An Efficient Structured Perceptron for NP-hard Combinatorial Optimization Problems

Bastián Véjar[1], Gaël Aglin[2], Ali İrfan Mahmutoğulları[1], Siegfried Nijssen[2], Pierre Schaus[2], and Tias Guns[1]

[1] Dept. Computer science, KU Leuven, Leuven, Belgium
[2] ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium

Abstract. A fundamental challenge when modeling combinatorial optimization problems is that often multiple sub-objectives need to be weighted against each other, but it is not clear how much weight each sub-objective should be given: consider routing problems with different definitions of distance, or planning problems with different definitions of costs, where the importance of these costs is not known a priori. In recent work, it has been proposed to use machine learning algorithms from the domain of structured output prediction to learn such weights from examples of desirable solutions. However, until now such techniques were only evaluated on relatively simple optimization problems. We propose and evaluate three techniques that make it feasible to apply the structured perceptron on NP-hard optimization problems: 1) using heuristic solving methods during the learning process, 2) solving well-chosen satisfaction problems, 3) caching solutions computed during the learning process and reusing them. Experiments confirm the validity and speed-ups of these techniques, enabling structured output learning on larger combinatorial problems than before.

Keywords: Structured output prediction · Combinatorial optimization · Structured Perceptron

1 Introduction

Combinatorial Optimization (CO) deals with solving optimization problems such as scheduling and planning problems. These problems can be formalized as problems in which we wish to find a solution with the maximum objective function value among a finite number of solutions that satisfy a set of constraints. In many applications, however, multiple objectives can be considered. Notable examples of this appear in engineering [1, 6], economics [13], and logistics [7]. As an example, in a logistics application, two different objectives may be important for a decision-maker: to evaluate the cost and the travel time of a path or a tour.

Since objectives of a Multi-Objective Combinatorial Optimization Problem (MOCOP) are possibly conflicting, a nontrivial problem is how to formalize MOCOP problems using a single objective function. The classical methods either try to enumerate all non-dominated solutions (see, for example, [5]) or use a weighted linearization of the objectives –turning these into sub-objectives– to transform the problem into a single-objective optimization problem, for example, [10]. However, the weights in the latter approach may not be explicitly known to the decision-maker.

The motivation for this paper comes from the scheduling and production process of a steel mill company. The company tries to set the weights of the different sub-objectives of their scheduling process, e.g. the difference in height and width of the products, based on a perception of their importance. This process requires several iterations of tuning and observing the production process with these updated parameters. The company however has a set of historical optimal solutions that a machine learning approach can use.

Structured Output Prediction (SOP) is a family of machine learning techniques used for learning to predict structured objects. The SOP methods are known to be effective if calculating the output under structural constraints for a given input can be done in polynomial time. Although SOP can also be used to learn the objective weights in a MOCOP in theory, the training of some SOP algorithms, such as the Structured Perceptron (SP), requires solving multiple instances of MOCOPs with different weights. In the steel mill company example, the corresponding MOCOP is NP-hard, and finding an optimal solution requires a large amount of computational power and training time. In a scenario like this, the application of the SP is challenging or even impossible in practice.

Motivated by the real-world application example mentioned above, in this paper, we propose a number of techniques that make it feasible to apply the SP to hard MOCOP problems, where the goal is to learn the weights of the sub-objectives. A core idea is that we replace the need for an exact solution of the MOCOP during the learning process with less demanding alternatives in terms of solution time and power. Our main observation is that during the learning process, an optimal solution of a MOCOP can be replaced by any solution that shows that the current choice of weights does not yet favor the desired solution according to the training data. Based on this observation we introduce computational enhancements such as timeouts, a reformulation of the problem based on the better-score condition, and scanning a cache of historical solutions. Finally, we conduct computational experiments to observe the benefit of the proposed methods on multi-objective versions of two NP-hard combinatorial optimization problems: the Knapsack Problem (KP) and the Price Collecting Traveling Salesman Problem (PCTSP).

The contributions of our work are:

- We propose a novel valid update approach for the training of the SP. This approach exploits any solution that for the current choice of weights has a better score than the preferred solution, to avoid having to solve a MOCOP optimally for each instance at each training epoch.
- We also present computational improvements to speed up the training of the SP. We use a heuristic approach, adding the score criterion as a constraint and solving a SAT, and solution caching instead of solving an NP-hard optimization problem optimally.

