal solution and the other four are contained in a same optimal solution. Then the subtree specified by the queue has depth 5, but the optimal subtree has depth 2.

Branch-and-Bound Search Strategy

At line 3 of Algorithm 1, BBS is invoked to explore the subtree specified by $subtree$ first, and then continues exploring the rest of the entire search tree. This is a normal depth-first BBS except that the initial assignments are specified. A simple strategy presented in Algorithm 3 makes FPMS work as a plugin that can be combined with any standard search strategy heu. Initially, at each node of a search tree, the algorithm selects and removes a variable-pair in $subtree$. This ensures that the subtree specified by $subtree$ will be explored first. Each variable-pair in $subtree$ is used only once, so that the search will not be limited in the subtree and the whole search tree will be explored. After $subtree$ is exhausted, the standard search strategy heu takes over.

```
Algorithm 3: SEARCHSTRATEGY($subtree$, $heu$)
  Input: a queuet of variable-value pairs; a search
           strategy $heu$
  Output: a variable-value pair
1 if $subtree$ is not empty then
2     select and remove the first $x/v$ from $subtree$;
3     return $x/v$;
4 return a variable-value pair selected by $heu$;
```

Discussions

In frequent pattern mining problems, a very small support threshold usually gets a number of frequent patterns and a very large support threshold may get no frequent patterns. In most cases, the number of frequent patterns increases with decreasing the support threshold sup. If a sup value fails to find a frequent pattern, then sup - 1 will likely have no, or very few frequent patterns which are short. Reducing sup by half instead may help to find proper frequent patterns quickly. Although the strategy has a risk of finding a large number of frequent patterns suddenly, this is not a problem since we have a measurement to select the best frequent pattern from all the discovered patterns. For instance, the ECLAT algorithm finds no frequent pattern with sup = 20. Assuming that there is a useful frequent pattern I_1 with length 5 and support number 14. If sup is recursively reduced by 1, then we may find a frequent pattern I_2 with length 2 and support number 18. I_2 is not useful since it may be too short to speed up the search. If sup is reduced by half, then the second round runs the ECLAT algorithm with sup = 10. Both I_1 and I_2 will be found, and the best one will be selected by the measurement.

In selecting the best frequent pattern, we select the largest $quality(I) \times basic(I)$ for maximization but the smallest $\frac{quality(I)}{basic(I)}$ for minimization. One might wonder why we cannot simply select the largest $-quality(I) \times basic(I)$ for minimization instead. This is because the largest $-quality(I) \times basic(I)$ does not get the best frequent pattern for minimization problem, since we want a frequent pattern with a larger $basic(I)$ but for the same $quality(I)$, that would give a larger $-quality(I) \times basic(I)$ with a smaller $basic(I)$.

Constraint programming has been successfully applied in finding frequent patterns. One may wonder about why CP techniques are adopted for the mining procedure. First, finding a good subtree for a COP and solving the COP cannot be easily combined into one problem. If we use CP to find frequent patterns, we still need to solve the two problems separately. Second, the aim of this work is to show FPM can help constraint solving. The classic ECLAT algorithm is sufficiently efficient for this purpose. Thirdly, a closed or maximal frequent pattern seems to be more relevant for FPMS, but finding such patterns with CP is another optimization problem which might be costly. Our target is not to spend too much time on the mining, because we cannot practically ensure the found subtree is optimal. However, utilizing closed or maximal frequent patterns is interesting future possibility.

Some complete search strategies may combine heuristic search with BBS, e.g. using heuristic search to find an initial high-quality solution quickly before running BBS, which gives a good initial upper bound. The idea of utilizing the initial upper bound can be combined with FPMS. Besides, one may want to use the initial high-quality solution as the search entrance for BBS. However, it is not easy to decide which s-assignments in the initial solution should be used at the top of the search tree. Our proposal is better since (a) we are mining from a set of high-quality solutions instead of relying on only one and (b) the mining procedure tells us exactly which s-assignments to use since they are frequent. BBS is a depth first strategy. If the top decisions are incorrect, it still needs huge efforts to finish the search.

