Improving Local Search Algorithm for Pseudo Boolean Optimization

YUJIAO ZHAO, School of Computer Science and Information Technology, Northeast Normal University, China
YIYUAN WANG, School of Computer Science and Information Technology, Northeast Normal University, China and Key Laboratory of Applied Statistics of MOE, Northeast Normal University, China
YI CHU, Institute of Software, Chinese Academy of Sciences, China
WENBO ZHOU, School of Computer Science and Information Technology, Northeast Normal University, China and Key Laboratory of Applied Statistics of MOE, Northeast Normal University, China
SHAOWEI CAI, Key Laboratory of System Software, Institute of Software, Chinese Academy of Sciences and School of Computer Science and Technology, University of Chinese Academy of Sciences, China
MINGHAO YIN, School of Computer Science and Information Technology, Northeast Normal University, China and Key Laboratory of Applied Statistics of MOE, Northeast Normal University, China

Pseudo-Boolean optimization (PBO) is usually used to model combinatorial optimization problems, especially for some real-world applications. Despite its significant importance in both theory and applications, the performance of current PBO solvers is still limited. This paper develops a novel local search algorithm for PBO, which has four main ideas. First, we design a new primary scoring function and a two-level selection strategy to evaluate all candidate variables. Second, we introduce a new weighting scheme to accurately guide the search process toward more promising directions. Third, we propose a novel deep optimization strategy to disturb some search processes. Fourth, an efficient solution space exploration mechanism is applied to help the algorithm jump out of local optimum. We conduct experiments on a broad range of public benchmarks, including three large-scale practical application benchmarks, two benchmarks from PB competitions, an integer linear programming optimization benchmark, a crafted combinatorial benchmark, and a combinatorial optimization knapsack benchmark to compare our proposed algorithm against twelve state-of-the-art competitors, including seven recently-proposed pure stochastic local search PBO solvers, a non-traditional stochastic local search combined with complete oracle, two complete PB solvers, and two mixed integer programming (MIP) solvers. Our proposed algorithm has been shown to perform best on these three real-world benchmarks. On the other five benchmarks, our algorithm shows competitive performance compared to state-of-the-art competitors, and it significantly outperforms all other local search algorithms, indicating that our algorithm greatly advances the state of the art in local search for solving PBO.

JAIR Associate Editor: Kai-Wei Chang

Authors' Contact Information: Yujiao Zhao, ORCID: 0000-0001-9285-2793, zhaoyj@nenu.edu.cn, School of Computer Science and Information Technology, Northeast Normal University, Changchun, Jilin, China; Yiyuan Wang, ORCID: 0000-0002-3071-3461, yiyuanwangjlu@126.com, School of Computer Science and Information Technology, Northeast Normal University, China and Key Laboratory of Applied Statistics of MOE, Northeast Normal University, Changchun, China; Yi Chu, ORCID: 0000-0003-4681-7414, chuyi2020@iscas.ac.cn, Institute of Software, Chinese Academy of Sciences , Beijing, China; Wenbo Zhou, ORCID: 0000-0002-1009-4544, zhouwb646@nenu.edu.cn, School of Computer Science and Information Technology, Northeast Normal University, China and Key Laboratory of Applied Statistics of MOE, Northeast Normal University, Changchun, China; Shaowei Cai, ORCID: 0000-0003-1730-6922, caisw@ios.ac.cn, Key Laboratory of System Software, Institute of Software, Chinese Academy of Sciences and School of Computer Science and Technology, University of Chinese Academy of Sciences , Beijing, China; Minghao Yin, ORCID: 0000-0002-6226-2394, ymh@nenu.edu.cn, School of Computer Science and Information Technology, Northeast Normal University, China and Key Laboratory of Applied Statistics of MOE, Northeast Normal University, Changchun, China.

DOI: 10.1613/jair.1.16626

JAIR Reference Format:
Yujiao Zhao, Yiyuan Wang, Yi Chu, Wenbo Zhou, Shaowei Cai, and Minghao Yin. 2025. Improving Local Search Algorithm for Pseudo Boolean Optimization. *Journal of Artificial Intelligence Research* 83, Article 18 (July 2025), 38 pages. DOI: 10.1613/jair.1.16626

1 Introduction

The Boolean satisfiability (SAT) problem is a prototypical NP-complete problem [12], whose aim is to determine whether a given propositional formula is satisfiable or not. The SAT problem plays a core role in many domains of computer science and artificial intelligence [17], such as electronic design automation [33] and AI planning [22]. Many real-world problems can be encoded into SAT and its optimization version MaxSAT and solved using their powerful solvers. However, due to the limited expressive power of SAT and MaxSAT, their encoding in conjunctive normal form (CNF) often generates very large problem instances. For such cases, linear pseudo-Boolean (PB) constraints provide a more expressive and natural way to express constraints than SAT and MaxSAT [30]. Besides, linear PB constraints are very close enough to CNF to benefit from the recent advances in SAT solving [30].

In practice, PB constraints occur in many areas, including VLSI design, economics, computer vision, and manufacturing [41, 42, 30]. The PBO consists of a set of PB constraints and an objective function, whose goal is to find a solution that minimizes the objective function and satisfies all PB constraints. The decision problem of PBO is Pseudo-Boolean solving (PBS). The PBS focuses on determining whether a variable assignment exists that satisfies all PB constraints [30].

1.1 Related Work

Solvers for PBO can be categorized as complete and incomplete. Existing complete solvers dedicated to solving PBO can be categorized into three families. Firstly, being seen as a natural extension of the CNF used in SAT, PB constraints can be seamlessly transformed into CNF and then directly solved by SAT solvers employing conflict-driven clause learning (CDCL) technique. The well-known PBO/PBS solvers *MiniSat+* [15], *Open-WBO* [28], and *NaPS* [31] are based on this idea. However, encoding PB constraints in CNF as disjunctive clauses loses the semantics of higher-level constraints, diminishing reasoning capabilities. To alleviate this issue, an alternative approach involves extending CDCL to use native reasoning techniques directly for PB constraints. Representative PBO/PBS solvers include *Sat4j* [25], *RoundingSat* [16, 14], *RS/LP* [13], etc. Finally, motivated by the success of so-called implicit hitting set (IHS) implementations for MaxSAT, Smirnov et al. developed a native IHS approach to solve PBO, named *PBO-IHS* [35, 34]. In addition, since PB constraints can be transformed into equivalent 0-1 integer linear programming constraints, mixed integer programming (MIP) solvers can be directly applied to solving PBO. Representative and high-performance MIP solvers include *SCIP* [4] - one of the fastest non-commercial solvers, and *Gurobi* [18] - one of the most powerful commercial solvers.

Compared to complete PBO solvers, incomplete PBO solvers mainly focus on local search methods, which can find an approximate solution within a reasonable time. Lei et al. [26] proposed a novel local search algorithm called *LS-PBO* to handle PBO. Key features of *LS-PBO* include a converting method to obtain corresponding objective constraints, a weighting scheme to guide the search direction, and a scoring function to flip some candidate variables. Subsequent researches have primarily focused on designing efficient heuristic strategies based on *LS-PBO*, which can be broadly categorized into two categories. The first focuses on improving key factors that affect the performance of local search. For example, *DeciLS-PBO* [21] introduced a novel constraint heuristic that prioritizes constraints that are less satisfied during each iteration. Chen et al. [10] presented *DLS-PBO*, which incorporates a new dynamic scoring mechanism to effectively escape local optimum even after flipping thousands of variables. Building on *DLS-PBO*, *ParLS-PBO* [10] was proposed as the first parallel local search-based PBO solver, coordinating multiple local search threads through a shared solution pool that is updated based on both solution quality and diversity. The second line of work aims to integrate local search with powerful

reasoning techniques from complete solvers. For instance, *LS-PBO$_{ss}$* and *NuPBO$_{ss}$*, proposed by Chen et al. [9], adopt local search as the main solver and invoke unit propagation to modify the current assignment when the search is detected to be trapped in a local optimum. Recently, *OraSLS* [20] was introduced as a local search algorithm operating over the assignment space of the objective variables. However, *OraSLS* operates entirely within the feasible solution space by invoking a complete PB solver whenever an objective variable's assignment is considered for flipping, to determine whether the current assignment can be extended to a complete feasible solution. Besides, for some special cases of PBO such as NK-Landscapes and MAX-kSAT, a new perturbation strategy called VIGbP was proposed [38].

1.2 Contributions

In this work, to further improve the performance of local search algorithms on solving PBO, we propose a local search framework for PBO. This framework is based on four ideas, focusing on addressing the critical factors that influence the efficiency of local search.

First, we design a novel two-level scoring mechanism including a primary scoring function and a two-level selection strategy to evaluate the benefits derived from flipping a variable. The primary scoring function considers the violation degree of unsatisfied constraints and utilizes a smooth function to balance the violation degree of different constraints. Moreover, to address the issue about tie-breaking in the primary scoring function, we propose a fragile scoring function *hhscore* as the secondary scoring function. The proposed *hhscore* can greatly differentiate between two kinds of satisfied constraints by using the definition of satisfied threshold.

Second, we introduce a new weighting scheme to accurately guide the search process toward more promising directions. Instead of merely increasing the weight of PB constraints when a local optimum is reached and setting an upper bound on the weight of constraints, we implement a new weighting scheme that applies stricter criteria for updating the weights of both the PB constraints and the objective constraints.

Third, we present a novel deep optimization strategy (DeepOpt) to deeply probe some search regions using lock and unlock operations during the local search. Recently, the DeepOpt strategy was first proposed by Chen et al. [8] and has been successfully applied in solving two variants of dominating set problem. In our proposed DeepOpt strategy, we preferentially select some variables in unsatisfied constraints as unlock operations. Moreover, we use some trigger conditions to decide whether the algorithm calls DeepOpt or not.

Fourth, we construct an efficient solution space exploration mechanism, which is divided into a constraint selection phase and a variable flipping phase. In the constraint selection phase, multi-armed bandit (MAB) technology is used to help the local search algorithm select appropriate PB constraints and objective constraints to be satisfied when getting stuck in a local optimum. Recently, Zheng et al. [44, 45] first applied the MAB model to local search algorithms for solving MaxSAT, significantly improving the algorithm's performance. Furthermore, we propose a constraint sampling selection strategy that adopts two sampling ways to collect some samples among all unsatisfied constraints to modify the current candidate solution.

In the variable flipping phase, we design a diversity flip method to enhance the search capability, allowing for the exploration across different search regions of the solution space. In the proposed method, we employ a probabilistic heuristic "best from multiple selections" (BMS) [5] to select a candidate variable. Besides, we also apply a new scoring function to simultaneously flip two variables.

By incorporating these four ideas, we develop a local search algorithm for PBO. Extensive experiments are carried out to evaluate our algorithm on the benchmarks used in the literature. Experimental results indicate that the proposed algorithm outperforms the current state-of-the-art solvers on almost all the benchmarks, with performance that rivals the strongest commercial solver *Gurobi*.

This paper is an improved version of our conference papers [11, 46]. The primary scoring function and the new weighting scheme have been mentioned in conference paper [11]; moreover, the two-level selection strategy, the

deep optimization strategy, the sampling selection strategy, and the diversity flip method have been introduced in conference paper [46]. In our article, we first integrate all the strategies proposed in two papers due to the high performance of paper [11] on the standard benchmarks and the efficient performance of paper [46] on the large-scale practical application benchmarks. In addition, we apply machine learning technique in the context of PBO and propose a machine learning model to guide the search direction. Finally, an important factor to note is that preprocessing reduction is a key step in MIP solvers. Therefore, we integrate the PaPILO[1], a parallel preprocessing reduction solver for integer and linear optimization into our proposed solver. It works by reducing some constraints and variables to decrease the instance size before solving.

The rest of this paper is organized as follows. Section 2 introduces some necessary background knowledge. After that, we present our proposed algorithm and its components in Sections 3-7. Section 8 discusses the main differences between this paper and two extended conference papers. Experimental results are shown in Section 9. Finally, we make conclusions.

2 Preliminaries

Since a non-linear pseudo-Boolean (PB) constraint can be translated into an equivalent set of linear PB constraints [30], we only start with a review of the basics of linear PB constraints here. A linear PB constraint is defined over a finite set of Boolean variables. Boolean variable x_i can take only two values *false* (0) and *true* (1). A literal l_i over a Boolean variable x_i is either x_i or $\neg x_i = 1 - x_i$. A linear PB constraint is a 0-1 integer inequality.

$$\sum_i a_i l_i \rhd b \tag{1}$$

where a_i and b are integer constants, l_i are literals and $\rhd \in \{=, >, \geq, <, \leq\}$ is one of the classical relational operators. All PB constraints can be normalized into the following form.

$$\sum_i a_i l_i \geq b \tag{2}$$

where all the literals l_i are distinct, and all the coefficients a_i and the *degree* b are non-negative integers.

A PB formula is a conjunction of PB constraints, denoted as $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where C_p ($p \in \mathbb{Z}, 1 \leq p \leq m$) is a PB constraint. The PBO problem consists of a PB formula F and an objective function $O : \sum_c a_c l_c$ where a_c is a non-negative integer coefficient. Given a PB constraint $C_p : \sum_i a_i^p l_i^p \geq b^p$, the sum of its coefficients of the literals is defined as $sum_{coe}(C_p) = \sum_i a_i^p$, its average coefficient is defined as $avg_{coe}(C_p) = sum_{coe}(C_p)/|L(C_p)|$ where $L(C_p)$ is the set of literals in C_p, and its maximum coefficient a_{max}^p is the maximum value of coefficients in C_p. The average coefficient of an objective function O is defined as $avg_{coe}(O) = \sum_c a_c/|L(O)|$ where $L(O)$ is the set of literals in O.

In practical solving, given the objective function $O : \sum_c a_c l_c$, each pair of literal l_c and its coefficient a_c is transformed into an objective constraint denoted as O_c through a transformation rule. The transformation rule is to negate the sign of each literal in the objective function and then make the product of the negated literal and its coefficient greater than or equal to the literal's coefficient. For example, for the above pair of literal l_c and its coefficient a_c, we can transform this pair into an objective constraint, i.e., $O_c : a_c \neg l_c \geq a_c$. The goal of this transformation is to standardize the objective function to the format of PB constraints, thereby making the minimization of the objective function equivalent to satisfying the maximum number of objective constraints.

Given a PB formula F, its complete assignment is a mapping that assigns 0 or 1 to each variable. Given a complete assignment of F, if a literal evaluates to true, we say it is a *true literal* and otherwise it is a *false literal*. A PB constraint C_p and an objective constraint O_c are considered *satisfied* when the left and right terms of

[1] PaPILO repository: https://github.com/lgottwald/PaPILO

the constraint evaluate to integers which satisfy the relational operator. Otherwise, C_p and O_c are considered *unsatisfied*. The sum of coefficients of true literals in C_p are denoted as $SatL(C_p) = \sum_{l_i^p=1 \wedge l_i^p \in C_p} a_i^p$. Consider the objective constraint $O_c : a_c \neg l_c \geq a_c$, since O_c contains only one literal $\neg l_c$, the assignment of $\neg l_c$ determines whether O_c is satisfiable. To facilitate this, we utilize $SatL(O_c)$ to record the coefficient of the true literal, meaning that $SatL(O_c) = a_c$ if $\neg l_c = 1$; otherwise, $SatL(O_c) = 0$ if $\neg l_c = 0$.

Given an assignment α, we define the value of violation of a PB constraint C_p and an objective constraint O_c as

$$viol(C_p) = max(0, b^p - SatL(C_p)),$$

$$viol(O_c) = a_c - SatL(O_c)$$

In other words, if the PB constraint C_p and the objective constraint O_c are satisfied under α, then $viol(C_p) = 0$ and $viol(O_c) = 0$; otherwise (i.e., C_p and O_c are unsatisfied), $viol(C_p)$ and $viol(O_c)$ are the integer distance of C_p and O_c from being satisfied. An assignment α of F is feasible if and only if α satisfies all PB constraints in F. The value of the objective function of a feasible solution α is denoted as $obj(\alpha)$. The PBO problem aims to obtain a feasible solution for F with the minimum objective value. This is equivalent to satisfying as many objective constraints as possible while ensuring all PB constraints are satisfied.

The stochastic local search algorithms that employ constraint weighting schemes usually maintain weight for each constraint. We use $w(C_p)$ to represent the weight of each PB constraint C_p, and $w(O_c)$ to represent the weight of each objective constraint O_c.

Example 2.1. Consider a PBO instance F_1, which consists of three PB constraints:

$$C_1 : 4\neg x_1 + 1x_2 + 1x_3 \geq 4, C_2 : 3\neg x_1 + 1x_2 + 2x_3 \geq 4, C_3 : 2x_1 + 1x_2 + 1x_3 \geq 2$$

and an objective function $O = 2x_1 + 3\neg x_2$. According to the transformation rule, O will be transformed into two objective constraints:

$$O_1 : 2\neg x_1 \geq 2, O_2 : 3x_2 \geq 3$$

An optimal solution to F_1 is $\alpha^* = (0, 1, 1)$ and $obj(\alpha^*) = 0$.

3 Two-level Scoring Mechanism

In this section, we introduce a primary scoring function used to evaluate the benefits of flipping a variable. Then, we discuss a secondary scoring function to reinforce local search algorithms for PBO and propose a two-level selection strategy to decide a candidate variable.

3.1 A New Primary Scoring Function

This section first reviews the scoring function used in *LS-PBO*, and then introduces our newly proposed scoring function based on a smoothing strategy.

Given a PBO instance, which consists of m PB constraints and k objective constraints, we assume that the current assignment is α.

Scoring Function in *LS-PBO*. Before introducing our new scoring function, we first describe the existing scoring function proposed in *LS-PBO* [26], which is presented as follows.

- For a PB constraint C_p, the penalty function is defined as $penalty(C_p) = w(C_p) \times viol(C_p)$; then the score of a variable x is defined as the decrement of the sum of the penalty function values of all PB constraints caused by flipping x, which is denoted by $hscore(x)$.
- For an objective constraint O_c, the penalty function is defined as $penalty(O_c) = w(O_c) \times viol(O_c)$; then the objective score of a variable x is defined as the decrement of the penalty function value of the objective constraint caused by flipping x, which is denoted by $oscore(x)$.
- The score of a variable x is defined as $score(x) = hscore(x) + oscore(x)$.

Table 1. The violation value of PB and objective constraints under all assignments of the PBO instance F_2 in Example 3.1.