– We test our approach on synthetic data for two different combinatorial problems, namely, the knapsack (KP) and prize-collecting traveling salesman problems (PCTSP), of different sizes. The results show that our approach is an improvement over the standard SP not only in terms of computation time but also in terms of the quality of learning.

2 Background: Structured Output Prediction

The general goal of SOP is to predict the most relevant structured output given a description of an input problem. Formally, let $\mathcal{D} = \{(\boldsymbol{x}, \boldsymbol{y})\}$ be a dataset, where $\boldsymbol{x}$ is an input structure (e.g. properties and structural constraints) and $\boldsymbol{y}$ is its corresponding structured output. $\mathcal{X}$ is the set of all possible input structures and $\mathcal{Y}$ is the structured output space consisting of all possible outputs that fulfill the required structure. The objective of SOP is to learn a hypothesis function $h \in \mathcal{H}$ such that for any instance $\boldsymbol{x}$, it produces an output $\boldsymbol{y'} = h(\boldsymbol{x})$ such that $\boldsymbol{y'} \in \mathcal{Y}$ and the distance $dist(\boldsymbol{y}, \boldsymbol{y'})$ between the desired and the predicted outputs is as small as possible. The definition of $dist(\boldsymbol{y}, \boldsymbol{y'})$ is problem specific, e.g. the number of non-overlapping elements in a set prediction problem.

Multiple algorithms for SOP exist, including the structured perceptron [4], stochastic sub-gradient [12], and cutting-plane algorithms [8]. We will build on the seminal SP algorithm of Collins (see [4]). The SP is limited to learning a linear function over its input. However, this setting exactly matches our case of learning the weights of a linear objective function (next section).

The key issue in structured output prediction is that $\boldsymbol{x}$ and $\boldsymbol{y}$ are structured. To be able to learn a linear function with weights $\boldsymbol{w}$, the SP assumes that we can define a problem-specific representation function Φ that maps every valid input/output pair to a feature vector: $\Phi : \mathcal{X} \times \mathcal{Y} \mapsto \mathbb{R}^{|\boldsymbol{w}|}$. Given an input structure $\boldsymbol{x}$ and a (learned) weight vector $\boldsymbol{w}$, decoding an output structure (called the *inference* task) then becomes: $h(\boldsymbol{x}) = \arg\max_{\boldsymbol{y'} \in \mathcal{Y}} \boldsymbol{w}^\top \Phi(\boldsymbol{x}, \boldsymbol{y'})$.

SOP shares many similarities to techniques from inverse optimization (IO) for mixed-integer linear problems [3, 14]. The SP method could be seen as a variation of IO without an initial weight vector, with multiple solutions for different feasible spaces, and focusing on generalizing to unseen instances rather than reconstructing the underlying weight vector. The ideas from our work can be extended to other methods from SOP and IO in follow-up work.

Structured Perceptron Algorithm Algorithm 1 presents its pseudocode. It starts with an arbitrary initialization of the weights on line 1, e.g. using unit weights. As long as a global time limit T has not been reached, it will go over each $(\boldsymbol{x}, \boldsymbol{y})$ instance in the data, and call the inference algorithm to find the output structure that maximizes $\boldsymbol{w}^\top \boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y'})$ given the current weight vector $\boldsymbol{w}$. The condition on line 5 then checks whether the score of $\boldsymbol{y'} : \boldsymbol{w}^\top \Phi(\boldsymbol{x}, \boldsymbol{y'})$ is higher than the score of the desired solution $\boldsymbol{y}$. If so $\boldsymbol{y} \notin h(\boldsymbol{x})$; hence there is an error and we should update the weights to make $\boldsymbol{y}$ more favorable. This is done in line 6, where coefficients for features where $\boldsymbol{y}$ has a higher value are increased and others are decreased (scaled by some learning rate η).