One might argue that FPMS is similar to streamlining constraints (Gomes and Sellmann 2004) since both drive search to a specific area of the search tree. The main difference between the two methods is that FPMS is a complete method that it explores the whole search tree after exploring the specific area, whereas streamlining constraints destory completeness.

A major limitation of FPMS is that it is suitable for only loosely constrained optimization problems where feasible solutions are easy to find but an optimal solution is hard to find. This is because the probing procedure uses a random search strategy. If feasible solutions are hard to find, the random search strategy will cost much time to generate the transaction database. In order to use the probes to reflect the whole search space, the database should contain the solutions that are generated randomly. So FPMS relies on the probes generated by a random search. If we use some more effective search strategies, the transaction database may contain bias since such search strategies are usually guided by some measurements.

Experiments

The experiments were run in Choco 4.10 (Prud'homme, Fages, and Lorca 2017). The environment is JDK8 under CentOS 6.4 with Intel Xeon CPU E7-4820@2.00GHz processor and 58 GB RAM. We have tested FPMS on four different benchmarks:

	UncapWLP		MKnap		TTPPV		TSP	
$r\%$	L	N	L	N	L	N	L	N
1%	60	8	276	32	55	14	12	6
2%	64	8	268	32	42	11	11	6
5%	61	8	240	31	43	10	6	5
10%	49	8	244	32	48	11	7	5
20%	25	8	206	29	43	12	7	5
30%	24	8	192	29	41	11	4	4

Table 2: Results of optimal subtrees found in transaction databases generated with different top rate $r\%$ (with top number 10)

	UncapWLP		MKnap		TTPPV		TSP	
top number	L	N	L	N	L	N	L	N
1	28	8	122	29	39	11	6	4
5	43	8	206	32	48	12	6	4
10	49	8	244	32	48	11	7	5
20	48	8	299	32	48	11	6	4
50	48	8	322	32	48	11	6	4

Table 3: Results of optimal subtrees found with different top number of frequent patterns (with top rate 10%)

Problem	Total	Optimal Subtree Found?			
		Unknown	No	Yes	Y-Rate
UncapWLP	15	7	0	8	100%
MKnap	38	5	3	30	91%
TTPPV	20	0	9	11	55%
TSP	15	9	1	5	83%

Table 4: Number of instances where FPMS finds optimal subtrees

- UncapWLP: the 15 instances from OR-Library[2]. The decision variables are binary indicating whether a warehouse is open. The real numbers in the original data files are rounded down to integer numbers in the experiment.
- MKnap: the 55 instances of mknap1 and mknap2 from OR-Library. 17 instances where all strategies costs less than 1 second to complete search are eliminated. The decision variables are binary indicating whether an item is in the knapsack. The real numbers in the original data files are rounded down to integer numbers in the experiment.
- TTPPV: the balanced 8-team instances under the 5 predefined venues from CSPLib[3]. With the predefined venues, we tested 5 instances with circular distances and 15 instances with the real distances. The real distances are *NL8*, *Super8* and *Galaxy8* from OR-Library[4]. The modelling follows Pesant's description (Pesant 2012). Each decision variable $O[i][j]$ means the opponent of team i in round j.
- TSP: the 15 instances are real world problems from TSPLib[5]. The largest one we tested contains 52 cities,

[2]http://people.brunel.ac.uk/~mastjjb/jeb/orlib/uncapinfo.html
[3]http://www.csplib.org/Problems/prob068/
[4]https://mat.gsia.cmu.edu/TOURN/
[5]https://wwwproxy.iwr.uni-heidelberg.de/groups/comopt/