$viol/obj$	Assignment (x_1,x_2,x_3)							
	(0,0,0)	(0,0,1)	(0,1,0)	(0,1,1)	(1,0,0)	(1,0,1)	(1,1,0)	(1,1,1)
$viol(C_1)$	0	0	0	0	4	3	3	2
$viol(C_2)$	1	0	0	0	4	2	3	1
$viol(C_3)$	29	13	14	0	0	0	0	0
$viol(O_1)$	0	0	0	0	2	2	2	2
$viol(O_2)$	3	3	0	0	3	3	0	0

An Intuitive View. The above scoring function has a drawback, which is due to its underlying penalty function. The penalty function above considers the weights and *viol* values of PB and objective constraints. In this way, it measures the importance of a variable in a constraint C_p by the coefficient of the corresponding variable in C_p. Nevertheless, it should be noted that, as the value of $score(x)$ is determined by all PB constraints and the objective constraint involving x, the above penalty function may overemphasize the importance of x in constraints with relatively large coefficients, resulting in an unreasonable value of its *score*. In the following, we illustrate our intuition through a simple PBO instance.

Example 3.1. Consider a PBO instance F_2, which only differs from instance F_1 in the coefficients of the third PB constraint:

$$C_1 : 4\neg x_1 + 1x_2 + 1x_3 \geq 4, C_2 : 3\neg x_1 + 1x_2 + 2x_3 \geq 4, C_3 : 29x_1 + 15x_2 + 16x_3 \geq 29$$

and an objective function $O = 2x_1 + 3\neg x_2$ that can be transformed into $O_1 : 2\neg x_1 \geq 2$ and $O_2 : 3x_2 \geq 3$. The values of *viol* of PB and objective constraints under all assignments are presented in Table 1. Solutions for F_2 are those resulting in the zero value of *viol* for all PB constraints. From Table 1, the optimal solution is α^* = (0, 1, 1), and obj(α^*) is 0.

Given instance F_2, consider a scoring function without any weighting scheme, or equivalently, the weight of each constraint is 1, i.e., $w(C_1) = 1$, $w(C_2) = 1$, $w(C_3) = 1$, $w(O_1) = 1$ and $w(O_2) = 1$. The initial assignment α = (0, 0, 0). In accordance with the definition of the scoring function in *LS-PBO*, $score(x_1)$ = 20, $score(x_2)$ = 19, and $score(x_3)$ = 17. Actually, in order to optimize the assignment, stochastic local search algorithms tend to select the variable to be flipped as the one with the largest score, so in this situation, x_1 is picked. After flipping x_1, the assignment becomes α = (1, 0, 0), and the score value of each variable becomes $score(x_1)$ = -20, $score(x_2)$ = 5, $score(x_3)$ = 3. Then, x_2 should be flipped based on *score* value. However, no matter whether x_2 or x_3 is flipped, the Hamming distance between the current assignment and the optimal solution is the same as that between the initial assignment and the optimal solution, which is two. The search is not progressing in the direction towards the optimal solution.

In practice, PBO instances encoded from real-world problems are much more complex than the given illustrative example. If stochastic local search algorithms conduct the search in incorrect directions, it would be difficult to identify a promising search space that is more likely to contain the optimal solution or those close to optimality.

As presented in Table 1, when we focus on the value of $viol(C_3)$, for those cases where the value of $viol(C_3)$ is not 0, its value is much larger than the *viol* value of other PB constraints. Considering that each PB constraint has a penalty value directly proportional to its *viol* value, utilizing scoring function aims to guide the search towards the area with a lower sum of penalty values. Consequently, through making use of such scoring function, the algorithms would prefer the falsified literal with the largest coefficient to be true (in instance F_2, under the assumption that the current assignment is (0, 0, 0), this falsified literal is x_1 in the PB constraint C_3).

Our New Scoring Function. In our opinion, a good scoring function for PBO should balance the *viol* values of different constraints. To this end, we propose to smooth the *penalty* values of constraints. For simplicity, we denote the smoothing function of a PB constraint C_p as $smooth(C_p)$, and the smoothing function of an objective constraint O_c as $smooth(O_c)$. Based on the idea of balancing the *viol* value, we define the following smoothed scoring function:

- For a PB constraint C_p, the penalty function is defined as $penalty(C_p) = \frac{w(C_p) \times voil(C_p)}{smooth(C_p)}$; then the score of a variable x is defined as the decrement of the sum of the penalty function values of all PB constraints caused by flipping x, which is denoted by $hscore(x)$.
- For an objective constraint O_c, the penalty function is defined as $penalty(O_c) = \frac{w(O_c) \times viol(O_c)}{smooth(O_c)}$; then the objective score of a variable x is defined as the decrement of the penalty function value of the objective constraint caused by flipping x, which is denoted by $oscore(x)$.
- The score of a variable x is defined as $score(x) = hscore(x) + oscore(x)$.

In order to instantiate the above scoring function, we propose to use a method for smoothing by using the average of the constraint coefficients, i.e., $smooth(C_p) = round(avg_{coe}(C_p))$, $smooth(O_c) = round(avg_{coe}(O))$ ($round$ is a rounding function). Consider the PBO instance F_2 in Example 3.1, which has $smooth(C_1) = 2$, $smooth(C_2) = 2$, $smooth(C_3) = 20$, $smooth(O_1) = 3$ and $smooth(O_2) = 3$. Assume that each PB constraint and each objective constraint has a weight of 1 and the current assignment $\alpha = (0, 0, 0)$. Based on the new scoring function, $score(x_1)$= -2.72, $score(x_2) = 2.25$, and $score(x_3) = 1.3$. Hence, stochastic local search algorithms would select variable x_2 to be flipped. Flipping x_2 would change the current assignment α to (0,1,0), and the *score* value of each variable would become $score(x_1)$ = -2.97, $score(x_2)$ = -2.25, $score(x_3)$ = 0.7. Afterward, stochastic local search algorithms would select variable x_3 to be flipped. If x_3 is flipped, assignment α becomes (0, 1, 1), which is the optimal solution of instance F_2.

According to this illustrative example, it can be observed that, by using the average of constraint coefficients to smooth the *penalty* value, the issue of the large difference among coefficients of a variable in various constraints can be alleviated, resulting in a more effective scoring function.

3.2 Two-Level Selection Strategy

This section presents the two-level selection strategy by first defining the auxiliary scoring function, followed by the overall variable flipping rule.

3.2.1. Fragile Scoring Function.

In our algorithm, we use the newly proposed scoring function with a smooth strategy (i.e., *score*) as the primary scoring function, as described in Section 3.1. According to our preliminary experiments, 3% candidate variables on average have the same largest *score* value during the search. To address the issue about tie-breaking in the primary scoring function, previous work uses the *age* information of variables as the secondary scoring function, where *age* is defined as the number of steps since the last time it is flipped. But the experimental results show that the use of only *age* cannot effectively guide the search process. Thus, to further choose a variable among these variables with the same best *score* value, we design a novel fragile scoring function denoted as *hhscore*.

Before introducing *hhscore*, we first introduce a necessary concept. For a PB constraint C_p, $gap(C_p) = min\{b^p + a^p_{max}, sum_{coe}(C_p)\}$ is used to denote the *satisfied threshold* of C_p, which plays a key role in our proposed *hhscore*. Based on the satisfied threshold of PB constraints, we define a fragile satisfied PB constraint in the following.

Definition 3.2. For a satisfied PB constraint C_p (i.e., $SatL(C_p) \geq b^p$), the C_p is a fragile satisfied PB constraint if and only if $SatL(C_p) < gap(C_p)$.

On the one hand, if a satisfied PB constraint C_p is fragile, we think that flipping any variable in this PB constraint would probably make this constraint become unsatisfied. On the other hand, if a satisfied PB constraint is not fragile (i.e., $SatL(C_p) \geq gap(C_p)$), we think this PB constraint is solid, which means that flipping any true literal in C_p would not make C_p become unsatisfied with a high probability. Although we have also tried a similar property that measures the number of true literals in a satisfied PB constraint C_p, such as $gap(C_p) = b^p + 1$, we did not find it useful in our algorithm.

To maintain the information of "fragile" for each literal during the search process, we define *inner* as below.

Definition 3.3. Suppose that a Boolean variable x_i whose literal l_i appears in some PB constraints (e.g., C_p) and the coefficient of l_i in C_p is a_i^p. The value of $inner(x_i, C_p)$ is calculated as follows.

(a) C_p is an unsatisfied PB constraint, i.e., $SatL(C_p) < b^p$:
- If l_i is a true literal in C_p, $inner(x_i, C_p) = 0$;
- If l_i is a false literal in C_p, $inner(x_i, C_p) = max\{a_i^p - (b^p - SatL(C_p)), 0\}$.

(b) C_p is a fragile satisfied PB constraint, i.e., $b^p \leq SatL(C_p) < gap(C_p)$:
- If l_i is a true literal in C_p, $inner(x_i, C_p) = -min\{a_i^p, SatL(C_p) - b^p\}$;
- If l_i is a false literal in C_p, $inner(x_i, C_p) = min\{a_i^p, gap(C_p) - SatL(C_p)\}$.

(c) C_p is satisfied but not a fragile PB constraint, i.e., $gap(C_p) \leq SatL(C_p)$:
- If l_i is a true literal in C_p, $inner(x_i, C_p) = -max\{a_i^p - (SatL(C_p) - gap(C_p)), 0\}$;
- If l_i is a false literal in C_p, $inner(x_i, C_p) = 0$.

The value of $inner(x_i, C_p)$ is subject to three distinct factors including the state of C_p (i.e., *satisfied* or *unsatisfied*), the value of l_i (i.e., *true* or *false*), and the coefficient of l_i (i.e., a_i^p). To make the readers easily understand the above Definition 2, we list all the situations corresponding to different *inner* values in Figure 1.

Considering all the PB constraints in which variable x's literal appears, the proposed fragile scoring function $hhscore$ of variable x_i is defined as below.

$$hhscore(x_i) = \sum_{p=1}^{n_x} inner(x_i, C_p) \tag{3}$$

where n_x is the number of PB constraints including the literal of x_i.

3.2.2. Selection Rule.

Combining the respective advantages of *score* and *hhscore*, we propose a two-level selection strategy as follows.

Flipping Rule. Flip a variable x_i with the biggest $score(x_i)$ value, breaking ties by preferring the one with the biggest $hhcore(x_i)$ value, further ties are broken randomly.

The proposed secondary scoring function is inspired by *subscore* [6], but has two essential differences. First, our proposed scoring function can be considered as a general version of *subscore* because the value of satisfied threshold *gap* is equal to 2 for the SAT problem, which is the same function as $subscore$. Second, previous work uses a linear combination of *subscore* and another scoring function as the primary scoring function, whereas our work considers a two-level scoring function to guide the search process. In addition, experiments show that the trigger fraction of using *hhscore* at least once for all tested instances is about 99.6%.

Example 3.4. Consider a PBO instance F_3, differing from F_1 only in the sign of the literals in the objective function:

$$C_1 : 4\neg x_1 + 1x_2 + 1x_3 \geq 4, C_2 : 3\neg x_1 + 1x_2 + 2x_3 \geq 4, C_3 : 2x_1 + 1x_2 + 1x_3 \geq 2$$

and an objective function $O = 2x_1 + 3x_2$ that can be transformed into two objective constraints:

$$O_1 : 2\neg x_1 \geq 2, O_2 : 3\neg x_2 \geq 3.$$

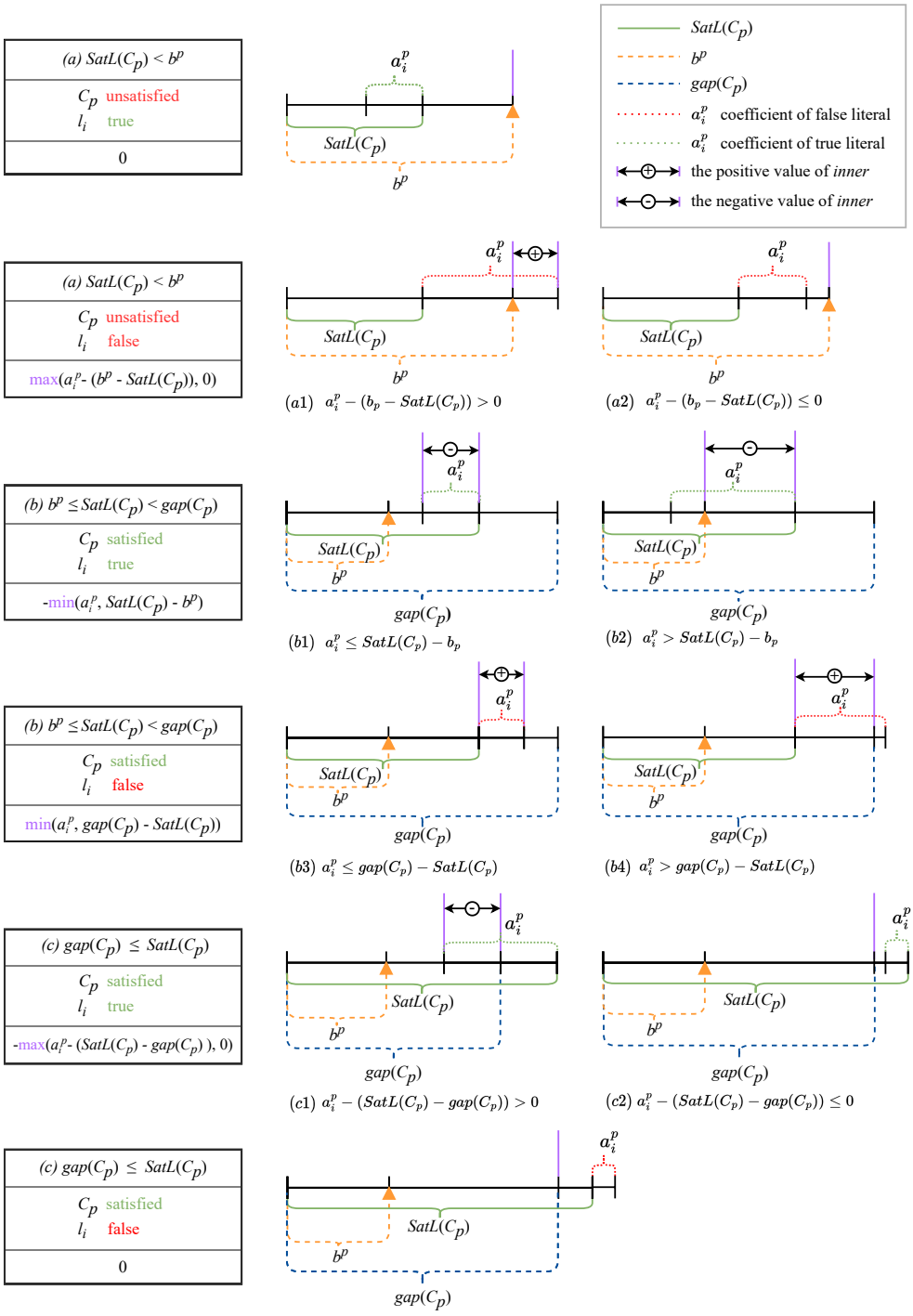


Fig. 1. A graphical explanation of *inner*.

Given instance F_3, consider a scoring function without any weighting scheme and smooth strategy, or equivalently, the weight of each constraint is 1, i.e., $w(C_1) = w(C_2) = w(C_3) = w(O_1) = w(O_2) = 1$ and the smooth value of each constraint is 1, i.e., $smooth(C_1) = smooth(C_2) = smooth(C_3) = smooth(O_1) = smooth(O_2) = 1$. Assuming the current assignment is α = (0,1,0), then C_1 and C_2 meet the definition of fragile satisfied PB constraints because $b^1 \leq SatL(C_1) < gap(C_1)$ and $b^2 \leq SatL(C_2) < gap(C_2)$. According to the definition of the proposed scoring function, $score(x_1)$ = -7, $score(x_2)$ = 1, and $score(x_3)$ = 1. At this point, a situation occurs where the primary scores of two variables x_2 and x_3 are the same.

Following the proposed flipping rule, breaking ties by preferring the one with the biggest *hhscore* value, further ties are broken randomly. According to the formula in Definition 2, $hhscore(x_2)$ = -1 and $hhscore(x_3) = 3$. Clearly, at this point, the choice is to flip x_3.

Observing instance F_3, with the assignment α = (0,1,0), flipping x_2 or x_3 is random without *hhscore*. However, flipping x_2 turns C_3 from satisfied to unsatisfied, whereas flipping x_3 would not lead to any situation that makes any PB constraint unsatisfied, hence the preference to flip x_3.

4 A New Weighting Scheme

Combinatorial optimization problems with both hard and soft constraints (i.e., PB and objective constraints in PBO) usually require effective weighting schemes that balance the weights of hard and soft constraints. A potential problem was pointed out in a previous study [7]: the excessive weight given to soft constraints may make it difficult to satisfy all the hard constraints, thereby hindering the algorithm's capability of finding solutions. Moreover, an existing study [37] demonstrates that designing an effective weighting scheme for problems with hard constraints is challenging as it requires weighting unsatisfied constraints while maintaining the distinction between hard and soft constraints.

To alleviate the above problem, the weighting scheme proposed in *LS-PBO* sets an upper bound ζ (an integer parameter) to the maximum value of each objective constraint weight. We use *UnSatSet* to denote the set of unsatisfied PB constraints. For a PBO instance F, the average of the product of the $avg_{coe}(C_p)$ and $w(C_p)$ of all constraints C_p is denoted as $wavg_{coe}(F)$, that is, $wavg_{coe}(F) = (\sum_{p=1}^{m}(avg_{coe}(C_p) \times w(C_p)))/m$. Assuming that the current assignment is α, the best solution that has been found is α^*, and its corresponding objective function value is $obj(\alpha^*)$.

The weighting scheme adopted in *LS-PBO* is described as follows:

- **Initialization rule:** at the start of the local search process, the weight of each PB constraint C_p is initialized as 1, i.e., $w(C_p) := 1$; the weight of each objective constraint O_c is also initialized as 1, i.e., $w(O_c) := 1$.
- **Update rule:** when the search is trapped in a local optimum (i.e., there is no variable whose *score* value is greater than 0), for each C_p in $UnSatSet$, $w(C_p) := w(C_p)+1$; if $obj(\alpha) \geq obj(\alpha^*)$ and $w(O_c) \times avg_{coe}(O_c) - wavg_{coe}(F) \leq \zeta$, $w(O_c) := w(O_c) + 1$, where ζ is a parameter introduced by *LS-PBO*.

For PB constraints, the weights of unsatisfied PB constraints increase by 1 whenever the algorithm is trapped in a local optimum. However, this rule overlooks the effect of the frequency in updating unsatisfied PB constraint weights. Therefore, we conduct experiments to analyze the impact of different update frequencies of constraint weights on algorithm performance.