Algorithm 1 Structured Perceptron

```
Require: Training dataset $\mathcal{D} = \{(\boldsymbol{x}, \boldsymbol{y})\}$, maximum time limit $T$, learning rate $\eta$
1: $\boldsymbol{w} \leftarrow \mathbf{1}$    ▷ Parameter initialization
2: while time limit $T$ has not been reached do
3:     for each instance $(\boldsymbol{x}, \boldsymbol{y})$ in $\mathcal{D}$ do
4:         Find $\boldsymbol{y'} \in \mathcal{Y}$ that maximizes $\boldsymbol{w}^\top \boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y'})$    ▷ Call inference algorithm
5:         if $\boldsymbol{w}^\top \boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y'}) > \boldsymbol{w}^\top \boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y})$ then
6:             $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta(\boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y}) - \boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y'}))$    ▷ Parameter update
7: return $\boldsymbol{w}$
```

3 SP for Multi-Objective Combinatorial Optimization

The problem definition can be given as: For a given data set of features (that fully characterize the optimization problem) and solution pairs, the goal of SP for MOCOP is to learn a weight vector w that minimizes the distance between the weighted linearization of the predicted and true solutions. The learned weights can be used in decision-making for unseen features in the future.

Let $z_1(y), \ldots, z_p(y)$ be the p objective functions of a MOCOP and let $\mathcal{C}(y)$ represent the set of constraints it must satisfy. A formulation with a weighted linearization of the sub-objectives can be written as:

$$\begin{aligned} \max_{y'} \quad & w_1 z_1(y') + \ldots + w_p z_p(y') \\ \text{s.t.} \quad & \mathcal{C}(y') \end{aligned} \tag{1}$$

where the weights $w_1, \ldots, w_p$ represent the importance of sub-objectives $1, \ldots, p$, respectively. The formulation (1) is commonly used in industrial applications where the different sub-objectives are typically penalties or bonuses, such as cost, energy use, profitability, overtime penalties, smoothness penalties, and so forth. However, setting the weights $\boldsymbol{w} = (w_1, \ldots, w_p)$ can be a difficult task, where machine learning could help in learning a weight vector $\boldsymbol{w}$ based on, e.g. historically manually created solutions.

3.1 Input structure and representation function $\boldsymbol{\Phi}$

We now show how the SP algorithm can be used, in case of a set of problem instances and solutions for which the constraint set $\mathcal{C}$ may change, but for which the number of sub-objective functions $(z_1, ..., z_p)$ stays the same as well as the nature of what they compute. For example, the first sub-objective might compute the total cost of the solution, the second the total energy consumption, the third the time-to-build, the fourth how much the total overdue time is, etc.

More formally we assume a dataset $\{(\boldsymbol{x},\boldsymbol{y})^i\}$ where $\boldsymbol{x}^i = (\mathcal{C}^i, (z_1^i, \ldots, z_p^i))$ fully characterizes the optimization problem, and $\boldsymbol{y}^i$ is a solution that satisfies $\mathcal{C}^i$.

Given a MOCOP's input structure $\boldsymbol{x} = (\mathcal{C}, (z_1, \ldots, z_p))$ we can now see that the representation function Φ is simply the vector of the sub-objective values, i.e., $\Phi(\boldsymbol{x},\boldsymbol{y}) = (z_1(\boldsymbol{y}), \ldots, z_p(\boldsymbol{y}))$ and the inference task in Algorithm 1 line 4 becomes : $h(\boldsymbol{x}) = \arg\max_{\boldsymbol{y'} \in \mathcal{C}} w_1 z_1(\boldsymbol{y'}) + \ldots + w_p z_p(\boldsymbol{y'})$, that is exactly our weighted single-objective formulation.

Example 1. For example, assume a bi-objective KP where $\boldsymbol{y}$ is a vector of binary decision variables, and we have two value vectors v_1 and v_2 and a size vector s with a total size limit of l. The mathematical problem formulation is $\max_{\boldsymbol{y'}} w_1(v_1^\top \boldsymbol{y'}) + w_2(v_2^\top \boldsymbol{y'})$ s.t. $s^\top \boldsymbol{y'} \leq l$ which corresponds to input structure $\boldsymbol{x} = ((s,l),(v_1,v_2))$ and the representation function $\Phi(\boldsymbol{x},\boldsymbol{y}) = (v_1^\top \boldsymbol{y}, v_2^\top \boldsymbol{y})$. Note how for different instances $\{(\boldsymbol{x},\boldsymbol{y})^i\}$ the items might be different, as well as their sizes and values, but the goal is always to balance the first sub-objective (e.g. cost) with the second (e.g. durability).

3.2 Scaling up the SP algorithm

The main issue with using SOP methods like the SP in this setting is that on line 4 of Algorithm 1, we will have to solve the single-objective CO repeatedly for each instance at each epoch. Solving it for one instance may take minutes to hours, which would make training prohibitively slow.