Search	Problem	10	100	1000	2000	3000	3600	Complete	Overall
DW BIVS	UncapWLP	**2** : 1	2 : **3**	**2** : 1	**3** : 2	3 : 3	3 : 3	8 : 8	**8** :7
	MKnap	8 : **17**	5 : **14**	2 : **14**	1 : **14**	1 : **14**	1 : **14**	23 : **26**	8 : **30**
	TTPPV	**9** : 8	6 : **14**	**7** : 6	2 : **4**	**1** : 0	0 : 0	12 : **14**	4 : **16**
	TSP	**3** : 0	**3** : 2	**7** : 1	**7** : 1	**7** : 0	**7** : 0	6 : 6	**11** : 4
ABSO BIVS	UncapWLP	2 : **3**	1 : **6**	0 : **4**	0 : **6**	0 : **7**	0 : **7**	8 : 8	4 : **11**
	MKnap	6 : **11**	3 : **10**	2 : **7**	1 : **6**	1 : **5**	0 : **6**	32 : 32	15 : **23**
	TTPPV	**10** : 7	3 : **15**	4 : **8**	**4** : 3	**1** : 0	0 : 0	**20** : 19	**14** : 6
	TSP	**2** : 1	2 : **3**	**7** : 3	**7** : 3	**7** : 3	**7** : 3	5 : **6**	**10** : 5
ABSO MinVS	UncapWLP	**2** : 1	1 : **7**	0 : **4**	0 : **6**	1 : **6**	1 : **6**	8 : 8	3 : **12**
	MKnap	4 : **20**	3 : **14**	1 : **12**	0 : **13**	0 : **13**	0 : **12**	25 : **30**	9 : **29**
	TTPPV	**10** : 7	4 : **12**	2 : **3**	0 : 0	0 : 0	0 : 0	20 : 20	10 : 10
	TSP	1 : 1	2 : **3**	1 : **9**	2 : **8**	2 : **8**	1 : **8**	5 : **6**	3 : **12**
ABSO OBSVS	UncapWLP	**3** : 1	1 : **7**	0 : **4**	1 : **5**	1 : **6**	1 : **6**	8 : 8	3 : **12**
	MKnap	5 : **17**	0 : **14**	0 : **11**	0 : **10**	0 : **10**	0 : **10**	28 : **33**	6 : **32**
	TTPPV	**11** : 8	5 : **11**	1 : **3**	0 : 0	0 : 0	0 : 0	20 : 20	9 : **11**
	TSP	1 : **2**	1 : **4**	3 : **7**	1 : **9**	1 : **8**	1 : **7**	6 : 6	1 : **14**

Table 5: The overall comparison of naive search and FPMS

since all the strategies cannot complete the search when city number is larger than 29. The modelling uses circuit, element and sum constraints. The decision variables are the successors variables.

Given d the maximum domain size of decision variables, the number m of solutions in $\mathcal{T}_{r\%}$ is set to $20 \times d$. The initial support threshold is set to 20. To avoid some lucky patterns being frequent, the minimum support threshold is set to 3. To make the found frequent patterns having suitable basic scores (useful lengths and sufficient supports), the max length is set to 5 and the min length of frequent patterns is set to 3 (or 2 if the number of variables in less than 20). Timeout of total runtime is 1 hour and timeout of MINING procedure is 1 minute. If the mining is timeout, it returns the best found frequent patterns so far. If not specified, the random seed is 0. Universally optimal parameters for all problems do not exist. Therefore, we did not spend too much time on tuning the parameters. The above parameters are set with simple intuitions and small experiments.

Firstly, we investigated two important parameters of FPMS, the top rate $r\%$ of selecting the good sample solutions and the top number of frequent patterns for generating the subtree. In Table 2, we present the results of optimal subtrees found in different transaction databases generated with different rates of top sample solutions. These databases contain the the same number of transactions, so a smaller rate needs a larger number of random sample solutions and more time to generate all random solutions. The L column is the total length of optimal subtrees for each problem and the N column is the number of instances where optimal subtrees are found. The results show that with the increasing of $r\%$, the total length decreases in general. This is also observed in the numbers of instances where optimal subtrees are found.

In Table 3, we present the results of optimal subtrees generated with different numbers of top frequent patterns. The results show that with the increasing of top number, the total length increases in general, so is the number of instances where optimal subtrees are found. When the top number is large, some frequent patterns with relatively low quality are selected and these patterns affect the ordering of the s-assignments. As a result, we lose some accuracy of measuring the s-assignments, and some wrong s-assignments are put at the head of $subtree$.