At first, we define *visitTime* to track the number of times that each PB constraint appears in the *UnSatSet* when the algorithm is trapped in a local optimum. If *UnSatSet* is not empty, the *visitTime* of each unsatisfied PB constraint is increased by one. A PB constraint's weight is only increased after its *visitTime* reaches a certain threshold, which is denoted as *visitThres*. This setup allows for the analysis of how varying thresholds affect the algorithm performance.

For the experiment, we select three large-scale application benchmarks (i.e., MWCB, WSNO and SAP) and five standard benchmarks (i.e., PB2016, PB2024, MIPLIB, CRAFT and KNAP). Details of these benchmarks are

provided in Section 9.1. For each benchmark, we randomly select five instances, totaling 40 instances for testing. For each instance, the candidate set for *visitThres* is set to $\{1, 2, \ldots, 40\}$. Each instance uses the same random seed (i.e., 1) and runs for 300 seconds. We compare the experiment results of different *visitThres* values, and record the *visitThres* corresponding to the best solution as $visitThres_{best}$. Note that each instance may have multiple $visitThres_{best}$ since different *visitThres* can yield the same optimal solution.

The results of the experiment are displayed in Figure 2. The x-axis represents the 40 instances, whereas the y-axis corresponds to different *visitThres*. The $visitThres_{best}$ values corresponding to the optimal solutions are marked with blue triangles. The original update rule, which updates the weight of a PB constraint when its *visitThres* is 1 within the *UnSatSet*, is represented by a red horizontal line used as a reference. Triangles on this line suggest that the initial strategy is optimal. It is evident that *visitThres* with different values substantially affects the performance across most instances. Therefore, varying the *visitThres* of PB constraints could be a promising strategy to improve the performance of the weighting scheme.

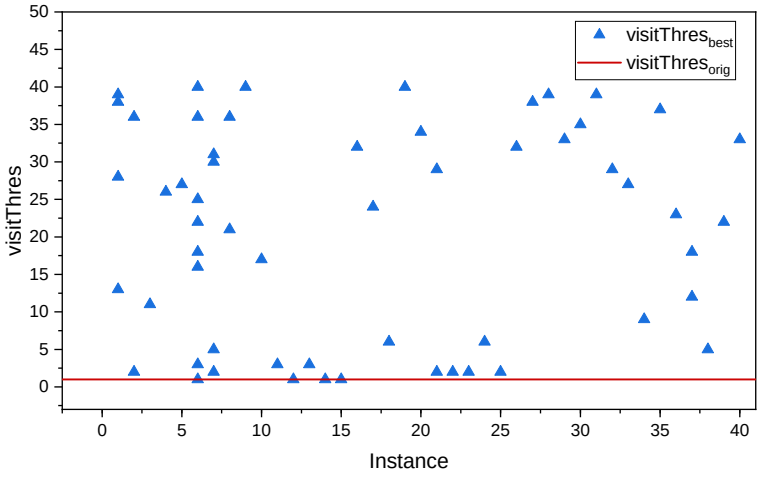


Fig. 2. Experimental analysis of the impact of different *visitThres* on algorithm performance.

In addition, for objective constraints, the setting of ζ greatly affects the performance of *LS-PBO*, and if the average of the coefficients of the objective constraints is much greater than the average of the coefficients of the PB constraints, the weight of the objective constraints basically would not be updated.

We propose to deal with these problems by modifying the conditions of updating weights. Specifically, we propose stricter conditions for increasing the *weight* of PB constraints and objective constraints, respectively. Our proposed weighting scheme is as follows:

- **Modified initialization rule:** at the start of the local search process, the weight of each PB constraint C_p is initialized as 1, i.e., $w(C_p) := 1$; the weight of each objective constraint O_c is initialized as 0, i.e., $w(O_c) := 0$.
- **Modified update rule:** when the search is trapped in a local optimum (i.e., there is no variable whose *score* value is greater than 0), there are two cases as below. At first, the *visitTime* for each PB constraint is initially set to 0.
  - *UnSatSet* is not empty.
    - Increase the $visitTime(C_p)$ of each PB constraint C_p in *UnSatSet* by 1;

  - If the *visitTime* for any PB constraint C_p in *UnSatSet* exceeds $b^p/avg_{coe}(C_p)$, $w(C_p) := w(C_p) + 1$ and $visitTime(C_p)$ is reset to 0.
- *UnSatSet* is empty indicating there are no unsatisfied PB constraints: for each O_c, $w(O_c) := w(O_c) + 1$.

In the beginning, the weight of each objective constraint is initialized as 0, so that the algorithm would first focus on finding solutions. If the search is trapped in a local optimum, the weight of a PB constraint C_p is only updated when its *visitTime* reaches the threshold $b^p/avg_{coe}(C_p)$. This is because, when the *visitTime* of a PB constraint is low, we consider that this constraint has not been sufficiently explored yet. $b^p/avg_{coe}(C_p)$ represents the average number of exploration attempts required to satisfy a constraint. If a constraint remains unsatisfied after average number of attempts, its weight is increased to ensure the algorithm focuses more on it in subsequent searches, thereby improving the chances of finding a feasible solution.

The weight of the objective constraints are increased only when the current assignment α is a solution (all PB constraints are satisfied under α). Accordingly, if the algorithm frequently visits solutions, then the objective constraints would have a greater chance of increasing their weight. Otherwise, there would be limited opportunities to increase the weight of objective constraints.

Local search algorithms usually search the entire space and focus on exploring some promising spaces using several heuristic strategies [40]. Recently, a general perturbation mechanism called deep optimization has been proposed by Chen et al. [8], which can deeply probe some search regions based on lock and unlock operations and then converge to a new solution quickly. Note that deep optimization is somewhat similar to some classic search frameworks such as large neighborhood search [32]. According to this general framework, we propose a new deep optimization approach DeepOpt for PBO.

The pseudo-code of our proposed DeepOpt is reported in Algorithm 1 and the corresponding trigger conditions of DeepOpt will be displayed in Section 7. We use the $unsat(\alpha)$ function to denote the set of unsatisfied PB constraints under an assignment α. At first, the algorithm uses $UnSatSet$ and $SatSet$ to store unsatisfied and satisfied PB constraints under a perturbation initialization assignment, respectively (Line 1). Afterward, a candidate variable set *CandSet* is initialized to an empty set (Line 2), which stores all unlocked variables in the following search. Note that, in our work, we use *CandSet* to store all unlocked variables which means that these variables can be flipped in the following search phase, while the remaining variables can be seen as locked variables.

5 Deep Optimization for PBO

The DeepOpt usually consists of two phases including a selection phase (Lines 3–15) to generate several candidate local search spaces and a search phase (Lines 16–18) to repair these search spaces.

In the first phase, we flip several variables to achieve the purpose of early preparation. There are two exit conditions in this phase. The first one is that $|CandSet|$ is larger than $\gamma \times n$ where n is the number of variables (Line 3), whereas the second one is that the number of unsatisfied constraints under the current assignment is larger than *MaxHard* (Line 15) where γ and *MaxHard* are two parameters in DeepOpt. At each iteration, if $UnSatSet$ is not empty, the algorithm preferentially chooses a random unsatisfied PB constraint C from $UnSatSet$ (Lines 4–5) and puts all variables of C into *CandSet* (Line 9). For each variable $x \in C$, if x's literal in C is true, it occurs with 50% probability to flip x (Line 8). If the algorithm selects a random satisfied constraint C, the algorithm only adds all variables of C into *CandSet* (Line 14).

In the second phase, the algorithm applies the search process to perturb the assignment α until the limit of iterations L_{opt} is reached (Line 16). The algorithm employs the BMS strategy [5] in the process of selecting a candidate variable, i.e., randomly choosing $|CandSet|/2$ variables to compose a candidate set. This BMS strategy not only enhances the diversity of variable selection and prevents the algorithm from repeatedly choosing the same variable, but also effectively controls computational overhead, as only a subset of candidate variables needs

Algorithm 1: DeepOpt

Input: PBO instance F and a perturbation initialization assignment α
Output: A perturbation solution α of F

1 $UnSatSet := unsat(\alpha)$ and $SatSet := F \setminus UnsatSet$;
2 $CandSet := \emptyset$;
3 **while** $|CandSet| \leq \gamma \times n$ **do**
4 **if** $UnSatSet \neq \emptyset$ **then**
5 select a random unsatisfied PB constraint C;
6 $UnSatSet := UnSatSet \setminus \{C\}$;
7 **for** *each variable* $x \in C$ **do**
8 **if** *x's literal is true && with 50% probability* **then** $\alpha := \alpha$ with x flipped ;
9 $CandSet := CandSet \cup \{x\}$;
10 **else**
11 select a random satisfied PB constraint C;
12 $SatSet := SatSet \setminus \{C\}$;
13 **for** *each variable* $x \in C$ **do**
14 $CandSet := CandSet \cup \{x\}$;
15 **if** $|unsat(\alpha)| > MaxHard$ **then break** ;
16 **for** $step = 0; step < L_{opt}; step{+}{+}$ **do**
17 select a variable x among $|CandSet|/2$ samples from $CandSet$ based on **Flipping Rule**;
18 $\alpha := \alpha$ with x flipped;
19 **return** α;

to be evaluated in each iteration. Afterward, the algorithm flips a variable x among a candidate set based on the flipping rule (Lines 17–18). At last, the algorithm returns the final assignment α (Line 19).

6 Solution Space Exploration Mechanism

Local search algorithms for PBO usually explore the solution space by flipping the selected Boolean variables in the neighborhood that offer significant benefits to find better solutions. The scoring function in Section 3 measures the benefits of flipping each Boolean variable. Typically, flipping variables with positive *score* brings benefits. However, when the *score* of all variables are zero or negative, flipping any variable cannot yield direct benefits. In this situation, selecting the appropriate variable requires employing specific strategies.

In our algorithm, we propose a new framework for exploring the solution space to address the situation described above. The framework primarily comprises two important components: constraint selection and variable flipping. In this section, we first introduce the overall workflow of the proposed framework. Details of important functions in constraint selection stage and variable flipping stage will be presented in the following subsections.

6.1 Main Framework of Solution Space Exploration

When there are no variables with positive *score* (i.e., the algorithm is stuck in a local optimum), the solution space exploration (SSE) algorithm is deployed to search the neighboring space and move to a new candidate solution by determining one or more variables to be flipped.

Algorithm 2: Solution Space Exploration (SSE)

```
   Input :PBO instance F, the current assignment α and the best global optimal assignment α*
1  CSet := ∅;
2  if α* is infeasible then
       /* There must exist unsatisfied PB constraints.                    */
3      sel_pb := PickPB(F, α);
4      CSet := CSet ∪ {sel_pb};
5  else
6      if ∃ unsatisfied PB constraints then
7          select a random unsatisfied PB constraint C;
8          CSet := CSet ∪ {C};
9          if |unsat(α)| ≥ β then
10             for i = 1 to β do
11                 select a random unsatisfied constraint C;
12                 CSet := CSet ∪ {C};
13     else
14         sel_oc := PickObj(F, α, α*);
15         CSet := CSet ∪ {sel_oc};
16 DiversityFlip(F, α, CSet);
```

The solution space exploration algorithm in Algorithm 2 mainly consists of two phases: the constraint selection phase (Lines 1–15) and the variable flipping phase (Line 16). Constraint selection phase aims to limit the size of the neighborhood space, focusing on specific constraints with potential, whereas variable flipping phase involves using heuristic methods to flip well-performing variables from these selected constraints.

Specifically, in the constraint selection phase, the algorithm first initializes the candidate constraint set $CSet$ to be empty (Line 1). Then, it checks whether any feasible solution has been identified up to the current step. If none are found, it indicates that there are unsatisfied PB constraints and the *PickPB*() function (which will be introduced in Section 6.2.2) is used to select an appropriate unsatisfied PB constraint (Lines 2–4). The *PickPB*() function aims to help the algorithm escape from infeasible regions and expedite the identification of a feasible solution. If any feasible solution has already been found at the current step, there are two options. If there exist unsatisfied PB constraints, the proposed constraint sampling selection strategy is adopted to choose one or more constraints for enabling further exploration of the solution space (Lines 6–12). The specific details of the constraint sampling selection strategy will be discussed in the following part. Otherwise, it means only unsatisfied objective constraints exist, the *PickObj*() function (which will be introduced in Section 6.2.2) is called to select an objective constraint that may lead to a better solution (Lines 13–15).

During the variable flipping phase, the algorithm will employ a variable flipping technique *DiversityFlip* to choose one or more variables to flip from the candidate set, which is comprised of variables from the selected constraints (Line 16).

Constraint Sampling Selection Strategy. In this part, we will introduce our proposed constraint sampling selection strategy. Based on having found any feasible solution, if unsatisfied PB constraints exist at the current step, the algorithm adopts two kinds of sampling ways to select PB constraints. The first sampling method is to select a random unsatisfied PB constraint (Line 7). If the number of unsatisfied PB constraints exceeds the parameter value β, then the algorithm activates the second sampling method, i.e., adding β unsatisfied constraints

into a collection *CSet* that stores constraints (Lines 9–12). The natural inspiration for the constraint sampling selection strategy is to select variables from various unsatisfied constraints, providing more search directions.

6.2 Multi-Armed Bandit Model for PBO

In the constraint selection stage, when flipping any Boolean variable cannot improve the current solution, the well-performing algorithms for solving PBO proceed by randomly selecting an unsatisfied PB constraint and then choosing a variable with the highest score to flip. If there are no unsatisfied PB constraints, these algorithms still randomly determine an objective constraint to be satisfied. However, the random constraint selection strategy is not good enough because the high degree of randomness may only have a small probability of finding the correct search direction.

Recently, to handle the issue of similar high randomness in the solution exploration process within the MaxSAT problem, Zheng et al. [44, 45] proposed a new local search algorithm, called *BandHS*, employing two multi-armed bandit (MAB) models including *soft* MAB and *hard* MAB to choose the appropriate soft clauses and literals in hard clauses to satisfy. The MAB models use rewards to evaluate the benefits of selecting soft clauses and literals in hard clauses. Afterward, potential values are utilized to determine which literals and soft clauses are chosen from multiple candidates in the next step to satisfy the hard clauses and achieve a better solution. In our work, we propose a new MAB-PBO method for PBO, which not only trains the MAB-PBO model to select appropriate objective constraints but also extends it to choose suitable PB constraints to be satisfied.

The following subsection will detail the MAB-PBO method, including how to select the PB and objective constraints to be satisfied and how to update the potential values for both PB and objective constraints, respectively.

6.2.1. Reward-Driven Constraint Selection Strategy.

When the search process is trapped in a local optimum, selecting the appropriate constraints to guide the search is crucial. The MAB method provides a powerful tool for making decisions. By employing our proposed MAB-PBO model, we dynamically evaluate the potential value of each constraint, thus identifying the most promising constraints for further exploration. The continuous learning and adjustment of constraint potential values refine the constraint selection process, increasing the chances of finding optimal solutions.

Potential value is used to identify and select constraints more accurately. To implement it, our proposed MAB-PBO model maintains potential values V^{C_p} for each PB constraint C_p and V^{O_c} for each objective constraint O_c, where V^{C_p} and V^{O_c} directly indicate the expected rewards for selecting a PB constraint and an objective constraint, respectively. A larger V^{C_p} implies that selecting the PB constraint has the potential to satisfy more PB constraints and a larger V^{O_c} means that selecting the objective constraint will lead to a better solution.

Considering the balance between exploration and exploitation, similar to previous *soft* and *hard* MAB models, our proposed MAB-PBO model adopts the upper confidence bound method [19] for selecting the appropriate constraints. This method ensures that while exploiting known profitable constraints, the model also explores potentially underutilized constraints that could lead to breakthroughs in solution quality. In the MAB-PBO model, the equation for calculating the upper confidence bound U^{C_p} for each PB constraint C_p is as follows:

$$U^{C_p} = V^{C_p} + \sqrt{\frac{\ln(N^C)}{t^{C_p} + 1}} \tag{4}$$

where N^C represents the total number of times that the algorithm is stuck in a local optimum with unsatisfied PB constraints present, and t^{C_p} records the number of times that PB constraint C_p has been selected in the MAB-PBO model.

Similarly, the equation for calculating the upper confidence bound U^{O_c} for each objective constraint O_c in the MAB-PBO model is as follows:

$$U^{O_c} = V^{O_c} + \sqrt{\frac{\ln(N^O)}{t^{O_c}+1}} \tag{5}$$

where N^O is the total number of times that the algorithm is stuck in a local optimum without unsatisfied PB constraints (i.e., only unsatisfied objective constraints exist), and t^{O_c} is the number of times that objective constraint O_c has been selected in the MAB-PBO model.

The upper confidence bound calculation formula consists of two parts, considering both the reward and the uncertainty. The reward is reflected by the potential value of the constraint (i.e., V^{C_p} and V^{O_c}), while uncertainty indicates the estimation error of the constraint's potential value (i.e., $\sqrt{\frac{\ln(N^C)}{t^{C_p}+1}}$ and $\sqrt{\frac{\ln(N^O)}{t^{O_c}+1}}$). Fewer selections mean lower understanding and higher uncertainty (i.e., $\sqrt{\frac{1}{t^{C_p}+1}}$ and $\sqrt{\frac{1}{t^{O_c}+1}}$). Uncertainty encourages the algorithm to explore constraints that are selected less frequently, to better estimate their true rewards. Thus, the algorithm favors the constraint with the highest upper confidence bound, balancing exploration and exploitation by choosing constraints with high rewards and significant uncertainty.

6.2.2. Potential Value Updating Method.

The update of potential values is a crucial mechanism for optimizing and adapting the MAB-PBO method to complex environments. It aims to improve the effectiveness and rewards of constraint selection through continuous learning and adjustment. Therefore, the reward function, serving as feedback after a constraint is selected, is crucial in its design. Subsequently, we first introduce the design of the reward functions for PB constraints and objective constraints. Then, we discuss the method for updating the potential values of the constraints.

The MAB-PBO model for selecting PB constraints is to quickly find a feasible solution by effectively reducing the total violation value through the selection of high-quality PB constraints. Therefore, the change in the total violation values before and after selecting a PB constraint is taken as the reward. Furthermore, the MAB-PBO model for selecting objective constraints seeks to find better objective function values by selecting high-quality objective constraints. The change in the value of the objective function before and after selecting the objective constraints is considered as the reward.