The key observation that we exploit in this paper is that we do not need to compute (and prove) the optimal solution for the learning to work. In fact, any feasible solution $\boldsymbol{y'}$ for which $\boldsymbol{w}^\top \boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y'}) > \boldsymbol{w}^\top \boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y})$ (the update condition on line 5) can be used to do a *valid parameter update* step in line 6.

We explore in this paper three orthogonal techniques for quickly finding a feasible solution $\boldsymbol{y'} \in \mathcal{C}$ that satisfies the update condition. We consider computational improvements that have been used in other related areas such as bilevel optimization [2] and decision-focused learning [9, 11].

Heuristic solving. The first approach we can use is to *replace* the exact solver by a heuristic solver. This could be a greedy algorithm, a local search algorithm, or even the exact solver with a limited timeout. While this is not guaranteed to find a solution that satisfies the valid update condition, if it does find such a solution then it can be used. If not, this instance will be skipped in the hopes of encountering other instances for which the heuristic method does find a usable solution.

Satisfying the update condition. An orthogonal approach is based on the realization that an exact solver will spend a lot of time proving optimality. A way to avoid this effort is to stop the solver as soon as it finds a solution that satisfies the update condition. Alternatively, we can remove the objective function from the formulation and instead add the valid update condition directly as a constraint. This turns the problem into a satisfaction problem, which we expect will often be easier to solve than the optimization variant.

Caching feasible solutions. A third approach is to *cache* all feasible solutions found in previous solve calls. That is, for any $\mathcal{C}^i$, a feasible solution found for one weight vector w will still be a feasible solution for another weight vector w (this might not be true for different instances with different $\mathcal{C}^i$ hence a separate cache for every $\mathcal{C}^i$ should be maintained). Given such a cache, before calling the solver (on line 4), we can first do a simple linear scan over the solutions in the cache to see if a feasible solution satisfies the valid update condition. If so, no solver needs to be called and an update step can immediately be applied. Otherwise, no solution in the cache can be used and a solver should be called.

4 Optimization Problems studied

In this section, we present multi-objective versions of two COPs, namely the Knapsack Problem (KP) and the Price Collecting Traveling Salesman Problem (PCTSP), used in the computational experiments. The motivations, mathematical models, and the data generation process for both problems are given in the Appendix.

Multi-Objective Knapsack Problem This problem is the one introduced in example 1, but generalized to p sub-objectives.

Multi-Objective Prize Collecting Traveling Salesman Problem Let $G = (V, E)$ be a complete undirected graph, where V is the set of cities and E is the set of edges connecting the cities. Each edge $(i, j) \in E$ has p different values $v_1^{ij}, \ldots, v_p^{ij}$ that can be interpreted as different features of that edge such as travel distance, fuel consumption, the total ascent amount, etc., for which smaller values are preferred. Each city $i \in V$ has a penalty γ^i for not being visited and a reward π^i for being visited. Then, the objective of the single-objective formulation (1) for the multi-objective PCTSP is $\min_y \sum_{(i,j)\in E} \left(w_1 v_1^{ij} + \ldots + w_p v_p^{ij}\right) y_{ij} + w_{p+1} \sum_{i\in V} \gamma^i (1 - o_i)$ where y_{ij} is the binary variable indicating whether edge (i, j) is in the tour ($y_{ij} = 1$) or not and o_i is the binary variable indicating whether city is visited ($o_i = 1$) or not. The constraints of the model ensure there is a valid single circuit, link the o_i and y_{ij} variables, and constrain the given minimum reward.

5 Experiments

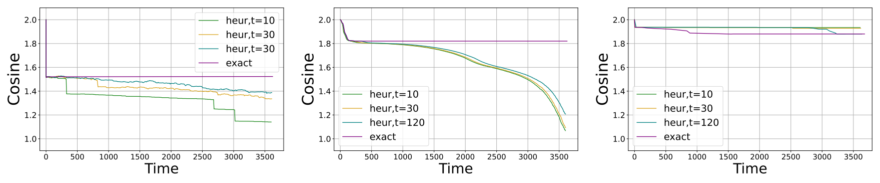
Three metrics are used to evaluate the performance of our proposed techniques: (1) the cosine distance between the real weights and the predicted weights during the learning process (lower is better), (2) the solution accuracy which measures the similarity of the decisions made with the predicted weight vector compared to the real one (higher is better) and the optimality gap with respect to the true coefficients. The mathematical formulas and more details of these metrics are given in the Appendix. The research questions we answer in the computational experiments are:

- **RQ1**: How does using a heuristic versus an exact solver impact the speed and quality of learning good weight vectors?
- **RQ2**: Is there an additional benefit to just computing *any* solution that leads to a valid update rule during training?
- **RQ3**: To what extent does the use of a solution cache avoid the need to call the solver, and what is the influence on learning speed and quality?
- **RQ4**: How similar are the *solutions* obtained from solving with the true weight vector and the predicted weight vectors?