In the following, we select the parameters balance time cost, optimal subtree length and accuracy of ordering the s-assignments, e.g. top 10% sample solutions and top 10 frequent patterns. We summarize the numbers of instances where FPMS finds optimal subtrees in Table 4. The unknown instances are due to timeout. Excluding the unknown instances, the Y-Rate column presents the proportions of instances where optimal subtrees are found. The results show that FPMS finds the optimal subtrees for over 55% of the instances, by mining in a relatively small number ($20\times d$) of probes.

Secondly, to examine the efficiency of FPMS, the following variable heuristics are considered: the classic dom/wdeg (DW) (Boussemart et al. 2004) and ABSO, which is the combination of OBS (Palmieri and Perez 2018) and activity-based search(Michel and Van Hentenryck 2012). The following value heuristics are considered: BIVS (Fages and Prud'Homme 2017), the classic minimum value of domain (MinVS) and the OBS value selector (OBSVS)(Palmieri and Perez 2018). We have plugged FPMS into four search strategies including DW+BIVS, ABSO+BIVS, ABSO+MinVS and ABSO+OBSVS, which are recommended in (Fages and Prud'Homme 2017; Palmieri and Perez 2018). All search strategies without FPMS are collectively coined as naive search. To make the comparison fair, all the strategies are equipped with the sampling procedure and the best sample bound is used as the initial upper bound.

In Table 5, we present the overall results. The better one

software/TSPLIB95/tsp/

Instance	Optimal Subtree Length				FPMS				Naive Search		
	VarNum	FPM	Longest	Mean	Total	FBT	Obj	Mining	Total	FBT	Obj
UncapWLP-103	25	5	**8**	0.68	1391	**125**	893758	0.09	**999**	400	893758
UncapWLP-104	25	**8**	5	0.35	**430**	**13.6**	928918	0.10	769	52	928918
MKnap-WEISH22	80	**11**	7	0.91	**1962**	**486**	8947	1.38	2660	2301	8947
MKnap-WEISH23	80	**10**	7	0.89	**681**	**201**	8344	2.95	1358	996	8344
TTPPV-NL8d	56	**6**	3	0.24	3493	**21.2**	23269	0.16	**2905**	1678	23269
TTPPV-Galaxy8c	56	**4**	2	0.19	**2456**	**91**	1491	0.14	3469	1279	1491
TSP-gr24	24	1	1	0.1	1129	718	1272	0.74	**1123**	**652**	1272
TSP-fri26	26	0	**1**	0.07	**772**	**104**	**937**	0.47	3600	-	1016

Table 6: Detailed comparison results of representative hard instances.

in each comparison is in bold. In each cell with $n_1 : n_2$, n_1 is for naive search and n_2 is for FPMS. Columns 2-7 present that in different time limits (from 10 seconds to 3600 seconds), the numbers of instances where the corresponding strategy finds better solutions than the other. For instance, the 2 : 3 in the cell at row 2 column 4 means that, there are 2 instances of UncapWLP where the naive search of DW+BIVS finds better solutions in 100 seconds, and 3 instances where the FPMS of DW+BIVS finds better solutions. On the other instances of UncapWLP, they find same quality of solutions in 100 seconds. The Complete column presents the number of instances where the searches complete and the Overall column presents the overall comparison with respect to the following rules:

1. If both of them complete the search, then the one that costs less cpu time performs better; else go to 2.

2. If only one of them completes the search in 1 hour, then the complete one performs better; else go to 3.

3. If they find a same solution quality in 1 hour, then the one costs less cpu time performs better; else go to 4.

4. The one finds a better solution in 1 hour performs better.

It is shown that in 10 seconds time limit, we got mixed results, but in 100 and 1000 seconds time limits, FPMS performs better than naive search in general. on Mknap, FPMS clearly dominates naive search in all the four search strategies. On UncapWLP, FPMS outperforms naive search in three search strategies and loses slightly in DW+BIVS. On TTPPV, FPMS performs better in DW+BIVS and ABSO+OBSVS, and it wins slightly in ABSO+MinVS, but it is outperformed in ABSO+BIVS. On TSP, there is a clear trend that naive search performs better in the two strategies with BIVS and FPMS performs better in the other two strategies. This is because BIVS is efficient for TSP, as in shown in (Fages and Prud'Homme 2017). The initial subtree built by BIVS is already a good subtree. However, when the value selector is not very efficient, FPMS dominates naive search on TSP. Therefore, the results indicate that when a search strategy is efficient for a problem, FPMS may not help, but it is helpful when there is no specific search strategy for the problem.