Assuming that α denotes the current assignment when calling the *SSE* function and α' represents the assignment of the last *SSE* function called. Intuitively, $voil(\alpha')$ and $voil(\alpha)$ represent the degree of violation under these assignments. Furthermore, $voil(\alpha')$ - $voil(\alpha)$ can be simply considered as the reward. However, we think the difficulty of reducing a significant degree of violation to a moderate level is different from reducing a minor violation to an even lower level. For example, reducing the violation value from 100 to 90 and from 10 to 0 should not have the same reward, as the latter is more significant. To address this issue, the reward function of the MAB-PBO model for each PB constraint C_p is designed as follows:

$$R^{C_p}(\alpha, \alpha') = \frac{voil(\alpha') - voil(\alpha)}{voil(\alpha') + 1} \tag{6}$$

Similarly, $obj(\alpha')$, $obj(\alpha)$ and $obj(\alpha^*)$ represent the objective function values under α', α and α^* where α^* is the best global optimal assignment found so far. The reward function of the MAB-PBO model for each objective constraint O_c is designed as follows:

$$R^{O_c}(\alpha, \alpha', \alpha^*) = \frac{obj(\alpha') - obj(\alpha)}{obj(\alpha') - obj(\alpha^*) + 1} \tag{7}$$

Algorithm 3: PickPB

Input : PBO instance F, the current assignment α
Output: A selected PB constraint sel_{pb}

1 $U_*^C := -\infty$;
2 update the potential value V^{C_p} for each PB constraint C_p according to Eq.8.;
3 $\alpha' := \alpha$;
4 $N^C := N^C + 1$;
5 **for** $i = 0; i < NumSample; i{+}{+}$ **do**
6 select a random unsatisfied PB constraint C_l;
7 calculate U^{C_l} according to Eq.4;
8 **if** $U^{C_l} > U_*^C$ **then**
9 $U_*^C := U^{C_l}, sel_{pb} := C_l$;
10 $t^{sel_{pb}} := t^{sel_{pb}} + 1$;
11 **return** sel_{pb};

In the MAB-PBO model, the reward value is not solely dependent on the act of selecting a PB constraint or an objective constraint but also includes the impact of earlier actions. Therefore, similar to *soft* and *hard* MAB models, we adopt the delayed reward approach [2] to update several recently selected constraints when a reward is obtained.

Specifically, $R^{C_p}(\alpha, \alpha')$ is the reward obtained from selecting a PB constraint and is calculated by Eq.6. The set $\{C_1, C_2, \ldots, C_d\}$ records the latest selected PB constraints where the positions of constraints in the set are arranged in ascending order of the selection times and a parameter d is used to control its size. Among this set, C_d is the most recent one. Due to the different temporal order of constraint selection, distinct reward values should be assigned accordingly. The equation for updating the estimated values of d PB constraints is as follows:

$$V^{C_p} = V^{C_p} + \lambda^{d-p} \cdot R^{C_p}(\alpha, \alpha'), p \in \{1, \ldots, d\}, \tag{8}$$

where λ is the reward discount factor.

Similarly, $R^{O_c}(\alpha, \alpha', \alpha^*)$ is the reward obtained by selecting an objective constraint and is calculated using Eq.7. $\{O_1, O_2, \ldots, O_d\}$ is the set storing the recently selected objective constraints in ascending order of the selection time. Among this set, O_d is the most recent one. The equation for updating the estimated values of d objective constraints is as follows:

$$V^{O_c} = V^{O_c} + \lambda^{d-c} \cdot R^{O_c}(\alpha, \alpha', \alpha^*), c \in \{1, \ldots, d\}, \tag{9}$$

where λ is the reward discount factor.

The procedures for selecting PB constraints and objective constraints using the PB and objective MAB-PBO models are detailed in Algorithms 3 and 4, respectively. The main difference between *PickPB* and *PickObj* is the formula used to calculate the constraint reward function when updating the constraint potential value. Based on the reward function of the objective constraint (i.e., Eq. 7), *PickObj* requires the additional input parameter α^*, which represents the best assignment obtained by the algorithm so far.

Although *PickPB* and *PickObj* are two independent algorithms, they are introduced together due to the similarity in their algorithmic processes.

First, the algorithm initializes the recorded maximum upper confidence bound value U_*^C or U_*^O of PB or objective constraint to $-\infty$ (Line 1). After then, the potential values of constraints are updated according to

Algorithm 4: PickObj

Input : PBO instance F, the current assignment α and the best global optimal assignment α^*
Output : A selected objective constraint sel_{oc}

1 $U_*^O := -\infty$;
2 update the potential value V^{O_c} for each objective constraint O_c according to Eq.9.;
3 $\alpha' := \alpha$;
4 $N^O := N^O + 1$;
5 **for** $i = 0; i < NumSample; i{+}{+}$ **do**
6 select a random unsatisfied objective constraint O_l;
7 calculate U^{O_l} according to Eq.5;
8 **if** $U^{O_l} > U_*^O$ **then**
9 $U_*^O := U^{O_l}$, $sel_{oc} := O_l$;
10 $t^{sel_{oc}} := t^{sel_{oc}} + 1$;
11 **return** sel_{oc};

Equation 8 or 9 (Line 2). α' is used to record the current assignment α (Line 3). Since at least one variable will be flipped after each iteration, α will change in the next update of the constraint potential value. Thus, α' is used to retain the previous assignment. Additionally, the value of N^C or N^O is incremented by 1 (Line 4).

Lines 5–9 demonstrate selecting the constraint with the highest upper confidence bound value from the candidate constraints. Since the PBO problem contains a large number of PB and objective constraints, searching all constraints is inefficient. Therefore, we adopt a sample approach to limit the number of candidate constraints based on the size of the sampling parameter *NumSample*. Then, the algorithm updates the number of times $t^{sel_{pb}}$ or $t^{sel_{oc}}$ for the PB constraint sel_{pb} or the objective constraint sel_{oc} (Line 10). Finally, the algorithm obtains the selected constraint sel_{pb} or sel_{oc} (Line 11).

Our proposed MAB-PBO model for selecting objective constraints adopts the similar approach as *soft* MAB model [44]. However, the MAB-PBO model for selecting PB constraints significantly differs from the *hard* MAB model [45], primarily in two aspects. In terms of solving MaxSAT, the algorithm selects a random falsified clause and chooses a suitable literal from the clause based on *hard* MAB method. For solving PBO, our algorithm picks a falsified PB constraint according to the proposed MAB-PBO method and uses our two-level scoring mechanism to select a good literal. Thus, it is easily observed that the objects that need to be decided upon are different. This indicates that our MAB-PBO model specifically considers PB constraints as decision objects, strategically selecting potential constraints to expedite the escape from unsatisfiable domains. Secondly, the reward functions involved in the models are different. The *hard* MAB model uses the change in the number of falsified hard clauses before and after an assignment changes as the reward feedback, whereas the MAB-PBO model uses the change in the sum of violations of all PB constraints before and after an assignment change as the reward feedback.

6.3 Diversity Flip Strategy

During the variable flipping stage, we develop a novel *DiversityFlip* method, and the pseudo-code is outlined in Algorithm 5. First, we introduce a new scoring function, which is used in our *DiversityFlip* method. When flipping two variables x_s and x_z simultaneously, $score_t(x_s, x_z)$ is defined as the sum of the decrease of the total penalty of unsatisfied PB constraints and objective constraints. Although it is easy to see the computation complexity of $score$ is quite lower than $score_t$, $score_t$ can find a better flipping operation compared to $score$. The method of selecting more candidate elements simultaneously has also been used in some different NP-hard

problems [36, 43]. Moreover, in the *DiversityFlip* method, we only consider $score_t$ in a special case that the number of literals in the selected constraint C is 2 (i.e., $|L(C)| = 2$).

In the phase of variable flipping, the number of constraints in $CSet$ is either one or β. When $|CSet|$ equals 1, there are three scenarios: if C in $CSet$ is a unit clause (i.e., $|L(C)| = 1$), the algorithm flips the only variable x in C (Lines 3–4); If C is a binary clause (i.e., $L(C) = \{x_1, x_2\}$), the algorithm attempts to find two variables x_i and x_j that are neighbors of x_1 and x_2, respectively. The term *neighbor* means that x_1 and x_i, as well as x_2 and x_j are involved in the same constraints. The algorithm tries to find two pairs of variables $\{x_1, x_i\}$ and $\{x_2, x_j\}$ with the largest $score_t$ value (Lines 6–7). If there exists a positive flipping operation among these two pairs, the algorithm flips a pair with the better $score_t$ value (Lines 8–11). Otherwise, the algorithm still flips only one variable among x_1 and x_2 based on the flipping rule (Lines 12–13); If $|L(C)| > 2$, the algorithm will sample $|L(C)|/2$ variables into $VSet$ and flip the best variable among them (Lines 14–17). In the subsequent process, since there are β constraints in $CSet$, the algorithm will collect some samples from each C in $CSet$ into $VSet$ (Lines 19–20). At last, the algorithm selects and then flips the best variable x from $VSet$ (Lines 21–22).

The intuitive explanations behind *DiversityFlip* come from two aspects. First, single flipping mechanism for local search is easy to fall into a local optimum, while sampling strategy can explore the solution space more effectively in a look-ahead way. In addition, we especially focus on the case of two literals, to keep running time low but a good performance.

7 NuPBO-DeepOpt+ Algorithm

In this section, we propose an effective local search algorithm *NuPBO-DeepOpt+* for PBO, whose main framework is presented in Algorithm 6. The *NuPBO-DeepOpt+* is mainly divided into the initialization and search phases.

In the initialization phase (Lines 1–6), to obtain an initial assignment α, all variables are set to 0. The best assignment α^* and the assignment α' recorded from the last execution of *SSE* function are both initialized to α. The objective value obj^* is set to $+\infty$. The potential values V^{C_p} and V^{O_c} for each PB constraint and objective constraint are set to 1. The counters t^{C_p} and t^{O_c} are used within the *SSE* function to record the selection times for each PB constraint and objective constraint are initialized to 0. The total selection counters N^C and N^O within the *SSE* function are also initialized to 0. Additionally, the algorithm initializes the related weight information according to the modified initialization rule described in Section 4.

In the following, there is an outer loop (Lines 7–34) and an inner loop (Lines 10–33). During the search, whenever a better feasible assignment is obtained, α^* and obj^* are updated accordingly (Line 15). After each inner loop, the algorithm will restart with an all-zero assignment (Line 34). Finally, if the optimal solution is feasible, the algorithm returns α^* and $obj*$. Otherwise, *no feasible solution* is returned when reaching a time limit (Lines 35–38).

In each inner loop ($step < L$, the default $L = 2 \times 10^8$.), the algorithm searches for a local optimal assignment α. The algorithm uses $GoodSet$ to store variables whose $score$ is larger than 0 (Line 18). If $GoodSet$ is not empty, the algorithm flips a candidate variable based on the proposed flipping rule (Lines 19–21). Otherwise, it means that the algorithm tramps into local optimum. To escape local optima, the algorithm uses the modified updating rule to update the corresponding weight value (Line 23). Afterwards, the *SSE* algorithm is used to flip one or more variables to search the neighboring space in order to obtain a better solution (Line 24).

Trigger Conditions of DeepOpt: In the below part, we will introduce some trigger conditions of DeepOpt in our proposed algorithm. At first, three variables are defined as below.

1) $unhard$ is used to denote the number of unsatisfied PB constraints. Before each inner loop, $unhard$ is initialized to the number of unsatisfied constraints under the current assignment (i.e., $|unsat(\alpha)|$) (Line 9). In the inner loop, whenever $|unsat(\alpha)| < unhard$, $unhard$ is updated accordingly (Lines 11–12). After calling the DeepOpt method, $unhard$ will be updated by $|unsat(\alpha)|$ (Line 33).

Algorithm 5: DiversityFlip

```
Input : PBO instance F, an assignment α and the selected constraint set CSet
Output: A modified assignment α
1  if |CSet| == 1 then
2      select only constraint C in CSet;
3      if |L(C)| == 1 then
4          select only variable x in C and α := α with x flipped;
5      else if |L(C)| == 2 then
           /* Two variables x_1 and x_2 in C                                  */
6          select a variable x_i with the largest score_t(x_1, x_i) value, breaking ties randomly;
7          select a variable x_j with the largest score_t(x_2, x_j) value, breaking ties randomly;
8          if score_t(x_1, x_i) > 0 || score_t(x_2, x_j) > 0 then
9              if score_t(x_1, x_i) > score_t(x_2, x_j) then
10                 α := α with x_1 and x_i flipped;
11             else α := α with x_2 and x_j flipped;
12         else
13             select a variable x among x_i and x_j based on Flipping Rule and α := α with x flipped;
14     else
15         select |L(C)|/2 samples from L(C) and put them into VSet;
16         select a variable x from VSet based on Flipping Rule;
17         α := α with x flipped;
18 else
19     for each constraint C ∈ CSet do
20         select |L(C)|/2 samples from L(C) and put them into VSet;
21     select a variable x from VSet based on Flipping Rule;
22     α := α with x flipped;
23 return α;
```

2) $step_{opt}$ records the non-improvement steps after the initialization phase or the last DeepOpt operation. Before each inner loop, the algorithm sets $step_{opt}$ to 1 (Line 8). In each iteration of the inner loop, $step_{opt}$ is increased by 1 (Line 25). When the algorithm obtains a better assignment (Line 14) or the algorithm calls the DeepOpt method (Line 29 or 32), $step_{opt}$ will be set to 1 (Line 16 or 33). If $|unsat(\alpha)| < unhard$, then $step_{opt}$ will be cut in half (Line 13).

3) *coff* is used to control the frequency of using the DeepOpt method. Before each inner loop, *coff* is initialized to 1 (Line 8). When the algorithm finds a better assignment, the value of *coff* is divided by 2 to reduce the frequency of perturbations (Line 17). If $|unsat(\alpha)|$ is smaller than parameter *MinHard*, it means unsatisfied PB constraints are few enough to consider perturbations, avoiding tramping into a local optimum. The value of *coff* will be doubled with a 50% probability (Line 28). During this process, parameter δ controls the maximum value of *coff*.

The algorithm judges whether to call the DeepOpt method under each ($coff \times MinStep$) iteration (Line 26). If $|unsat(\alpha)| \leq MinHard$, the algorithm calls the DeepOpt method with a 50% probability (Line 27). Otherwise, if α^* is feasible, the algorithm will use α^* as a perturbation initialization assignment and then employ the DeepOpt method (Lines 30–32).

Algorithm 6: NuPBO-DeepOpt+

Input : PBO instance F and cutoff time $cutoff$
Output : An assignment α^* of F and its objective value obj^*

α := all variables are set to 0;
$\alpha^* := \alpha' := \alpha$ and $obj^* := +\infty$;
$V^{C_p} := 1$ and $t^{C_p} := 0$, for each PB constraints C_p;
$V^{O_c} := 1$ and $t^{O_c} := 0$, for each objective constraints O_c;
$N^C := N^O := 0$;
initialize the weight value of the PB constraints and the objective constraints through the modified initialization rule;
while $elapsed\ time < cutoff$ **do**
    $step_{opt} := 1$ and $coff := 1$;
    $unhard := |unsat(\alpha)|$;
    for $step = 1; step < L; step{+}{+}$ **do**
        if $|unsat(\alpha)| < unhard$ **then**
            $unhard := |unsat(\alpha)|$;
            $step_{opt} := step_{opt}/2$;
        if α *is feasible* && $obj(\alpha) < obj^*$ **then**
            $\alpha^* := \alpha$ and $obj^* := obj(\alpha)$;
            $step_{opt} := step := 1$;
            if $coff \neq 1$ **then** $coff := coff/2$;
        $GoodSet := \{x \mid score(x) > 0\}$;
        if $GoodSet \neq \emptyset$ **then**
            select a variable x in $GoodSet$ based on **Flipping Rule**;
            $\alpha := \alpha$ with x flipped;
        else
            update the weight of each constraint based on the **modified update rule**;
            $SSE(F, \alpha, \alpha^*)$;
        $step_{opt} := step_{opt} + 1$;
        if $step_{opt} \% (coff \times MinStep) == 0$ **then**
            if $|unsat(\alpha)| \leq MinHard$ && *with 50% probability* **then**
                if $coff \neq \delta$ **then** $coff := coff \times 2$;
                $DeepOpt(F, \alpha)$;
            else if α^* *is feasible* **then**
                $\alpha := \alpha^*$;
                $DeepOpt(F, \alpha)$;
            $unhard := |unsat(\alpha)|$ and $step_{opt} := 1$;
    α := restart with an all-zero assignment;
if α^* *is feasible* **then**
    return (α^*, obj^*);
else
    return *no feasible solution*;

Time Complexity Analysis: In this part, we analyze the time complexity of one iteration of the inner loop of the proposed *NuPBO-DeepOpt+* algorithm (i.e., Lines 10–33). Assume that the number of constraints is m and the number of variables is n. One iteration primarily consists of two components, including the variable selection and flipping part (Lines 19–24) and the deep optimization for escaping local optimum (Lines 25–33).

In the variable selection and flipping part, line 20 involves the process of selecting variables when the *GoodSet* is not empty. The time complexity is $O(n)$, due to the comparison of data with a length of n in the worst case. Lines 23–24 outline the variable selection process when the *GoodSet* is empty. First, the algorithm updates the weights of the constraints, and correspondingly adjusts the scores of the variables within those constraints due to changes in weights. Then, constraints are chosen from the candidate set of unsatisfied constraints and variables are selected from them. The entire process has a worst-case time complexity of $O(m \times n)$. Therefore, the time complexity of the variable selection and flipping part is $O(m \times n)$.

In the deep optimization part, several candidate local search spaces are generated through a loop that continues until *CandSet* is less than or equal to $\gamma \times n$. Each iteration of the loop may involve processing all variables in the constraints, thus, in the worst case, the overall time complexity is $O(\gamma \times n^2)$. In summary, considering the variable selection and flipping part as well as the deep optimization part, the time complexity of one iteration of the inner loop is $O(m \times n + \gamma \times n^2)$.

8 Differences Between *NuPBO-DeepOpt+*, *DeepOpt-PBO-v1* and *NuPBO*

The newly proposed *NuPBO-DeepOpt+* can be considered a combined version of *DeepOpt-PBO-v1* and *NuPBO*, which were earlier published in our proposed conference papers [46] and [11], respectively. There are five main differences among these three algorithms.

An integrated version. Based on prior experiments, we observe that *DeepOpt-PBO-v1* performs best in three large-scale real-world application benchmarks, and *NuPBO* excels in three standard benchmarks, including the PB competition 2016, PB competition 2024 and MIPLIB. Additionally, since the strategies employed by the two algorithms are not in conflict, thus we proceed with strategy integration. Specifically, we integrate the primary scoring function and the new weighting scheme from *NuPBO* into *DeepOpt-PBO-v1*, which includes a two-level selection strategy, a deep optimization strategy, a constraint sampling selection strategy, and a diversity flip method.

