Learning Heuristics for Combinatorial Assignment Problems by Optimally Solving Subproblems

Fredrik Präntare
Linköping University
Linköping, Sweden
fredrik.prantare@liu.se

Herman Appelgren
Linköping University
Linköping, Sweden
herman.appelgren@liu.se

Mattias Tiger
Linköping University
Linköping, Sweden
mattias.tiger@liu.se

David Bergström
Linköping University
Linköping, Sweden
david.bergstrom@liu.se

Fredrik Heintz
Linköping University
Linköping, Sweden
fredrik.heintz@liu.se

ABSTRACT

Hand-crafting accurate heuristics for optimization problems is often costly due to requiring expert knowledge and time-consuming parameter tuning. Automating this procedure using machine learning has in recent years shown great promise. However, a large number of important problem classes remain unexplored. This paper investigates one such class by exploring learning-based methods for generating heuristics to perform value-maximizing combinatorial assignment (the partitioning of elements among alternatives). In more detail, we use machine learning leveraged by generating and optimally solving subproblems to produce heuristics that can, for example, be used with search algorithms to find feasible solutions of higher quality more quickly. Our results show that our learned heuristics outperform the state of the art in several benchmarks.

KEYWORDS

Operations Research; Machine Learning; Deep Learning; Heuristic Search; Combinatorial Optimization; Neural Networks; Inapproximability; Coalition Formation; Combinatorial Auctions

ACM Reference Format:
Fredrik Präntare, Herman Appelgren, Mattias Tiger, David Bergström, and Fredrik Heintz. 2022. Learning Heuristics for Combinatorial Assignment Problems by Optimally Solving Subproblems. In *Proc. of the 21st International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2022), Online, May 9–13, 2022*, IFAAMAS, 9 pages.

1 INTRODUCTION

A fundamental challenge in computer science is that of designing different types of cost-effective, scalable assignment algorithms. We consider a highly challenging and general problem of this type, namely that of *utilitarian combinatorial assignment* (UCA), in which indivisible elements (e.g., sensors, goods, agents) have to be distributed in *bundles* (pairwise disjoint subsets) among a set of alternatives (e.g., targets, buyers, jobs) to maximize a notion of aggregated expected utility. This is a central problem in *artificial intelligence* (AI), *operations research* (OR), and *game theory* (GT); with applications in for example task/resource allocation [1, 9, 32], combinatorial auctions [40], multi-target tracking and sensor fusion [10], and team/coalition formation [32].

Unfortunately, this problem is computationally difficult—even under limiting restrictions. The state-of-the-art algorithms can only compute optima to problems with extremely few elements (up to roughly 25) [33]. Moreover, for concisely defined problems, we show that, unless P = NP, there exists no polynomial-time *approximation algorithm*[1] that can find a *feasible solution* (a potentially sub-optimal assignment) with a *provably* good worst-case ratio (i.e., the returned output is within some multiplicative factor of the optimum). In light of this, it is important to experimentally investigate if, when and how heuristic algorithms that do not provide worst-case ratio guarantees can generate feasible solutions of high-enough quality for problems with large-scale inputs and limited computation budgets. However, manually designing accurate heuristics, and deciding which one to use for a specific situation—in essence solving the *algorithm selection problem* manually [38]—can be extremely costly due to requiring both expert knowledge and tuning the heuristics' parameters. For similar reasons, existing heuristics tend to not perform well because they are not able to exploit the problem's underlying distribution in a satisfying manner. In essence, to challenge the state of the art without learning, one would have to manually design a new heuristic for every problem type and/or application. Instead, we circumvent this issue by creating a general-purpose method that learns to exploit a combinatorial assignment problem instance's underlying distribution without any *a priori* knowledge of it.

Automating similar procedures with learning methods has in recent years shown great promise for many types of problems, including (but not limited to) *route-finding* [11, 20], *graph problems* [19], *boolean satisfiability* [54], and *tree search* [28]. Moreover, heuristic search with a learned heuristic has achieved super-human performance in playing many difficult games with massive state spaces. A key problem in solving such games is to have a sufficiently good approximation of the expected utility one can achieve from any state. Progress within the deep learning field with *multi-layered neural networks* has made learning such an approximation possible in a number of settings [22, 47], but many remain unexplored.

Proc. of the 21st International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2022), P. Faliszewski, V. Mascardi, C. Pelachaud, M.E. Taylor (eds.), May 9–13, 2022, Online.