In Table 6, we present detailed comparison results of some hard and largest instances that can be solved by ABSO+BIVS in 1 hour. To our best knowledge, our work is the first that finds optimal subtrees without searching. Therefore, to examine the effect of FPMS, we compared also the length of optimal subtrees generated by FPMS and that generated by a random subtree generator, which simulates search with random variable and value ordering. The generator uses a random search strategy to generate a random subtree. At each node of the search tree, it randomly selects an uninstantiated variable x and assigns it with a random value in $dom(x)$. Constraint propagation is performed after each instantiation. The search terminates when the first failure is detected. Then we count the number of optimal s-assignments from the root s-assignment to the leaf. The counting procedure terminates when the first non-optimal s-assignment is found. Then we get the length of the optimal subtree in the random subtree. For each instance, we have generated 100 random subtrees with random seeds from 1 to 100. Note that some of the random subtrees contain optimal subtrees with length 0, since the root s-assignment is not optimal. In the Optimal Subtree Length columns, we present the length of optimal subtrees found by different strategies. The VarNum column is the number of variables in the instance which is the upper bound of the length of an optimal subtree. The FPM column is the length of the optimal subtree found by FPMS. The Longest column is the length of the longest optimal subtree of the 100 random subtrees, and the Mean column is the mean length of 100 random optimal subtrees. Note that some of the random subtrees makes a wrong decision at the root node. Thus the length of such optimal subtrees is 0. From the Mean column. we can see that random search is expected to find fewer optimal subtree. The length of optimal subtrees found by FPMS is much larger than the average of those found by the random strategy. Even the longest one of the 100 random optimal subtrees is outperformed by that of FPMS in the majority of these instances.

In the FPMS and Naive Search columns, the Total columns are total time cost, the FBT columns are the time cost of finding the best solution and the Obj columns are the corresponding objectives. The Mining column is the time cost of frequent pattern mining. We can see that the mining time costs are much less than the total time costs. FPMS finds subtrees with length up to one third of variable number (on UncapWLP-104). On most of these instances, FPMS finds the best solution faster than naive search, although it costs more time to prove the optimality on some of the instances. On the TSP-gr24 instance, although FPMS finds a short optimal subtree, it is still outperformed by naive search since BIVS is efficient on TSP. On the other TSP instance, although no optimal subtree is found, FPMS performs much better than naive search, since BIVS fails to find a good subtree and FPMS finds a good one. Note that FPMS finds an optimal solution much faster than naive search on the two TTPPV instances. This is not contradictory with the results in Table 5, because FPMS performs much better than naive search in the 100 second limit in ABSO+BIVS on TTPPV.

In summary, although FPMS is not guaranteed to find optimal subtrees, it finds optimal subtrees for more than half of the benchmark instances. On some of the instances, FPMS finds the optimal solutions much faster than naive search. It brings improvements in general.

Conclusion

In this paper, we propose FPMS for solving constraint optimization problems. The method applies frequent pattern mining on a set of high-quality solutions generated by random probing. The best frequent patterns are used as the initial assignments for branch-and-bound search. The experiments show that FPMS mines in a small number of random solutions and finds a subtree involving an optimal solution for more than 55% instances of UncapWLP, MKnap, TTPPV and TSP. In general, FPMS finds good subtrees involving high-quality solutions or even optimal solutions, and brings improvements in both total runtime and runtime of finding an optimal solution.

Acknowledgments

The authors thank Anthony Palmieri and Guillaume Perez for sharing the source codes of Objective-Based Selector. This work is supported by the National Natural Science Foundation of China under Grants (61802056, 61976050, 61972384, 61972063) and the Fundamental Research Funds for the Central Universities under Grant 2412018ZD017.