Apply reinforcement learning techniques. We newly propose the application of reinforcement learning method, designing a multi-armed bandit model to select appropriate PB constraints and objective constraints to be satisfied when stuck in the local optimum. In the early versions of *DeepOpt-PBO-v1* and *NuPBO*, both utilized random methods.

Combined with a preprocessor. Neither *DeepOpt-PBO-v1* nor *NuPBO* algorithms apply preprocessing technology. We combine *NuPBO-DeepOpt+* with a preprocessor to observe performance changes.

Add new comparison algorithms and benchmarks. *DeepOpt-PBO-v1* only compared one local search algorithm, namely *LS-PBO*, and two exact PBO solvers, *RoundingSat* and *PBO-IHS*. On top of that, *NuPBO* additionally included two MIP solvers, *SCIP* and *Gurobi*, both of which are also incorporated in the comparison of *NuPBO-DeepOpt+*. Furthermore, we introduce four recently proposed local search algorithms for comparison: *DeciLS-PBO*, *NuPBO*$_{ss}$, *DLS-PBO*, and *OraSLS*. In total, by combining *DeepOpt-PBO-v1*, *DeepOpt-PBO-v2*, and *NuPBO*, this work compares twelve algorithms, covering state-of-the-art local search methods, exact solvers, and MIP-based approaches. In addition, we conduct experiments on the KNAP benchmark, which was not included in *NuPBO*, as well as on the latest benchmark used in the most recent PB competition 2024.

Comprehensive validity testing. In the original *DeepOpt-PBO-v1* version, strategy effectiveness was only tested on three benchmarks: WMCB, SAP, and WSNO. We have expanded the experiments to cover all eight benchmarks. Overall, we evaluate the effectiveness of all proposed strategies across all benchmarks.

In fact, the paper [46] presents two versions, including *DeepOpt-PBO-v1* and *DeepOpt-PBO-v2*, with the sole difference being that the v2 version adopts a restart strategy after a number of steps (defaulted to 10000 steps). The reason for not integrating the v2 version is that the restart strategy leads to poorer performance as *NuPBO-DeepOpt+* can find feasible solutions more quickly, and frequent restarts would lead to restarting without thorough exploration.

9 Experimental Evaluation

In this section, we first introduce the eight selected benchmarks, twelve state-of-the-art competitors, and the adopted experimental setup. Then, we carry out extensive experiments to evaluate the performance of our proposed algorithm.

9.1 Experiment Preliminaries

We selected all used instances from [26, 14] as well as from the recent PB competition 2024. To be specific, we considered 4146 instances obtained from three application benchmarks and five standard benchmarks:

- **MWCB:** 24 instances from the minimum-width confidence band problem [3]. These MWCB instances were obtained based on the MIT-BIH arrhythmia database[2]. The number of variables ranges from 974000 to 11381750, and the number of constraints ranges from 591827 to 10522608.
- **WSNO:** 18 instances from the wireless sensor network optimization problem [23, 24]. We used the same encoding as previous work [26] to obtain an optimization version of WSNO. The number of variables ranges from 58580 to 2311113, and the number of constraints ranges from 214679 to 26701722.
- **SAP:** 21 instances from the seating arrangements problem [1]. These instances were originally proposed in the MaxSAT Evaluation 2017. The number of variables ranges from 1650 to 11550, and the number of constraints ranges from 7756 to 108474.
- **PB2016:** 1600 OPT-SMALL-INT instances from PB competition 2016[3]. The PB2016 benchmark is often considered as the main target for comparing against some other PB solvers [14, 34]. The number of variables ranges from 2 to 1041679, and the number of constraints ranges from 3 to 1883036.
- **PB2024:** 478 OPT-LIN instances form the the most recent PB competition 2024[4]. The number of variables ranges from 4 to 5121998, and the number of constraints ranges from 1 to 4920807.
- **MIPLIB:** The 0-1 integer linear programming optimization benchmark contains 267 instances from the mixed integer programming library MIPLIB 2017[5]. The number of variables ranges from 2 to 12471400, and the number of constraints ranges from 2 to 15270211.
- **CRAFT:** 955 crafted combinatorial instances are provided in the literature [39]. The number of variables ranges from 16 to 6924842, and the number of constraints ranges from 11 to 469920.
- **KNAP:** The Knapsack benchmark consists of a total of 783 instances [29]. The number of variables ranges from 20 to 50000, and the number of constraints is always 1.

The proposed algorithm is compared against twelve state-of-the-art solvers, including eight local search algorithms (i.e., *LS-PBO*, *DeciLS-PBO*, *DeepOpt-PBO-v1*, *DeepOpt-PBO-v2*, *NuPBO*, *NuPBO$_{ss}$*, *DLS-PBO*, and *OraSLS*), two exact PBO solvers (i.e., *RoundingSat* and *PBO-IHS*) and two MIP solvers (i.e., *SCIP* and *Gurobi*).

- ***LS-PBO*:** one of the basic local search algorithms for solving PBO, with the recent local search algorithms being improvements upon it [26].

[2] http://physionet.org/physiobank/database/mitdb/
[3] http://www.cril.univ-artois.fr/PB16/
[4] https://www.cril.univ-artois.fr/PB24/
[5] http://miplib.zib.de

- ***DeciLS-PBO*:** a recently proposed local search algorithm builds on *LS-PBO*, incorporating initialization and new constraint selection strategies [21].
- ***DeepOpt-PBO-v1*:** one of the state-of-the-art local search algorithms for solving PBO, it performs best on three large-scale application benchmarks [46].
- ***DeepOpt-PBO-v2*:** a revised version of *DeepOpt-PBO-v1*, incorporating a restart method [46].
- ***NuPBO*:** a recently proposed local search algorithm is built upon *LS-PBO*, incorporating a new scoring function and a new weighting scheme [11].
- ***NuPBO*$_{ss}$:** building upon *NuPBO*, *NuPBO*$_{ss}$ integrates unit propagation technique to escape from local optimum [9].
- ***DLS-PBO*:** based on *LS-PBO*, *DLS-PBO* proposes a dynamic scoring mechanism to further improve the search process [10].
- ***OraSLS*:** it can be viewed as a non-traditional stochastic local search, making use of the complete approach as a decision oracle and adopting various heuristic options to vary search behavior [20].
- ***RoundingSat*:** a recent complete PBO solver combining core-guided search with cutting planes reasoning [14].
- ***PBO-IHS*:** a recent complete PBO solver based on implicit hitting set [34].
- ***SCIP*:** one of the fastest non-commercial solvers for MIP. The version used is 7.0.3 [4].
- ***Gurobi*:** one of the most powerful commercial MIP solvers [18]. The version used is 10.0.2.

The codes of all competitors were kindly provided by the authors and we employ the default parameters in the corresponding literature, respectively. Our code is available at https://github.com/yiyuanwang1988/NuPBO-DeepOpt-plus. All algorithms are implemented in C++ and compiled by g++ with -O3 option. All the algorithms are run on Intel Xeon Gold 6238 CPU @ 2.10GHz with 512GB RAM under CentOS 7.9.

Table 2. Tuned parameters of our proposed algorithm.

Parameter	Range	Final value
DeepOpt		
MinStep	$\{10^3,10^4,10^5,10^6,10^7\}$	10^6
δ	{32,64,128,256,512}	128
MinHard	{5,10,15,20,25}	10
γ	{0.02,0.05,0.08,0.11,0.13}	0.05
MaxHard	{30,50,70,90,110}	50
L_{opt}	{10,30,50,70,90}	50
MAB model		
λ	{0.6,0.7,0.8,0.9,1}	0.9
d	{10,20,30,40,50}	20
NumSample	{10,20,30,40,50}	20
SSE		
β	{50,100,150,200,250}	100

According to our preliminary experiments by using the automatic configuration tool irace [27], the specific parameter settings are shown in Table 2. Specifically, since these benchmarks have different scales, we built a training set and randomly selected 10 instances from the corresponding tested benchmark. The tuning process is given a budget of 5000 runs for the training set with a time budget of 3600s per run.

For the application benchmarks, our proposed algorithm and the seven comparative pure local search algorithms are each run 20 times on every instance using different random seeds ranging from 1 to 20, whereas the two exact

PBO solvers and the two MIP solvers are run only once per instance. Specifically, since the *OraSLS* algorithm is a non-traditional local search that incorporates a complete oracle, thus it runs with a random seed as well. For the other five standard benchmarks (i.e., PB2016, PB2024, MIPLIB, CRAFT and KNAP), following the settings of previous works [14, 34], all the algorithms are run only once on each instance. We test the algorithms with a time limit of 3600 seconds.

For the application benchmarks, we use *min* to denote the best solution value found and *avg* to denote the average value over the 20 runs. If the algorithm fails to find a feasible solution, it will be marked as N/A. For all the benchmarks, we report the number of instances where the algorithm finds the best solution value among all algorithms, denoted by $\#win$. There are some unsatisfied instances in the PB2016, PB2024 and MIPLIB benchmarks. The two exact PBO solvers and two MIP solvers can guarantee the optimality of the solutions they obtain and thus can prove some of these unsatisfied instances, whereas *NuPBO-DeepOpt+* and other local search algorithms cannot do it because these algorithms belong to incomplete algorithms. For the above case, following the similar method from the literature [26], if all the algorithms fail to obtain any feasible solution for such an unsatisfied instance, then $\#win$ value of all the algorithms for this instance needs to be increased by 1. Moreover, $NuPBO_{ss}$ only supports instances with integers no larger than 32 bits, whereas PB2024 includes several instances with large integers exceeding this limit. As a result, these instances are all counted as unsolved by $NuPBO_{ss}$. The bold value indicates the best solution value obtained by all the algorithms. In detail, the bold value of each *avg* column indicates the best average solution when some algorithms obtain the same minimal solution values. For one instance, if only one algorithm finds the best minimal solution value, only its corresponding *min* column should be marked.

9.2 Experiment Results

Note that for the application benchmarks, two exact PBO solvers (i.e., *RoundingSat* and *PBO-IHS*), and two MIP solvers (i.e., *SCIP* and *Gurobi*) can obtain the same best solution as our proposed algorithm for only 4 instances of the WSNO benchmark. Although *Gurobi* can find feasible solutions for MWCB, our algorithm significantly outperforms it. Additionally, none of the four solvers can find a feasible solution for SAP. Thus, we do not report the detailed results of these four solvers on three application benchmarks. We mainly compare *NuPBO-DeepOpt+* with eight local search algorithms, including *LS-PBO*, *DeciLS-PBO*, *DeepOpt-PBO-v1*, *DeepOpt-PBO-v2*, *NuPBO*, $NuPBO_{ss}$, *DLS-PBO* and *OraSLS*. The experimental results on the application benchmarks are presented in Tables 3–5. Experimental results demonstrate that our proposed algorithm significantly outperforms all competing methods on the MWCB and SAP benchmarks. In particular, on MWCB, *NuPBO-DeepOpt+* improves the best-known solution for 19 instances, and on SAP it obtains the best-known solution for all instances and updates the best-known solution for 15 instances. For the WSNO benchmark, all local search algorithms can obtain the same best solution value, except for the *DeepOpt-PBO-v2*, $NuPBO_{ss}$ and *OraSLS* algorithms, which fail to achieve this value in four, five and fourteen instances. Furthermore, *NuPBO* can find a minimal average solution for 17 instances, while our proposed algorithm does it for only 2 instances. Our proposed algorithm adopts some perturbation mechanisms, which may make it difficult for the algorithm to steadily obtain a good solution.

Table 6 summarizes the results across all benchmarks. The experimental results show that, out of a total of 4146 instances, *NuPBO-DeepOpt+* achieves 3382 optimal solutions, ranking second overall, only behind *Gurobi* with 3632, and outperforming all other competing methods. Among all local search algorithms, *NuPBO-DeepOpt+* outperforms the currently best local search algorithm, *DLS-PBO*, by 208 optimal solutions. Specifically, *NuPBO-DeepOpt+* dominates all competitors on the real-world application benchmarks, i.e., MWCB, WSNO, and SAP. On the PB2024, MIPLIB, and CRAFT benchmarks, *Gurobi* outperforms other algorithms by obtaining more optimal solutions, while *NuPBO-DeepOpt+* ranks second. On the PB2016 benchmark, *Gurobi* also rankes first, with $NuPBO_{ss}$ outperforming *NuPBO-DeepOpt+* by 14 instances. However, when comparing these two algorithms

independently, *NuPBO-DeepOpt+* outperforms *NuPBO*$_{ss}$ by 25 instances. For the KNAP benchmark, both *SCIP* and *Gurobi* perform equally, with *PBO-IHS* following closely. In summary, across eight benchmarks, *NuPBO-DeepOpt+* shows competitive performance relative to *Gurobi* and significantly outperforms all other solvers, demonstrating its broad applicability and exceptional performance across various problem types.

Table 3. Experiment results on MWCB.

Instance	NuPBO-DeepOpt+		LS-PBO		DeciLS-PBO		DeepOpt-PBO-v1		DeepOpt-PBO-v2		NuPBO		NuPBO$_{ss}$		DLS-PBO		OraSLS
	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	min
1000_200_90	104165	105696.45	110450	111314.25	109795	111203.15	**103363**	104188.35	103430	104233	109798	110541.1	110251	110874.25	111568	113834.15	159198
1000_250_90	140231	143305.2	148181	149828.65	147075	149663.75	**140223**	141182.2	140237	141194.9	147936	149719.75	148770	149885.5	152545	153508.15	210446
1200_200_90	**102948**	105166.05	110993	112897.1	110396	113253.35	104063	104572.25	104063	104589.1	110689	111325.65	110609	111266.1	114596	116172.15	173397
1200_250_90	**140623**	142167.35	150212	152888.7	149924	153501.15	141211	142310.6	141238	142343.2	149456	150358.4	150275	151642.95	155741	157224.45	217252
1400_200_90	**103730**	105373.9	110792	112975.65	110085	112704.1	103948	104867.35	104057	104901.55	110025	110827.15	110382	111159.5	114945	115893.25	170806
1400_250_90	**140436**	142410.4	150981	152932.9	150570	152466.65	141567	142675.7	141746	142722.15	149255	150351.85	150450	152322.5	155533	156328.1	241337
1600_200_90	**116456**	118365.6	136944	143371.9	135717	147558.3	119226	120182.4	119226	120215.9	135982	138379.95	135583	138349.32	159799	163320.55	257483
1600_250_90	**159153**	161707.8	183797	196547.75	186249	202667.7	162651	163893.3	162656	163937.1	182895	186027.6	182092	185385.6	214306	219564.9	325181
1800_200_90	**202310**	208965.05	219536	224144.95	220705	224907.7	203135	205888.75	203135	205959.2	222093	225171.15	240483	244125.45	237514	239527.35	309438
1800_250_90	256383	263577.9	276336	283192.95	279648	281572.6	**253073**	257501.75	253078	257715.25	278489	282412.3	292766	298855	297742	302349.75	400547
2000_200_90	**226384**	231835.5	246109	250958.4	244478	250747.35	227294	229591.95	227424	229676.1	247583	251606.7	247433	251587.7	257690	263755.85	343568
2000_250_90	288812	292844	309645	314528.5	307565	312254.05	**286970**	**289889.4**	**286970**	290058.1	311016	315093.25	310834	315010.25	322469	327340.25	422152
1000_200_95	**111955**	114368.3	117064	117945.7	116537	117886.9	113384	114749.8	113501	114855.95	119380	119922.1	119408	119855.7	120901	122377.75	158323
1000_250_95	**150622**	152252	156543	157976.1	156144	158013.25	151882	153581.05	152007	153622	159081	160078.95	158561	160329.1	160539	161940.85	204533
1200_200_95	**112457**	114323.6	118045	119544.4	118361	119601.05	114585	115920.2	114675	115975	119538	120447.8	119956	120482.45	124022	124901.05	186380
1200_250_95	**151261**	153649.3	159310	161454.1	159357	161123.05	153104	155765.8	153138	155798.35	161975	162815.25	162445	163326.3	165767	167933.9	207816
1400_200_95	**112828**	114517.1	118913	119779.4	118607	119823.65	114195	115231.3	114195	115305.95	120306	121034.1	121024	121554.45	123667	125480.8	174026
1400_250_95	**153485**	154812.35	161658	162917.95	161040	162968.45	155158	156149.65	155174	156180.15	163357	164491.9	164103	165050.45	167228	169888.4	216082
1600_200_95	**167122**	170491.05	185707	190763.55	185265	192569.05	168271	171985.9	168271	172024.6	184493	186574.15	185166	187311.45	204283	211180	280474
1600_250_95	**212910**	217434.65	236547	244060.35	237184	243911.45	215266	218013.5	215316	218069.55	235060	237213.1	234794	238967.45	263053	271327.2	364859
1800_200_95	**233317**	238141.45	251744	256988.75	253035	257138.9	238699	241690.55	238702	241779.85	254501	256084.65	255259	257460.7	261129	266028.25	323078
1800_250_95	**293725**	296989.6	314968	318961.25	314056	319030.4	297981	302487.3	297981	302927.25	316026	318185.8	316696	320153.6	326295	330742	396487
2000_200_95	258679	263034.65	272832	277406.7	273052	279597.6	**257696**	**262062.9**	**257696**	262113.6	276525	278926.75	276419	278128.75	284007	286454.1	365352
2000_250_95	**322346**	327809.7	340859	346007.95	340105	347536.65	324379	328952.6	324379	329032.9	347059	348493.85	347059	348493.1	353838	355808.45	429033

Table 4. Experiment results on WSNO.