RQ1: Exact versus heuristic solving Figure 1 shows the cosine distance between the true and predicted weight vectors, using an exact or heuristic solver. Fig. 1a shows the results for knapsack (KP) with 1000 items, and Fig. 1b and 1c for the PCTSP with 100/200 stops.

First, for the KP, using an exact solver (shown in purple in Fig. 1a) does not allow for improving the predicted weight vector after the first ten iterations. It spends the rest of the training time unsuccessfully trying to find an optimal solution for the training instance. When using a timeout, leading to heuristic solutions, the algorithm continues to decrease its cosine distance because it can identify valid updates even for instances that are difficult to solve to optimality.

For the PCTSP with 100 stops, the exact solver encounters a similar issue (shown in purple in Fig. 1b), after a number of successful update steps it encounters a hard-to-solve instance and spends the remaining time attempting to solve it. On the other hand, the use of timeouts allows the learning to continue. Examining this behavior in more detail, we find that the algorithm simply skips a hard-to-solve instance during its first epoch to then solve other instances that lead to valid updates. Finally, for the PCTSP with 200 stops, the exact solver behaves analogous to this dataset. But unlike the previous cases, adding a timeout does not lead to better results; in most cases the timeout is too low to find a solution, meaning learning can't progress much either.

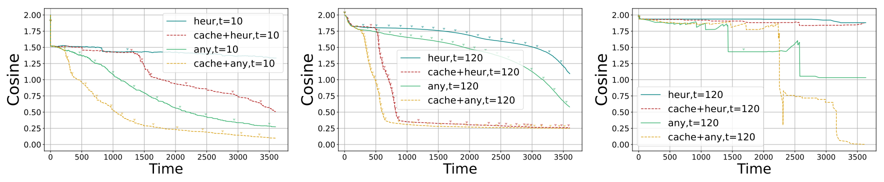


(a) KP - 1000 (b) PCTSP - 100 (c) PCTSP - 200

Fig. 1: Cosine distance over time for the heuristic versus an exact solver

RQ2: valid-update solution Figure 2 presents the cosine distance between the true and predicted weight vectors when using our valid-update condition as a constraint ('any'), we discuss the dotted 'cache' lines later. Based on our previous experiment, we used a timeout of 10 seconds for KP and 120 seconds for PCTSP instances. The inverted triangle markers indicate the completion of an epoch (each instance is solved once in an epoch).

We first focus our attention on the solid lines (blue and green, 'heur' vs. 'any'). For every dataset tested, our valid-update approach ('any') provides a benefit over heuristic solving as the cosine distances decrease faster. Solving the satisfaction problem allows for a larger number of valid updates compared to solving with a timeout. In Fig. 2c, we observe for PCTSP-200 spikes in the graph corresponding to an *increase* in the cosine distance for the 'any' method. We hypothesize that sometimes the solution found when searching for 'any' valid update, might be very different leading to a direction that is not well aligned with the true cost vector; however, we see that despite these spikes the learning does converge to lower cosine distances.



(a) KP - 1000 (b) PCTSP - 100 (c) PCTSP - 200

Fig. 2: Cosine distance over time for the techniques presented

RQ3: Caching feasible solutions We now focus on the yellow and red lines in Figure 2 to observe the effect of solution caching. In the first epoch, the performance is identical to not using a cache, as all caches are still empty. We observe that from the second epoch the learning proceeds quicker (more epochs per time unit) and better (faster decreasing cosine distances) when using a cache. Using a cache on top of the 'any' method (yellow lines) leads to the best results across all datasets.

RQ4: Solution quality of the predicted weights Finally, we examine the effect of the proposed methods on the *solutions* found.