[1] An *approximation* algorithm is one that yields some provable *a priori* guarantee on the quality of its output, such as on the distance of its output to optimal or the ground truth. In contrast, e.g., neural networks, while powerful in many applications, are *function approximators* that, in general, yield no approximation guarantees.

In a similar vein to these examples, we investigate and develop the first general-purpose learning-based method for generating heuristics for combinatorial assignment problems. We also present theoretical and experimental foundations for using function approximators, such as neural networks, to solve combinatorial assignment problems. More specifically, our main contributions include:

- We are the first to explore and provide a method for generating UCA heuristics. This work includes *i)* a novel training regime reminiscent of *curriculum learning* [3] and *problem reduction machine learning* (see e.g., [48]), with which we are the first to use an optimal solver to compute optimal solutions to smaller "subproblems" that we aggregate to predict optimal solutions for the full problem; *ii)* a general method for generating UCA training data; and *iii)* a configuration for using a state-of-the-art gradient boosted decision trees method in addition to two easy-to-implement neural network architectures for performing heuristic UCA.
- We benchmark our learning method on several problem instances, and show that the heuristics it generates outperform the state of the art in several standardized, difficult tests.

The remainder of this paper is structured as follows. We begin by presenting related work in Section 2. Then, in Section 3, we define important concepts and discuss UCA's computational hardness. In Section 4, we describe our method. In Section 5, we present our experiments. Finally, in Section 6, we conclude with a summary.

2 RELATED WORK

The most studied UCA applications can be divided into three areas, and the most related combinatorial optimization problems in them can be summarized as follows.

***In AI:* Coalition structure generation *(CSG)*.** A variation of UCA in which we seek to generate a number of value-maximizing agent groups. However, CSG does not model alternatives explicitly, which arguably makes CSG less suitable for alternative-oriented situations, e.g., when each group of agents should be assigned to achieve a specific goal. [36]

***In OR: The* generalized assignment problem *(GAP)*.** In the GAP, each alternative has a *capacity*, and the *value function* (defined in Section 3) is additive, so there cannot be any synergies between the elements. The GAP is for these reasons more similar to knapsack problems, and the lack of synergies makes the problem easier from a computational perspective since it yields approximability. It is also a special case of UCA, so heuristics for UCA can be used for the GAP (we explore this in Section 5). [6]

***In GT: The* winner determination problem *(WDP)*.** This is a variation of UCA for which only a subset of the bundles are allowed. These are given as a list of explicit "bids" that reveal how much a number of bidders value different bundles of goods. Each valuation is assumed to be non-negative. The goal in this problem is to find an allocation of goods that maximizes the auctioneer's profit. The algorithmic literature for the WDP has focused on "small" value functions (i.e., small bid lists). Contrastively, we are concerned with large, exhaustive ones that are exponential in the number of elements. While the WDP solvers can theoretically be used for many UCA problems (more specifically those with a non-negative value function), the best WDP solvers are unable to handle the large number of bundles used in UCA due to being designed for small-sized bid lists. WDP solvers are also designed to handle the situations where no bid allocation can be found due to missing bids, which is circumvented in UCA since its value function is exhaustive (i.e., all allocations are allowed). [25]

Although several meta-heuristic algorithms have been proposed for these problems [5, 50, 51, 53], no heuristic generation methods have been devised for them. Other noteworthy special cases of UCA include the *multiple traveling salesperson problem* [7], in which the salespersons correspond to alternatives, while the elements represent cities; *spectrum repacking*, which [29] explored with deep optimization; and many variations of *weighted matching*, which [52] surveyed from a machine learning perspective.

In addition to the aforementioned three problems, UCA is equivalent to *simultaneous coalition structure generation and assignment* when the indivisible elements are viewed as agents [32], and related to *multi-agent resource allocation in n-additive domains* when they are viewed as goods [9]. The state-of-the-art optimal algorithm for these problems was developed by [33], which is thus also the state of the art for optimal UCA. While their algorithm outperforms industry-grade solvers like *CPLEX* in difficult benchmarks, it can only solve fairly small problems, and there is no proven guarantee that it will always terminate in less time than exhaustive search. Apart from this work, a few heuristic UCA algorithms have been explored, including *Monte Carlo tree search*, *simulated annealing*, and *local search* [31]. A major drawback of the heuristics deployed by these algorithms is that they are not able to learn and exploit the underlying problem distribution.