Instance	NuPBO-DeepOpt+		LS-PBO		DeciLS-PBO		DeepOpt-PBO-v1		DeepOpt-PBO-v2		NuPBO		NuPBO$_{ss}$		DLS-PBO		OraSLS
	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	min
100_40_4	**210**	**210**	**210**	**210**	**210**	210.1	**210**	**210**	**210**	**210**	**210**	**210**	**210**	**210**	**210**	**210**	**210**
150_60_4	**602**	639.4	**602**	602.2	**602**	**602**	**602**	612.85	**602**	**602**	**602**	**602**	**602**	**602**	**602**	**602**	**602**
200_80_4	**715**	800.45	**715**	715.1	**715**	715.3	**715**	719.45	**715**	716.65	**715**	**715**	**715**	**715**	**715**	715.1	804
250_100_4	**1305**	1516.3	**1305**	**1305**	**1305**	**1305**	**1305**	1401.2	**1305**	1330.05	**1305**	**1305**	**1305**	**1305**	**1305**	**1305**	1553
300_120_4	**1257**	1293.05	**1257**	1257.05	**1257**	1257.05	**1257**	1330.25	**1257**	1373	**1257**	**1257**	**1257**	**1257**	**1257**	1257.35	2572
350_140_4	**1737**	1854.55	**1737**	1744.05	**1737**	1745.84	**1737**	1957.4	**1737**	1997.95	**1737**	**1737**	**1737**	1767.8	**1737**	1773.85	3057
400_160_4	**2240**	2406.1	**2240**	2240.5	**2240**	**2240**	**2240**	2509.55	2241	2598.85	**2240**	**2240**	N/A	N/A	**2240**	2240.1	2502
450_180_4	**1869**	2994.8	**1869**	1889.25	**1869**	**1869**	**1869**	2780.7	1878	2598.7	**1869**	1870.36	N/A	N/A	**1869**	1901	3651
500_200_4	**2577**	3785.2	**2577**	2616.2	**2577**	2624.2	**2577**	3676.8	2674	3637.95	**2577**	**2577**	N/A	N/A	**2577**	2666.5	3736
100_40_6	**140**	**140**	**140**	**140**	**140**	**140**	**140**	**140**	**140**	**140**	**140**	**140**	**140**	**140**	**140**	**140**	**140**
150_60_6	**402**	421.05	**402**	402.05	**402**	**402**	**402**	**402**	**402**	**402**	**402**	**402**	**402**	**402**	**402**	402.1	**402**
200_80_6	**477**	527	**477**	479.55	**477**	477.7	**477**	480	**477**	477.05	**477**	**477**	**477**	**477**	**477**	477.3	828
250_100_6	**870**	907.4	**870**	870.5	**870**	870.8	**870**	893.35	**870**	870.1	**870**	**870**	**870**	**870**	**870**	870.2	957
300_120_6	**839**	932.65	**839**	839.3	**839**	839.85	**839**	866.55	**839**	862.15	**839**	**839**	**839**	**839**	**839**	839.4	1252
350_140_6	**1158**	1256.25	**1158**	1158.85	**1158**	1159.15	**1158**	1267.7	**1158**	1288.25	**1158**	**1158**	**1158**	1160.8	**1158**	1169.95	1914
400_160_6	**1493**	1622.65	**1493**	1494.25	**1493**	1493.15	**1493**	1671.8	**1493**	1656.5	**1493**	**1493**	**1493**	1496.3	**1493**	1493.3	2019
450_180_6	**1246**	1499.45	**1246**	1247.8	**1246**	1295.75	**1246**	1588.45	1265	1641.6	**1246**	**1246.05**	N/A	N/A	**1246**	1275.6	1977
500_200_6	**1718**	1887.6	**1718**	1727.65	**1718**	1823.45	**1718**	1984.05	**1718**	1927	**1718**	**1718**	N/A	N/A	**1718**	1746.84	N/A

As shown in Figure 3, we conduct pairwise comparisons between *NuPBO-DeepOpt+* and twelve representative algorithms on five standard benchmarks, i.e., PB2016, PB2024, MIPLIB, CRAFT, and KNAP. The x-axis lists the five benchmarks, and the y-axis indicates the number of better solutions obtained. It can be seen that *NuPBO-DeepOpt+* is still outperformed by *Gurobi* on all five benchmarks. However, it outperforms *PBO-IHS* and *SCIP* on four of the benchmarks, and outperforms each of the other nine algorithms on all five benchmarks.

To verify the effectiveness of the proposed strategies, we compare *NuPBO-DeepOpt+* with six alternative versions: 1) *nosmooth* use the primary scoring function that does not incorporate the smoothing function; 2) *nohhscore* utilizes the *age* strategy instead of *hhscore*; 3) *noweight* uses the weighting scheme in *LS-PBO* rather than the modified scheme; 4) *nodeepopt* does not use the DeepOpt method; 5) *nomab* replaces the MAB model with a random method; 6) *nodiversityflip* uses the variable flipping rule in *LS-PBO* instead of the *DiversityFlip*

Table 5. Experiment results on SAP.

Instance	NuPBO-DeepOpt+		LS-PBO		DeciLS-PBO		DeepOpt-PBO-v1		DeepOpt-PBO-v2		NuPBO		$NuPBO_{ss}$		DLS-PBO		OraSLS
	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	min
100	**579**	579.75	580	583	**579**	583.4	**579**	**579.7**	**579**	579.8	**579**	580.2	**579**	579.85	580	583.15	N/A
110	**618**	619.45	619	626.05	620	624.75	**618**	**618.75**	**618**	618.85	619	620.65	**618**	620.25	**618**	619.05	N/A
120	**674**	**676.45**	679	685.6	678	685.45	675	677.15	675	677.8	**674**	679	**674**	678.3	**674**	676.75	N/A
130	**733**	**735.4**	738	744.6	737	744.6	**733**	736.9	734	738.4	735	738.2	734	736.9	736	739.85	N/A
140	**750**	752	757	763.65	752	763.75	**750**	**751.9**	**750**	753.75	751	754.55	752	753.95	753	755.15	N/A
150	**802**	**802.35**	821	828.45	818	827.05	803	808.4	803	810.3	803	806.75	**802**	805.5	**802**	804.1	N/A
160	**845**	853.15	867	872.75	862	872.8	849	854.85	849	855.75	856	859.2	848	855.55	854	857.85	N/A
170	**875**	882.55	897	907.5	896	904.95	880	884.4	880	885.55	882	890.1	881	888.05	884	888.7	N/A
180	**938**	943.25	971	977.4	964	975.1	939	948.45	941	951.7	948	954.55	946	951.75	944	952.5	N/A
190	**961**	968.95	996	1005.25	995	1002.7	965	973.2	970	977	974	982.2	970	976.05	971	976.05	N/A
200	**1022**	1032.8	1067	1073.4	1065	1074.6	1029	1040	1029	1042.55	1045	1048.9	1040	1045.45	1034	1043.35	N/A
210	**1060**	1070.35	1094	1112.75	1095	1111.55	1067	1074.7	1068	1077.9	1081	1086	1075	1081.3	1076	1079.8	N/A
220	**1103**	1111.95	1151	1163.1	1148	1162.65	1114	1127.3	1114	1129.6	1116	1126.9	1113	1122.75	1117	1121.1	N/A
230	**1136**	1153.25	1195	1205.75	1192	1204.5	1151	1166.85	1162	1169.7	1167	1170.7	1159	1164.4	1157	1160.75	N/A
240	**1171**	1179.45	1219	1234.55	1219	1233.8	1179	1192.6	1183	1195.5	1193	1203	1187	1193.4	1179	1189.85	N/A
250	**1217**	1227.65	1274	1290.55	1273	1288.35	1235	1243.1	1237	1245.5	1242	1249.45	1238	1242.35	1229	1236.9	N/A
260	**1255**	1268.45	1318	1334.9	1304	1333.55	1275	1286.15	1277	1287.45	1270	1290.9	1277	1284	1273	1278.45	N/A
270	**1325**	1333.65	1392	1403.55	1389	1402.35	1344	1353.1	1348	1356.1	1354	1361.3	1351	1354.5	1344	1348.15	N/A
280	**1332**	1355.05	1407	1424.3	1401	1423.7	1348	1370.75	1354	1375.35	1373	1380.5	1368	1373.45	1360	1367.5	N/A
290	**1386**	1391.5	1448	1471.15	1453	1468.3	1403	1416.35	1405	1419.5	1414	1421.2	1406	1414.65	1395	1407.6	N/A
300	**1456**	1469.85	1538	1548.4	1529	1546.7	1471	1491.5	1480	1496.1	1485	1500.2	1475	1489.1	1477	1483.8	N/A

Table 6. Summary results of comparing our proposed algorithm to its competitors on all the benchmarks. #inst denotes the number of instances in each benchmark.

Benchmark		MWCB	WSNO	SAP	PB2016	PB2024	MIPLIB	CRAFT	KNAP	Total
#inst		24	18	21	1600	478	267	955	783	4146
RoundingSat	#win	0	4	0	1097	275	75	812	392	2655
PBO-IHS	#win	0	4	0	1065	246	86	846	778	3025
SCIP	#win	0	4	0	1180	246	121	832	**783**	3166
Gurobi	#win	0	4	0	**1343**	**382**	**204**	**916**	**783**	**3632**
LS-PBO	#win	0	**18**	0	980	282	109	839	647	2875
DeciLS-PBO	#win	0	**18**	1	967	275	113	832	649	2855
DeepOpt-PBO-v1	#win	5	**18**	4	1054	283	119	857	693	3033
DeepOpt-PBO-v2	#win	2	14	3	1124	293	123	888	692	3139
NuPBO	#win	0	**18**	2	1208	329	134	888	569	3148
$NuPBO_{ss}$	#win	0	13	4	1228	315	139	890	535	3124
DLS-PBO	#win	0	**18**	3	1154	307	129	887	676	3174
OraSLS	#win	0	4	0	1084	229	104	802	444	2667
NuPBO-DeepOpt+	#win	**19**	**18**	**21**	1214	337	158	908	707	3382

method. The experimental results are presented in Table 7. Overall, across the 4146 instances, *NuPBO-DeepOpt+* demonstrates a clear advantage over all modified versions in terms of the total number of #better. In particular, the modifications involving *nodeepopt* and *nodiversityflip* have a substantial impact, with *NuPBO-DeepOpt+* achieving significantly better performance in terms of the #better metric across most benchmarks, resulting in favorable better/worse counts of 535/229 and 594/281, respectively. However, compared to the *nohhscore* version, although *NuPBO-DeepOpt+* performs better across all benchmarks, the improvement in each benchmark is not substantial. The new weighting scheme performs particularly well on MWCB, SAP, PB2016, PB2024, MIPLIB, and CRAFT. However, it has no effect on WSNO and worsens 69 instances on KNAP. The primary reason is the unique structure of the KNAP instances, which include only one PB constraint. Since the new weighting scheme does not restrict the weight of the objective constraint, it frequently increases the weight of the objective constraint because the PB constraint is easily satisfied. Additionally, the smooth strategy and the MAB-PBO model have a negative effect on MWCB. The possible reason is that each objective constraint in MWCB involves

Table 7. Comparative results for *NuPBO-DeepOpt+* and its modified versions with different strategies on all seven benchmarks. #better and #worse denote the number of instances where *NuPBO-DeepOpt+* obtains better and worse solution values, respectively.

Benchmark	#inst	vs. nosmooth		vs. nohhscore		vs. noweight		vs. nodeepopt		vs. nomab		vs. nodiversityflip	
		#better	#worse	#better	#worse	#better	#worse	#better	#worse	#better	#worse	#better	#worse
MWCB	24	0	**24**	**11**	8	**17**	7	**24**	0	0	**24**	**24**	0
WSNO	18	0	0	**1**	0	0	0	**1**	0	0	**1**	0	0
SAP	21	**10**	2	**11**	5	**18**	0	**18**	0	**10**	4	**20**	0
PB2016	1600	**268**	100	**407**	388	**308**	104	**274**	119	**178**	117	**301**	155
PB2024	478	**123**	21	**64**	60	**163**	19	**87**	34	**73**	55	**72**	69
MIPLIB	267	**83**	28	**45**	44	**115**	11	**62**	25	**60**	47	**63**	37
CRAFT	955	**18**	1	**40**	36	**83**	24	**40**	26	**29**	22	**23**	0
KNAP	783	**20**	18	**27**	24	32	**69**	**29**	25	**27**	20	**91**	20
Total	4146	**522**	194	**606**	565	**736**	234	**535**	229	**377**	290	**594**	281

two variables, and these variables appear in multiple objective constraints, which is different from all other benchmarks. Therefore, the smooth strategy and the MAB-PBO model are not suitable and need adjustment for this unique structure.

Additionally, we present the run time distribution to highlight the superiority of our *NuPBO-DeepOpt+* algorithm. Figures 4 and 5 report the run time of *NuPBO-DeepOpt+* and a competitor when achieving the same optimal solution, underscoring the effectiveness of *NuPBO-DeepOpt+*. Figure 4 presents the run time distributions of *NuPBO-DeepOpt+* compared to six algorithms, namely *LS-PBO*, *DeciLS-PBO*, *DeepOpt-PBO-v1*, *DeepOpt-PBO-v2*, *NuPBO*, and $NuPBO_{ss}$, while Figure 5 compares it against *DLS-PBO*, *OraSLS*, *RoundingSat*, *PBO-IHS*, *SCIP*, and *Gurobi*.

9.3 Parameter Sensitivity Analysis

To evaluate the parameter sensitivity, we conduct experiments by varying the values of ten parameters used in the algorithm to demonstrate the performance differences under different parameter configurations. From each of the eight benchmarks, we select one challenging instance, eight instances in total as test cases, with each parameter taking five different values. As in previous experimental settings, the running time is limited to 3600 seconds. For the WMCB, WSNO, and SAP instances, the algorithm is run 20 times (with seeds ranging from 1 to 20), and the best solution obtained is considered the final solution for that parameter configuration. For the PB2016, PB2024, MIPLIB, CRAFT, and KNAP instances, each is run once.

The experimental results are presented in Figures 6 and 7. Figure 6 includes the six parameters applied in the DeepOpt module, including *MinStep*, δ, *MinHard*, γ, *MaxHard*, and L_{opt}. Figure 7 includes the four parameters applied in the MAB model and SSE modules, including λ, d, *NumSample*, and β. Each figure compares the solutions for eight instances under different values of a parameter. The x-axis represents the five different values for each parameter, while the y-axis shows the normalized optimality gap (*nog*) for the instances. Assuming that the solution obtained under a given parameter configuration is denoted as sol_i, and the optimal solution is denoted as sol_{best}, the normalized optimality gap is calculated as $nog = \frac{sol_i - sol_{best}}{sol_{best}}$. The *nog* measures the variation in the solutions obtained across different parameter configurations, highlighting the sensitivity of various instances to parameter changes.

Based on Figures 6 and 7, it can be observed that the CRAFT and SAP instances exhibit significant fluctuations across the ten parameters, indicating that the performance of these instances is highly sensitive to parameter variations. The MWCB instance shows considerable fluctuations in the seven parameters, including *MinStep*,

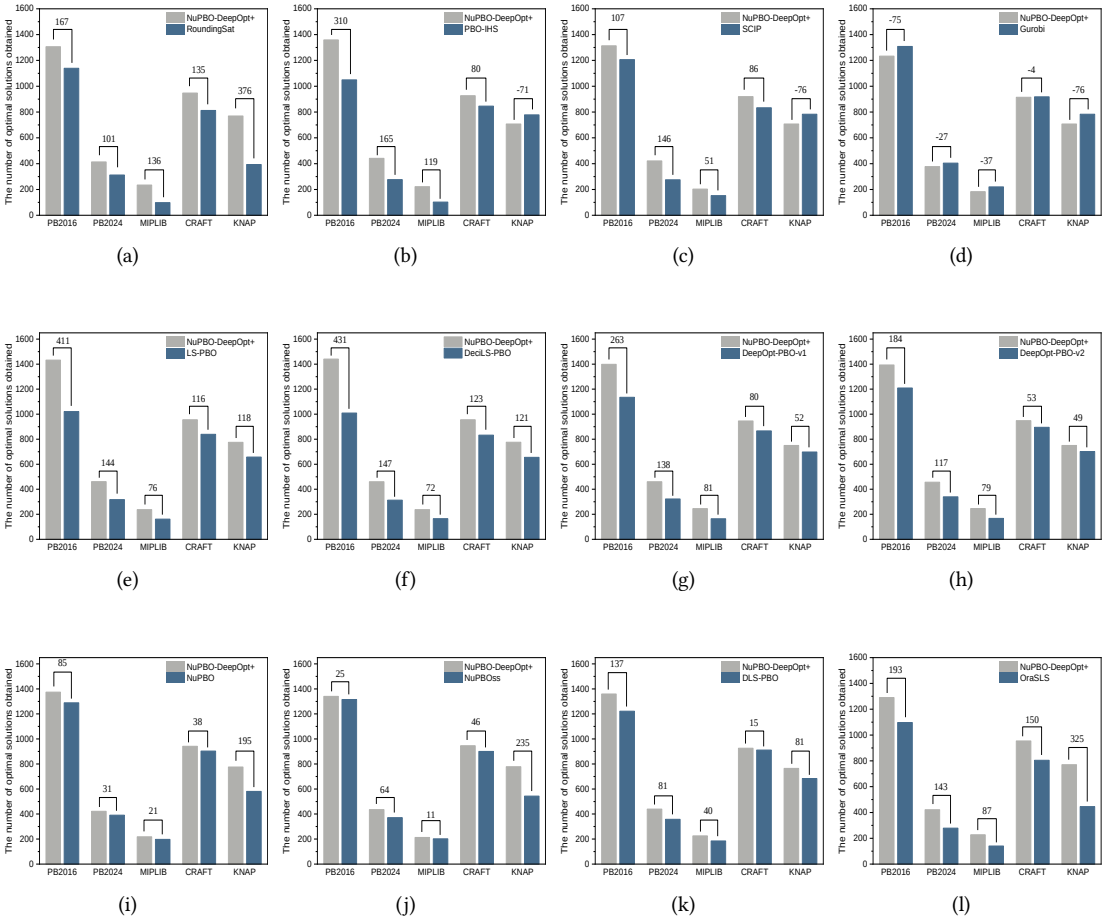


Fig. 3. Pairwise comparison between *NuPBO-DeepOpt+* and twelve comparative algorithms on five standard benchmarks in terms of the number of better solutions obtained: (a) *RoundingSat*, (b) *PBO-IHS*, (c) *SCIP*, (d) *Gurobi*, (e) *LS-PBO*, (f) *DeciLS-PBO*, (g) *DeepOpt-PBO-v1*, (h) *DeepOpt-PBO-v2*, (i) *NuPBO*, (j) *NuPBO$_{ss}$*, (k) *DLS-PBO*, and (l) *OraSLS*.

MinHard, *MaxHard*, L_{opt}, d, *NumSample*, and β, while the effects of δ, γ, and λ are relatively minor. PB2016 exhibits changes across all ten parameters, but the variations are not substantial. MIPLIB shows some variation in γ, *MaxHard*, λ, *NumSample*, and β, while the remaining five parameters remain unchanged. The WSNO instance shows a degradation in performance when the *MinHard* parameter is set to 25, with no significant changes observed in other parameters. Finally, the PB2024 and KNAP instances show no change in results across all parameters.