Learning method	KP - 1000		PCTSP - 100		PCTSP - 200	
	Score	Gap	Score	Gap	Score	Gap
Exact	96.8%	**4e-4**	33.4%	0.64	32.9%	2.08
Cache + heur, $t = 10$	96.4%	6e-4	92.6%	3e-4	33.8%	1.18
Cache + any, $t = 10$	**98.3%**	5e-4	**93.9%**	**1e-4**	**75.7%**	**3e-3**

The above table compares the similarity between the solution obtained with the ground truth weights versus the one obtained with the predicted weights (Score: higher is more similar) and the optimality gap with respect to the true coefficients (Gap: lower is better). We show the results for 'exact' and caching with heuristic/any valid solution. We can see that better cosine distances (previous experiments) also lead to better solutions. We also can see that the 'cache + any' technique leads to the best optimality gap for the PCTSP, with the biggest improvement occurring for our largest dataset.

we need to see if we use the term gap... I haven't updated the appendix yet

6 Conclusion

This paper investigates the use of structured output prediction to learn the weights of hard multi-objective combinatorial optimization problems. The key observation of the proposed approach is that we don't need to solve to optimality to get a valid update for learning. We propose and evaluate three techniques to speed up the solving, and hence the learning: the use of heuristic solving (time limits), the incorporation of a 'valid update' condition as a constraint, and a cache that allows to sometimes skip calling the solver altogether.

Our experiments on multi-objective KP and PCTSP problems showed that these techniques speed up the learning compared to using an exact solver during training, with the combination of caching and solving to find any valid update leading to the best and most scalable results. This opens the door to using structured output prediction to learn weights in more realistic and hard-to-solve industrial problems.

For future work, we consider the application of cut generation and methods from bi-level optimization when modified to handle multiple feasible regions. This approach is non-trivial, with scalability and convergence being major issues.

References

1. Bemporad, A., de la Peña, D.M.: Multiobjective model predictive control. Automatica **45**(12), 2823–2830 (2009)
2. Camacho-Vallejo, J.F., Corpus, C., Villegas, J.G.: Metaheuristics for bilevel optimization: A comprehensive review. Computers & Operations Research p. 106410 (2023)
3. Chan, T.C., Mahmood, R., Zhu, I.Y.: Inverse optimization: Theory and applications. Operations Research (2023)
4. Collins, M.: Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In: Proceedings of the 2002 conference on empirical methods in natural language processing (EMNLP 2002). pp. 1–8 (2002)
5. Deb, K., Sindhya, K., Hakanen, J.: Multi-objective optimization. In: Decision sciences, pp. 161–200. CRC Press (2016)
6. Ganesan, T., Elamvazuthi, I., Shaari, K.Z.K., Vasant, P.: Hypervolume-driven analytical programming for solar-powered irrigation system optimization. In: Nostradamus 2013: Prediction, Modeling and Analysis of Complex Systems. pp. 147–154. Springer (2013)
7. Jayarathna, C.P., Agdas, D., Dawes, L., Yigitcanlar, T.: Multi-objective optimization for sustainable supply chain and logistics: a review. Sustainability **13**(24), 13617 (2021)
8. Joachims, T., Hofmann, T., Yue, Y., Yu, C.N.: Predicting structured objects with support vector machines. Communications of the ACM **52**(11), 97–104 (2009)
9. Mandi, J., Stuckey, P.J., Guns, T., et al.: Smart predict-and-optimize for hard combinatorial optimization problems. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 34, pp. 1603–1610 (2020)

10. Marler, R.T., Arora, J.S.: The weighted sum method for multi-objective optimization: new insights. Structural and multidisciplinary optimization **41**, 853–862 (2010)
11. Mulamba, M., Mandi, J., Diligenti, M., Lombardi, M., Bucarey, V., Guns, T.: Contrastive losses and solution caching for predict-and-optimize. arXiv preprint arXiv:2011.05354 (2020)
12. Ratliff, N., Bagnell, J.A., Zinkevich, M.: Subgradient methods for maximum margin structured learning. In: ICML workshop on learning in structured output spaces. vol. 46 (2006)
13. Tapia, M.G.C., Coello, C.A.C.: Applications of multi-objective evolutionary algorithms in economics and finance: A survey. In: 2007 IEEE congress on evolutionary computation. pp. 532–539. IEEE (2007)
14. Wang, L.: Cutting plane algorithms for the inverse mixed integer linear programming problem. Operations research letters **37**(2), 114–116 (2009)