In more general for machine learning, there has been work in using a learned heuristic in search [2, 16]. There has also been work in combining previous optimization methods, such as *branch-and-bound*, with learning [12, 15, 24]. Another category is *end-to-end learning*, in which machine learning is used to learn a function that outputs solutions directly. While the end-to-end approach has been applied to important problems such as the *traveling salesperson problem* [19], the *multi-unit winner determination problem* [23], and the *propositional satisfiability problem* [43], the learned heuristic approach—which we also pursue in this paper—remains dominating [42], since it allows for combining the advantages of combinatorial optimization with new advances in machine learning.

3 BASIC CONCEPTS AND COMPLEXITY

UCA, the problem that we investigate in this paper, is defined as the following optimization problem.

Input: A triple $\langle E, A, \boldsymbol{v} \rangle$, where $E = \{e_1, \ldots, e_n\}$ is a set of elements, $A = \{1, \ldots, m\}$ is a set of alternatives, and $\boldsymbol{v} : 2^E \times A \mapsto \mathbb{R}$ is a function (called the *value function*) that maps a value (e.g., expected utility) to every pairing of a bundle $B \subseteq E$ to an alternative $a \in A$.

Output: A *combinatorial assignment* (Definition 1) $C = \langle B_1, \ldots, B_m \rangle$ over E that maximizes its *value* defined with $\boldsymbol{V}(C) = \sum_{i=1}^{m} \boldsymbol{v}(B_i, i)$.

Definition 1. *The tuple* $\langle B_1, \ldots, B_m \rangle$ *is a* combinatorial assignment over E *if* $B_i \cap B_j = \emptyset$ *for all* $i \neq j$, *and* $\bigcup_{i=1}^{m} B_i = E$.

Moreover, we overload $\boldsymbol{V}(P) = \sum_{i=1}^{m} \boldsymbol{v}(B_i, i)$ to also denote the value of a *partial assignment* (Definition 2) $P = \langle B_1, \ldots, B_m \rangle$, and define $\|P\| = \sum_{i=1}^{m} |B_i|$. (Note that we deliberately define the concept of a partial assignment so that a combinatorial assignment over E is also a partial assignment over E.) We also use Π_E for the set of all combinatorial assignments over E, and define $\Pi_E^k = \{C \in \Pi_E : |C| = k\}$ for $k \in \{1, \ldots, m\}$. We use the conventions $n = |E|$ and $m = |A|$, and say that a combinatorial assignment C^* is *optimal* if and only if:

$$\boldsymbol{V}(C^*) = \max_{C \in \Pi_E^m} \boldsymbol{V}(C).$$

Definition 2. *C is a* partial assignment over *E if C is a combinatorial assignment over any $E' \subseteq E$.*

While [31] already proved UCA's NP-hardness by a reduction from CSG to simultaneous coalition structure generation and assignment, we now prove that UCA is also hard to approximate. This rules out approximation algorithms with solution guarantees as a general-purpose option for UCA, since it provides evidence that we are not able to construct a method that can generate a solution of sufficiently high quality for *all* UCA instances in polynomial time. This further motivates our quest to explore machine learning, since it may enable us to handle many more problem classes without requiring human intervention to analyze what heuristics best suit a specific type of problem class (e.g., value distribution), or a specific application (e.g., multi-vehicle routing).

A first observation is that the size of the input for arbitrary UCA problems is exponential in the number of elements. More specifically, to specify an arbitrary value function you need, in worst case, $m2^n$ entries dedicated for it in the input. Using dynamic programming, one can find optimum in $O(m3^n)$ [33]. This is exponential in n, but polynomial in the input's size $O(m2^n)$, since:

$$m3^n = m2^{(log_2 3)n} = m(2^n)^{log_2 3}.$$

Suppose we only gave $k < m2^n$ of the value function's entries explicitly instead, and define the remaining values concisely (for example as an arbitrary value)—then, can we construct a polynomial-time approximation algorithm with provably good worst-case guarantees? This is unfortunately not possible, unless P = NP, as stated by Theorem 1. (Note that the same result can also be achieved through a reduction from the APX-hard WDP. This is because from a complexity perspective, the WDP is the special case of UCA in which we only allow non-negative and *monotone* value functions.) A corollary of Theorem 1's proof is that UCA is also *APX-hard*[2] (Theorem 2). Note that while the general UCA problem is hard to approximate, many important restricted instances can be approximated and/or solved more efficiently—see for example earlier results on various related combinatorial auction, maximum clique, and set packing problems such as [13, 14, 26].