9.4 Further Experiments

9.4.1. Integrating Presolving.

It can be observed that our proposed algorithm outperforms most of the competing solvers except for *Gurobi*, especially in the standard four benchmarks including PB2016, PB2024, MIPLIB, and KNAP. An important factor

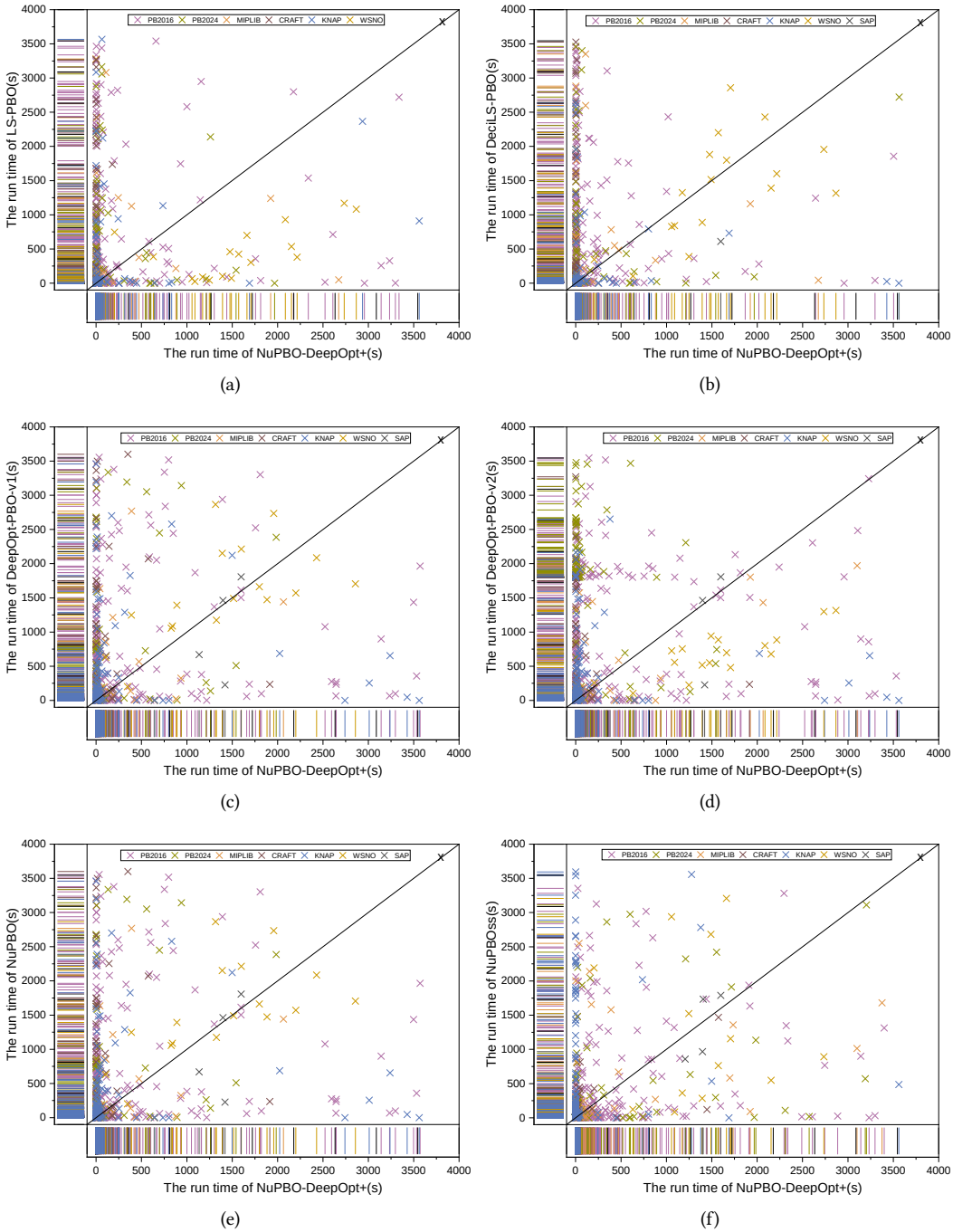


Fig. 4. Comparison of run time distributions between *NuPBO-DeepOpt+* and six local search algorithms: (a) *LS-PBO*, (b) *DeciLS-PBO*, (c) *DeepOpt-PBO-v1*, (d) *DeepOpt-PBO-v2*, (e) *NuPBO*, and (f) $NuPBO_{ss}$.

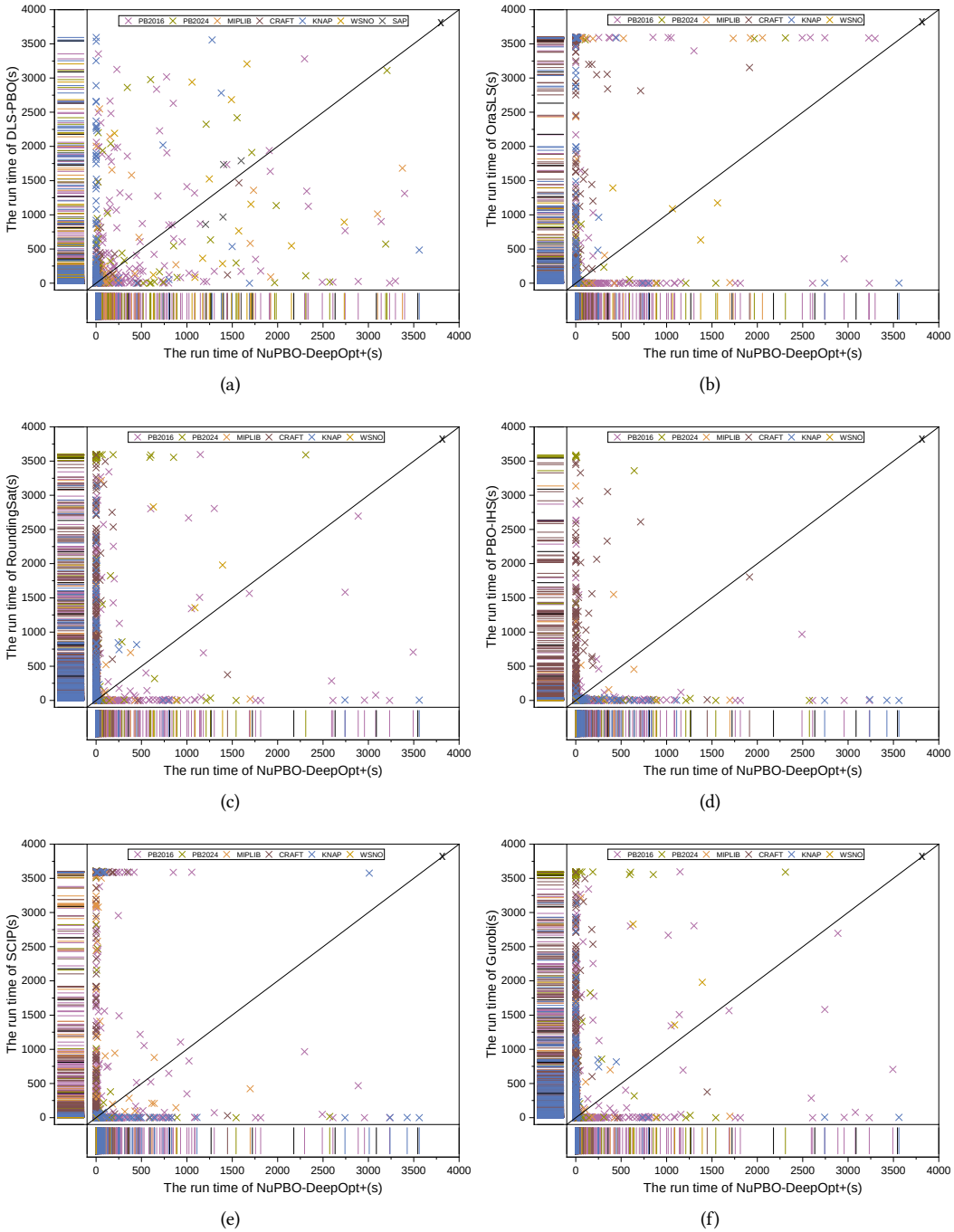


Fig. 5. Comparison of run time distributions between *NuPBO-DeepOpt+* algorithm with two local search algorithms, two exact PBO solvers and two MIP solvers: (a) *DLS-PBO*, (b) *OraSLS*, (c) *RoundingSat*, (d) *PBO-IHS*, (e) *SCIP*, and (f) *Gurobi*.

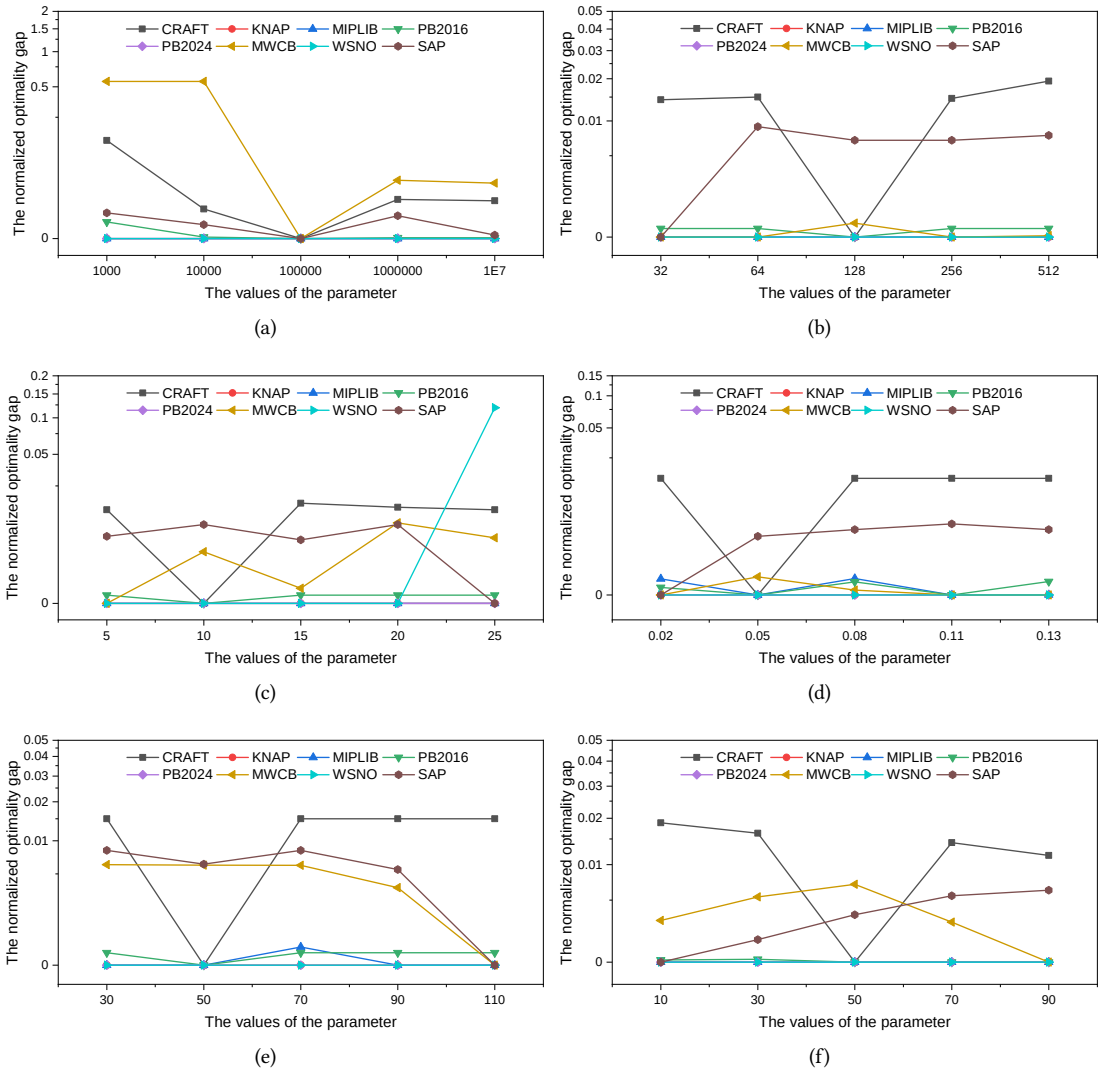


Fig. 6. Performance comparison of eight instances under different parameter configurations: (a) *MinStep*, (b) δ, (c) *MinHard*, (d) γ, (e) *MaxHard*, and (f) L_{opt}.

is that presolving is a very crucial part of MIP performance, whereas current local search solvers for PBO lack preprocessing [14]. A natural approach is to integrate the PAPILO preprocessor [6] with local search solvers for PBO to observe potential changes in performance. Since PBO is a special case of MIP problems (equivalent to 0-1 MIP problem), some operations of the PAPILO preprocessor can be modified to enable it to handle PB constraints. Subsequently, the reduced instances are solved using the algorithm we proposed.

[6]PaPILO repository: https://github.com/lgottwald/PaPILO

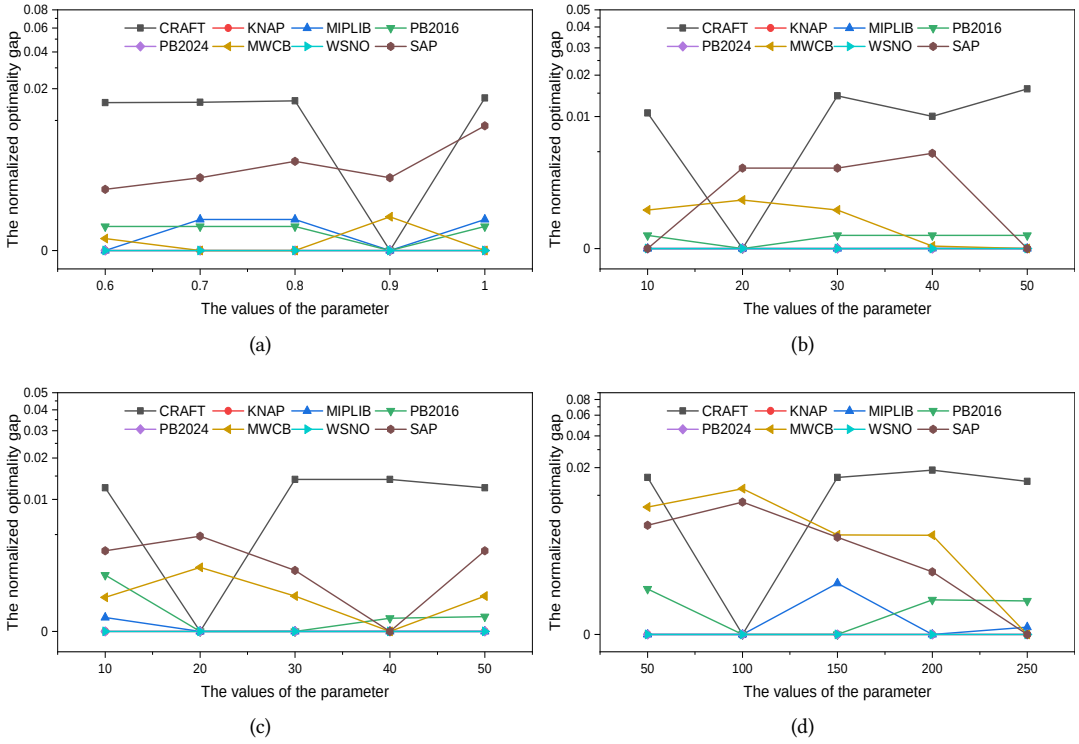


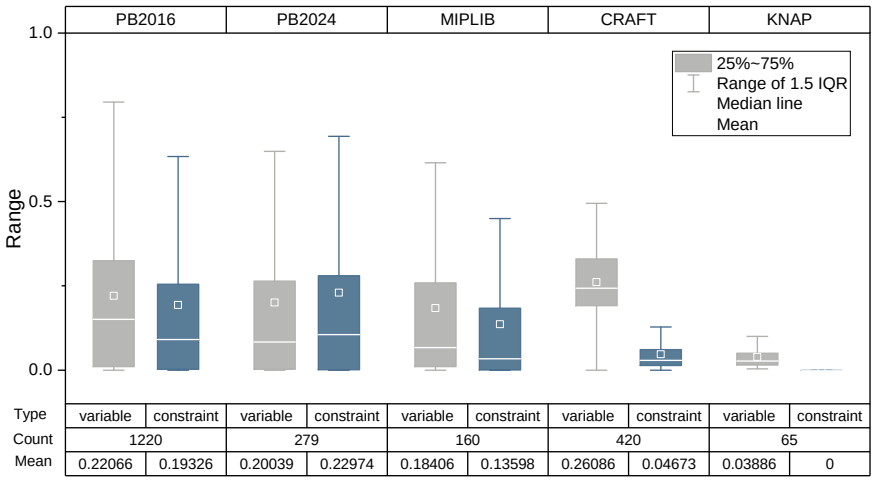
Fig. 7. Performance comparison of eight instances under different parameter configurations: (a) λ, (b) d, (c) *NumSample*, and (d) β.

Table 8. The number of instances that can be presolved and proven to be unsatisfiable in all eight benchmarks.

Benchmarks	MWCB #inst	SAP #inst	WSNO #inst	PB2016 #inst	PB2024 #inst	MIPLIB #inst	CRAFT #inst	KNAP #inst
presolved	0	0	0	1220	279	160	420	65
unsatisfiable	0	0	0	42	6	1	0	0

Table 8 shows the number of instances that can be presolved in eight benchmarks. Here, *presolved* represents the number of instances that can be reduced, while *unsatisfiable* indicates the number of instances that are proven to be unsatisfiable through presolving. It can be observed that the three application benchmarks, MWCB, SAP, and WSNO cannot be presolved. For the five benchmarks that can be reduced (i.e., PB2016, PB2024, MIPLIB, CRAFT and KNAP), we use R_v and R_c to respectively represent the ratio of reduction in variables and constraints relative to the original instance after applying a preprocessor.

- Variable reduction ratio (R_v): The reduction ratio can be calculated by the difference between the original number of variables and the reduced number of variables, divided by the original number of variables, as



	PB2016		PB2024		MIPLIB		CRAFT		KNAP	
Type	variable	constraint	variable	constraint	variable	constraint	variable	constraint	variable	constraint
Count	1220		279		160		420		65	
Mean	0.22066	0.19326	0.20039	0.22974	0.18406	0.13598	0.26086	0.04673	0.03886	0

Fig. 8. The range of reduction ratio in variables and constraints after preprocessing on the PB2016, PB2024, MIPLIB, CRAFT and KNAP benchmarks.

follows:

$$R_v = \frac{N_{org\_v} - N_{red\_v}}{N_{org\_v}}$$

where N_{org_v} and N_{red_v} represent the number of variables in an instance before and after applying a preprocessor.