References

Agrawal, R., and Srikant, R. 1994. Fast algorithms for mining association rules. In *Proc. VLDB'94*, 487–499.

Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Mining association rules between sets of items in large databases. In *Proc. SIGMOD'93*, 207–216. ACM.

Bachiri, I.; Gaudreault, J.; Quimper, C. G.; and Chaib-draa, B. 2015. RLBS: an adaptive backtracking strategy based on reinforcement learning for combinatorial optimization. In *Proc. ICTAI'15*.

Balafrej, A.; Bessiere, C.; and Paparrizou, A. 2015. Multi-armed bandits for adaptive constraint propagation. In *Proc. IJCAI'15*, 290–296. AAAI Press.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *Proc. ECAI'04*, volume 16, 146–150.

Cappart, Q.; Goutierre, E.; Bergman, D.; and Rousseau, L. M. 2019. Improving optimization bounds using machine learning decision diagrams meet deep reinforcement learning. In *Proc. AAAI'19*.

Chu, G., and Stuckey, P. J. 2015. Learning value heuristics for constraint programming. In *Proc. CPAIOR'15*, 108–123. Springer.

Dao, T. B. H.; Duong, K. C.; and Vrain, C. 2017. Constrained clustering by constraint programming. *Artificial Intelligence* 244:70–94.

Epstein, S., and Petrovic, S. 2007. Learning to solve constraint problems. In *Proc. ICAPS'07, Workshop on Planning and Learning*.

Fages, J. G., and Prud'Homme, C. 2017. Making the first solution good. In *Proc. ICTAI'17*. IEEE.

Gomes, C., and Sellmann, M. 2004. Streamlined constraint reasoning. In *Proc. CP'04*, 274–289. Springer.

Guns, T.; Dries, A.; Nijssen, S.; Tack, G.; and Raedt, L. D. 2017. MiningZinc: a declarative framework for constraint-based mining. *Artificial Intelligence* 244:6–29.

Guns, T.; Nijssen, S.; and Raedt, L. D. 2011. Itemset mining: a constraint programming perspective. *Artificial Intelligence* 175:1951–1983.

Han, J.; Pei, J.; and Yin, Y. 2000. Mining frequent patterns without candidate generation. In *Proc. SIGMOD'00*, 1–12. ACM.

Hurley, H.; Kotthoff, L.; Malitsky, Y.; and O'Sullivan, B. 2014. Proteus: a hierarchical portfolio of solvers and transformations. In *Proc. CPAIOR'14*.

Lazaar, N.; Lebbah, Y.; Loudni, S.; Maamar, M.; Lemière, V.; Bessiere, C.; and Boizumault, P. 2016. A global constraint for closed frequent pattern mining. In *Proc. CP'16*, 333–349. Springer.

Loth, M.; Sebag, M.; Hamadi, Y.; and Schoenauer, M. 2013. Bandit-based search for constraint programming. In *Proc. CP'13*, 464–480. Springer.

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.

Pesant, G. 2012. A constraint programming approach to the traveling tournament problem with predefined venues. In *Proc. PATAT'12*, 303–315.

Pesant, G. 2016. Counting-based search for constraint optimization problems. In *Proc. AAAI'16*, 3441–3447. AAAI Press.

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

Samorani, M., and Laguna, M. 2012. Data-mining-driven neighborhood search. *INFORMS Journal on Computing* 24:210–227.

Schaus, P.; Aoga, J. O. R.; and Guns, T. 2017. Coversize: A global constraint for frequency-based itemset mining. In *Proc. CP'17*, 529–546. Springer.

Xia, W., and Yap, R. H. C. 2018. Learning robust search strategies using a bandit-based approach. In *Proc. AAAI'18*, 431–437. AAAI Press.

Xu, Y.; Stern, D.; and Samulowitz, H. 2009. Learning adaptation to solve constraint satisfaction problems. In *Proc. LION'09*.

Zaki, M. J. 2000. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering* 12:372–390.

Zhou, Y.; Hao, J.; and Duval, B. 2017. When data mining meets optimization: a case study on the quadratic assignment problem. *http://arxiv.org/abs/1708.05214v1*.