Theorem 1 (Inapproximability). *Unless* P = NP, *there is no polynomial-time algorithm that approximates UCA to a ratio of $c \leq k^{1-\epsilon}$ for any $\epsilon > 0$, where k is the number of values in the input.*

[2]The complexity class APX is the set of optimization problems in NP that can be approximated with an approximation guarantee that is bounded by a constant.

Proof. We follow a similar proof procedure as the one that [39] provided for their combinatorial auction winner determination inapproximability result, with the difference that we alter their *weighted independent set* (WIS) reduction so that it works for UCA. First, recall that in the WIS problem, the input is an undirected graph with weighted vertices, and the output is an *independent set* (a subset of the vertices that are pairwise non-adjacent) with maximum aggregated weight. With this in mind, for sake of contradiction, assume that there exists a poly-time algorithm that approximates UCA to a ratio $c \leq k^{1-\epsilon}$. Then, that algorithm could be used to c-approximate the WIS problem in polynomial time. This can be shown through the following approximation-preserving polynomial-time reduction from the WIS to UCA. First, create one element for each edge and one alternative for each vertex in the graph. Let $\boldsymbol{I}(a)$ be the set of elements that represent an edge *incident* to the vertex represented by the alternative a. Now define $\boldsymbol{v}(B, a)$ (UCA's value function) to be equal to the weight of the vertex represented by a if the bundle $B = \boldsymbol{I}(a)$, and 0 otherwise. This completes the reduction, which means that the algorithm can also c-approximate the independent set problem in polynomial time, and can then also c-approximate the *maximum clique* problem. This leads to a contradiction, since [13] and [55] showed that, unless P = NP, there exists no algorithm that can always establish such a bound in polynomial time. □

Theorem 2 (APX-hardness). *UCA is* APX*-hard.*

Proof. This follows from Theorem 1's proof, which provides a linear reduction from an APX-hard problem to UCA. □

A final note is that UCA and CSG are problems that can be converted into each other in polynomial time with only a polynomial increase in the input size. From a complexity perspective, the general cases of these problems are in this sense equivalent. However, these conversions greatly affect the problems' search spaces (i.e., their sizes and distributions), and algorithms designed for e.g., CSG can in practice be extremely inefficient for UCA, and vice versa.

4 GENERATING HEURISTICS

In an attempt to counter UCA's inapproximability, and to formally express our machine learning approach to generating heuristics for UCA, first let $\langle e'_1, \ldots, e'_n \rangle$ be any permutation of E, and define:

$$\boldsymbol{V}^*(C) = \begin{cases} \boldsymbol{V}(C) & \text{if } \|C\| = n \\ \max_{C' \in \zeta(C, e'_{\|C\|+1})} \boldsymbol{V}^*(C') & \text{otherwise} \end{cases}, \quad (1)$$

where

$$\zeta\big(\langle B_1, \ldots, B_m \rangle, e\big) = \bigcup_{i=1}^{m} \Big\{ \langle B_1, \ldots, B_i \cup \{e\}, \ldots, B_m \rangle \Big\},$$

and C is a combinatorial assignment over $\{e'_1, \ldots, e'_{\|C\|}\}$, where $\|C\| \leq n$. As a consequence of Theorem 3, UCA boils down to computing recurrence (1). One of the main goals of this paper is to investigate approximating this recurrence. Such approximations can then be used in conjunction with tree/graph search algorithms such as *Monte Carlo tree search*, for example in a similar fashion as [47] did for solving difficult board games.

Theorem 3 (Optimal substructure). *If $P = \langle B_1, \ldots, B_m \rangle$ is a partial assignment over E, it holds that $\boldsymbol{V}^*(P) = \max_{C \in \Psi} \boldsymbol{V}(C)$, where $\Psi = \left\{ \langle B'_1, \ldots, B'_m \rangle \in \Pi_E^m : B_i \subseteq B'_i \text{ for } i = 1, \ldots, m \right\}$.*

Proof. This result follows directly by induction. □

Our approach can thus be viewed as treating a partial assignment over E as a "state", and UCA as the decision-making problem where we want to assign the unassigned elements in a way that results in an assignment that maximizes the expected value. Our approximators attempt to estimate the maximum possible "gain" from a specific partial assignment without conducting costly look-ahead.