- Constraint reduction ratio (R_c): The reduction ratio can be calculated by the difference between the original number of constraints and the reduced number of constraints, divided by the original number of constraints, as follows:

$$R_c = \frac{N_{org\_c} - N_{red\_c}}{N_{org\_c}}$$

where N_{org_c} and N_{red_c} represent the number of constraints in an instance before and after applying a preprocessor.

Figure 8 displays the distribution of variable and constraint reduction ratios for five benchmarks that can be reduced. The interquartile range (IQR) is a statistical measure used to describe the distribution of data points, measuring the range of the middle 50% of the data. In Figure 8, it is reflected as the size of the box. It can be clearly observed that the median (represented by a horizontal line) and mean (represented by a square) reduction ratios for the five benchmarks. Specifically, PB2016, PB2024 and MIPLIB have high reduction ratios for both variables and constraints, while CRAFT has a high reduction ratio for variables but low reduction for constraints. In contrast, KNAP shows a low reduction ratio for variables and no reduction for constraints.

We conduct experiments on five standard benchmarks that are amenable to presolving. Specifically, we run our developed *NuPBO-DeepOpt+* algorithm on these five benchmarks after presolving, using the same random seed and cutoff time. Table 9 displays a comparison of experimental results with and without presolving. We bold the better results between #better and #worse, as well as the faster run time. The algorithm with presolving performs 155, 53, and 47 better, and 69, 12, and 27 worse on PB2016, PB2024 and MIPLIB benchmarks respectively. However, since the instances in the CRAFT and KNAP benchmarks that can be preprocessed are easily solved, the performance differences on these two datasets are not significant.

Table 9. Comparison of the number of better solutions obtained and the runtime with and without the use of presolving.

Benchmarks	#inst	presolve vs. nopresolve			presolve vs. nopresolve	
		#better	#worse	#same	runtime(s)	runtime(s)
PB2016	1220	**155**	69	996	**120176.38**	158401.21
PB2024	279	**53**	12	214	27426.64	**23891.17**
MIPLIB	160	**47**	27	86	**36404**	41295.87
CRAFT	420	0	0	420	**13.42**	19.38
KNAP	65	**1**	0	64	**201.03**	233.14
Total	2144	**256**	108	1780	**184221.47**	223840.77

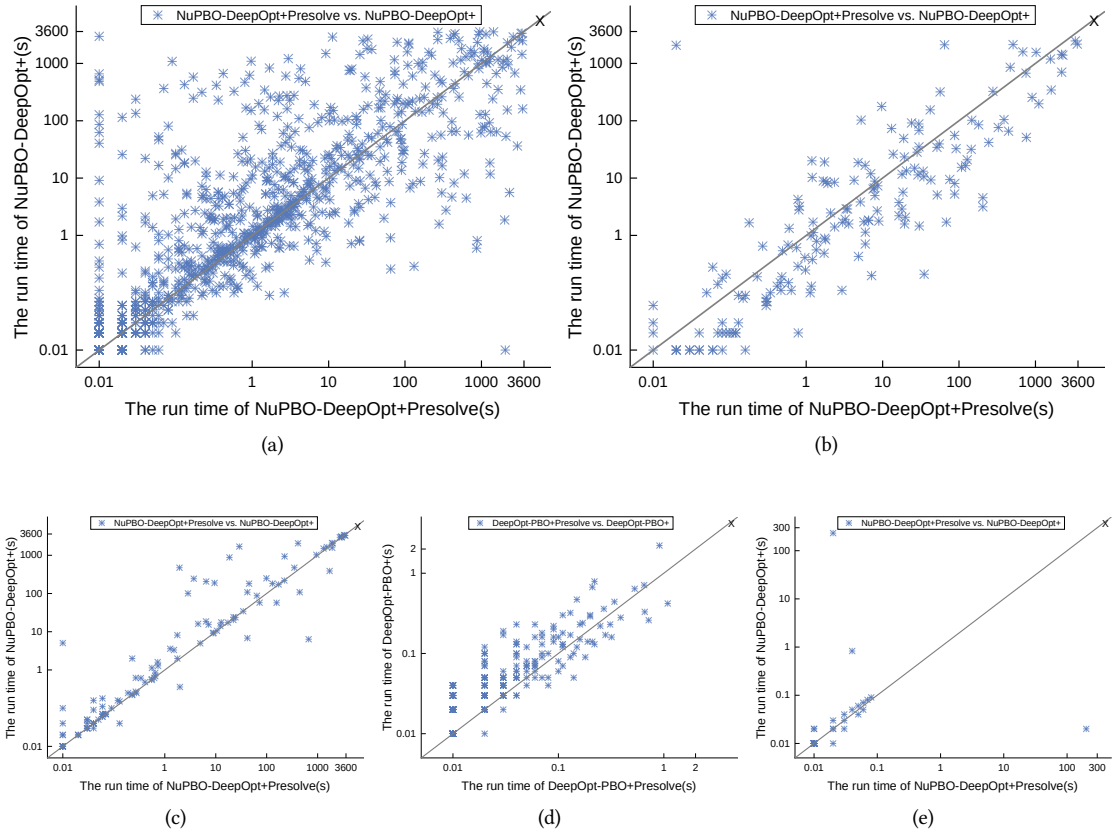


Fig. 9. Comparison of run time distributions between *NuPBO-DeepOpt+* algorithm with and without presolving on five benchmarks: (a) PB2016, (b) PB2024, (c) MIPLIB, (d) CRAFT, and (e) KNAP.

Additionally, we compare the run time between using a preprocessor and not using one. #*same* denotes the number of instances where both algorithms reached the same solution, and we record run time only for these instances. The results indicate that, in most benchmarks, including PB2016, MIPLIB, CRAFT, and KNAP, the

use of a preprocessor accelerates the algorithm's performance. However, in PB2024, the use of the preprocessor actually reduces the algorithm's efficiency. Figure 9 displays the comparison of run time distributions between the two algorithms across five benchmarks. The points labeled on the line $y = x$ represent instances where both algorithms achieve same solutions at the same time. Points above this line indicate that the *NuPBO-DeepOpt+* algorithm takes longer to solve an instance compared to the *NuPBO-DeepOpt+Presolve* algorithm, while points below it show the reverse. Overall, *NuPBO-DeepOpt+Presolve* algorithm demonstrates higher efficiency across most benchmarks.

9.4.2. Comparisons with Gurobi.

To evaluate the performance of the combined preprocessing, we conduct additional experiments to compare it with *Gurobi*. The primary reason for the comparison is that without the integration of the preprocessor, *NuPBO-DeepOpt+* underperforms compared to *Gurobi* across five standard benchmarks. This allows us to observe how the algorithm's performance improves with the inclusion of a preprocessor. Specifically, due to the influence of random seeds on the search direction of local search algorithms, we test our proposed *NuPBO-DeepOpt+* algorithm combined with presolving on standard benchmarks using 20 random seeds, ranging from 1 to 20. We conduct comparisons among *NuPBO-DeepOpt+*, *NuPBO-DeepOpt+* with presolving run using a single random seed denoted as *NuPBO-DeepOpt+Pre*$_{seed1}$, *NuPBO-DeepOpt+* with presolving run using 20 random seeds denoted as *NuPBO-DeepOpt+Pre*$_{seed20}$, and *Gurobi*.

According to the experimental results presented in Table 10, *NuPBO-DeepOpt+Pre*$_{seed20}$ slightly outperforms *Gurobi* in terms of the total number of best solutions obtained among all algorithms. Specifically, *NuPBO-DeepOpt+Pre*$_{seed20}$ surpasses *Gurobi* in the number of optimal solutions obtained on five benchmarks, including MWCB, WSNO, SAP, PB2016, and CRAFT. It demonstrates the algorithm's high performance in both real-world application scenarios and specialized competition benchmark of PB2016. On the PB2024, *NuPBO-DeepOpt+Pre*$_{seed20}$ and *Gurobi* deliver identical results. For the CRAFT benchmark, their performance differs only marginally. In contrast, *Gurobi* continues to perform exceptionally well in MIPLIB and KNAP, one possible reason is that MIPLIB is a dataset specifically dedicated to MIP problem. Moreover, the KNAP instances feature a single-constraint structure that our algorithm has not yet specifically optimized for, resulting in somewhat limited adaptability in that scenario.

Table 10. Summary results of comparing algorithm *NuPBO-DeepOpt+*, *NuPBO-DeepOpt+* with presolving run using a single random seed, *NuPBO-DeepOpt+* with presolving run using 20 random seeds, and *Gurobi* on all the benchmarks.

Benchmarks	#inst	NuPBO-DeepOpt+ #win	NuPBO-DeepOpt+Pre$_{seed1}$ #win	NuPBO-DeepOpt+Pre$_{seed20}$ #win	Gurobi #win
MWCB	24	0	0	**24**	0
SAP	21	2	2	**21**	0
WSNO	18	3	3	**18**	4
PB2016	1600	1233	1273	**1402**	1381
PB2024	478	311	321	**386**	**386**
MIPLIB	267	161	167	188	**215**
CRAFT	955	907	907	**921**	918
KNAP	783	707	707	738	**783**
Total	4146	3324	3380	**3698**	3687

10 Conclusion and Future Work

This work is centered on the critical factors that influence local search algorithms and proposes four techniques to improve the performance of local search algorithms. The first idea involves designing a new primary scoring function to balance the *viol* values of different constraints, and introduces a two-level selection strategy to address the issue about tie-breaking in the primary scoring function. The second idea includes new weight initialization rules to quickly find a feasible solution, and stricter weight update rules to balance the weights of PB and objective constraints. The third idea proposes a perturbation strategy to deeply probe some search regions and then quickly converge to a new solution. Finally, we construct a solution space exploration mechanism that includes using the multi-armed bandit method and sampling strategy to select constraints, along with a diversity flip strategy to flip one or more variables. Based on the above techniques, we develop a new local search algorithm *NuPBO-DeepOpt+*. Additionally, we conduct experiments combining *NuPBO-DeepOpt+* with a preprocessor to observe changes in algorithm performance. Experiments show that the proposed algorithm significantly outperforms most of the current state-of-the-art solvers and can rival the performance of *Gurobi*.

In future work, we will explore a hybrid framework that combines local search algorithms with complete solvers, aiming to fully leverage the strengths of both approaches to tackle more complex optimization problems. Secondly, we plan to investigate MIP solvers, drawing on advanced techniques from the MIP domain, and integrate these techniques into our algorithm to further enhance solution efficiency and performance.

Acknowledgments

Yi Chu and Yiyuan Wang are corresponding authors. We would like to thank the anonymous referees for their helpful comments. This work is supported by NSFC under Grant No. 62302492 and 61806050, National Cryptologic Science Fund of China 2025NCSF02046, the Fundamental Research Funds for the Central Universities 2412023YQ003, Jilin Science and Technology Department 20240602005RC.

References

[1] C. Ansotegui, F. Bacchus, M. Järvisalo, and R. Martins. 2017. *MaxSAT Evaluation 2017: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki.

[2] S. Arya and Y. Yang. 2020. Randomized allocation with nonparametric estimation for contextual multi-armed bandits with delayed rewards. *Statistics & Probability Letters*, 164, 108818.

[3] J. Berg, E. Oikarinen, M. Järvisalo, and K. Puolamäki. 2017. Minimum-width confidence bands via constraint optimization. In *CP*. Vol. 10416, 443–459.

[4] K. Bestuzheva et al. 2021. The scip optimization suite 8.0. *arXiv preprint arXiv:2112.08872*, 1, 1–114.

[5] S. Cai, J. Lin, and C. Luo. 2017. Finding a small vertex cover in massive sparse graphs: construct, local search, and preprocess. *Journal of Artificial Intelligence Research*, 59, 463–494.

[6] S. Cai and K. Su. 2013. Comprehensive score: towards efficient local search for SAT with long clauses. In *IJCAI*, 489–495.

[7] B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki. 1997. Local search algorithms for partial maxsat. *AAAI/IAAI*, 263268, 9.

[8] J. Chen, S. Cai, Y. Wang, W. Xu, J. Ji, and M. Yin. 2023. Improved local search for the minimum weight dominating set problem in massive graphs by using a deep optimization mechanism. *Artificial Intelligence*, 314, 103819.

[9] X. Chen, Z. Lei, and P. Lu. 2024. Deep cooperation of local search and unit propagation techniques. In *CP*. Vol. 307, 6:1–6:16.

[10] Z. Chen, P. Lin, H. Hu, and S. Cai. 2024. Parls-pbo: A parallel local search solver for pseudo boolean optimization. In *CP*. Vol. 307, 5:1–5:17.

[11] Y. Chu, S. Cai, C. Luo, Z. Lei, and C. Peng. 2023. Towards more efficient local search for pseudo-boolean optimization. In *CP*. Vol. 280, 12:1–12:18.

[12] S. A. Cook. 1971. The complexity of theorem-proving procedures. In *STOC*, 151–158.

[13] J. Devriendt, A. Gleixner, and J. Nordström. 2021. Learn to relax: integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search. *Constraints*, 26, 1, 26–55.

[14] J. Devriendt, S. Gocht, E. Demirović, J. Nordström, and P. J. Stuckey. 2021. Cutting to the core of pseudo-Boolean optimization: combining core-guided search with cutting planes reasoning. In *AAAI*. Vol. 35, 3750–3758.

[15] N. Eén and N. Sörensson. 2006. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2, 1-4, 1–26.
[16] J. Elffers and J. Nordström. 2018. Divide and conquer: towards faster pseudo-Boolean solving. In *IJCAI*, 1291–1299.
[17] J. Franco and J. Martin. 2009. A history of satisfiability. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications. Vol. 185, 3–74.
[18] L. Gurobi Optimization. 2021. *Gurobi optimizer reference manual*.
[19] Y. Hu, Y. Yu, and J. Liao. 2019. Cascaded algorithm-selection and hyper-parameter optimization with extreme-region upper confidence bound bandit. In *IJCAI*, 2528–2534.
[20] M. Iser, J. Berg, and M. Järvisalo. 2023. Oracle-based local search for pseudo-boolean optimization. In *ECAI*, 1124–1131.
[21] L. Jiang, D. Ouyang, Q. Zhang, and L. Zhang. 2024. Decils-pbo: an effective local search method for pseudo-boolean optimization. *Frontiers of Computer Science*, 18, 2, 182326.
[22] H. A. Kautz and B. Selman. 1992. Planning as satisfiability. In *ECAI*, 359–363.
[23] G. Kovásznai, B. Erdélyi, and C. Biró. 2018. Investigations of graph properties in terms of wireless sensor network optimization. In *Future IoT*, 1–8.
[24] G. Kovásznai, K. Gajdár, and L. Kovács. 2019. Portfolio SAT and SMT solving of cardinality constraints in sensor network optimization. In *SYNASC*, 85–91.
[25] D. Le Berre and A. Parrain. 2010. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7, 2-3, 59–64.
[26] Z. Lei, S. Cai, C. Luo, and H. H. Hoos. 2021. Efficient local search for pseudo Boolean optimization. In *SAT*. Vol. 12831, 332–348.
[27] M. López-Ibáñez, J. Dubois-Lacoste, L. P. Cáceres, M. Birattari, and T. Stützle. 2016. The irace package: iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3, 43–58.
[28] R. Martins, V. Manquinho, and I. Lynce. 2014. Open-WBO: a modular maxsat solver. In *SAT*. Vol. 8561, 438–445.
[29] D. Pisinger. 2005. Where are the hard knapsack problems? *Computers & Operations Research*, 32, 9, 2271–2284.
[30] O. Roussel and V. Manquinho. 2021. Pseudo-Boolean and cardinality constraints. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications. Vol. 336, 1087–1129.
[31] M. Sakai and H. Nabeshima. 2015. Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. *IEICE Transactions on Information and Systems*, E98.D, 6, 1121–1127.
[32] P. Shaw. 1998. Using constraint programming and local search methods to solve vehicle routing problems. In *CP*. Vol. 1520, 417–431.
[33] J. P. M. Silva and K. A. Sakallah. 2000. Boolean satisfiability in electronic design automation. In *DAC*, 675–680.
[34] P. Smirnov, J. Berg, and M. Järvisalo. 2022. Improvements to the implicit hitting set approach to pseudo-Boolean optimization. In *SAT*. Vol. 236, 13:1–13:18.
[35] P. Smirnov, J. Berg, and M. Järvisalo. 2021. Pseudo-Boolean optimization by implicit hitting sets. In *CP*. Vol. 210, 51:1–51:20.
[36] Z. Su, Q. Zhang, Z. Lü, C.-M. Li, W. Lin, and F. Ma. 2021. Weighting-based variable neighborhood search for optimal camera placement. In *AAAI*. Vol. 35, 12400–12408.
[37] J. Thornton and A. Sattar. 1998. Dynamic constraint weighting for over-constrained problems. In *PRICAI*, 377–388.
[38] R. Tinós, M. W. Przewozniczek, and D. Whitley. 2022. Iterated local search with perturbation based on variables interaction for pseudo-Boolean optimization. In *GECCO*, 296–304.
[39] M. Vinyals, J. Elffers, J. Giráldez-Cru, S. Gocht, and J. Nordström. 2018. In between resolution and cutting planes: A study of proof systems for pseudo-Boolean SAT solving. In *SAT*. Vol. 10929, 292–310.
[40] Y. Wang, S. Cai, J. Chen, and M. Yin. 2020. Sccwalk: an efficient local search algorithm and its improvements for maximum weight clique problem. *Artificial Intelligence*, 280, 103230.
[41] R. Wille, H. Zhang, and R. Drechsler. 2011. ATPG for reversible circuits using simulation, boolean satisfiability, and pseudo boolean optimization. In *ISVLSI*, 120–125.
[42] Y. Zhang, R. I. Hartley, J. Mashford, and S. Burn. 2011. Superpixels via pseudo-boolean optimization. In *ICCV*, 1387–1394.
[43] J. Zheng, K. He, and J. Zhou. 2023. Farsighted probabilistic sampling: A general strategy for boosting local search maxsat solvers. In *AAAI*, 4132–4139.
[44] J. Zheng, K. He, J. Zhou, Y. Jin, C.-M. Li, and F. Manya. 2022. Bandmaxsat: a local search maxsat solver with multi-armed bandit. In *IJCAI*, 1901–1907.
[45] J. Zheng, K. He, J. Zhou, Y. Jin, C. Li, and F. Manyà. 2025. Integrating multi-armed bandit with local search for maxsat. *Artificial Intelligence*, 338, 104242.
[46] W. Zhou, Y. Zhao, Y. Wang, S. Cai, S. Wang, X. Wang, and M. Yin. 2023. Improving local search for pseudo boolean optimization by fragile scoring function and deep optimization. In *CP*. Vol. 280, 41:1–41:18.

Received 17 June 2024; revised 21 May 2025; accepted 5 June 2025