Note that if we *sequentially* assign any remaining unassigned elements, Theorem 3 reveals one way in which UCA relates to dynamic programming and reinforcement learning, since we are then in essence trying to approximate the Q-function for a sequential decision-making problem. Of course, this type of ordering is entirely artificial, but it can be exploited by certain heuristic methods, such as the ones proposed in [31]. For the same reason, i.e. since UCA is not sequential, we expect architectures such as recurrent neural networks and *pointer networks* [49] to perform worse than their non-sequential (e.g., feedforward) counterparts. Section 5 includes experiments that corroborates this hypothesis.

Heuristic Function Models

A key question that arises from the previous section is: which learning-based function approximator (e.g., architecture/method) is the one that is most suitable for approximating (1)? Naturally, the answer depends on the context, such as the value function's distribution, the problem's input size, and the time available for training. Thus, as a proof of concept to show that our approach to generating UCA heuristics works, we attempt to approximate (1) using different types of approximators and benchmark them against the state-of-the-art general-purpose heuristics. We provide architectures for two neural networks that can be used for UCA: i) a deep *feedforward neural network* (FNN), and ii) a *recurrent neural network* (RNN). In addition, we also use a state-of-the-art *gradient boosting decision trees* method. If our approximators outperform the state of the art, we expect there to exist more specialized approximators that yield even better results, which can be explored further in future work.

Note that the artificially constructed partial assignment representations that we use henceforth are not (semantically) tabular in the conventional sense, such as in the data sets in for example [17] or [46]. Similar to a natural image, there are no explicit meaningful features in either our rows, columns or the matrix elements. However, as opposed to a typical natural image or coordinate frame, no trivial locality or smoothness in the input space is guaranteed (or expected). Consequently, it is likely challenging to design useful kernels, or similarity measures, for kernel-based or non-parametric methods. A main motivation of our work is to avoid such handcrafted in-depth heuristic design for every interesting domain.

Feedforward Neural Network. Our first approximation for (1) uses a θ-parameterized fully connected deep neural network $\boldsymbol{d}_\theta(P)$:

$$\boldsymbol{d}_\theta(P) \approx \boldsymbol{V}^*(P),$$

where P is a partial assignment. This network uses *ReLU* [22] activation functions and a parametric number d of w-wide hidden layers. The input is of size $mn+1$ and consists of a vectorized *binary assignment-matrix representation of P* (Definition 3), for which each row j is the *one-hot encoding* of element e_j's assignment, together with a scalar equal to its value $\boldsymbol{V}(P)$.

Definition 3. *The* binary assignment matrix *of a partial assignment $\langle B_1, \ldots, B_m \rangle$ over $\{e_1, \ldots, e_n\}$ is the $n \times m$ matrix $[x_{i,j}]$, where $x_{i,j} = 1$ if $e_j \in B_i$, and $x_{i,j} = 0$ otherwise.*

Recurrent Neural Network. Our second approximator consists of a RNN $\boldsymbol{r}_\theta(P)$ with parameters θ that approximates (1) as follows:

$$\boldsymbol{r}_\theta(P) \approx \boldsymbol{V}^*(P),$$

where $P = \langle B_1, \ldots, B_m \rangle$ is a partial assignment. It sequentially feeds triples $T_1, \ldots, T_{\|P\|}$ to a recurrent network with hidden states of dimension w. Each triple $T_i = \langle \alpha_i, \beta_i, \delta_i \rangle$ consists of:

- an element $\alpha_i \in \bigcup_{j=1}^m B_j$, with $\alpha_i \neq \alpha_k$ for $i \neq k$;
- the alternative $\beta_i \in A$ for which $\alpha_i \in B_{\beta_i}$; and
- a value δ_i, which is the *gain* (Definition 4) of assigning α_i to β_i over the partial assignment $\langle Q_1, \ldots, Q_m \rangle$, where

$$Q_j = B_j \setminus \{\alpha_i, \ldots, \alpha_{\|P\|}\}$$

for $j = 1, \ldots, m$. (So $\alpha_i, \ldots, \alpha_{\|P\|}$ are left "unassigned".)

The RNN's output is then processed through a parametric number of fully-connected layers with ReLU activation functions.

Definition 4. *The* gain *of assigning $e \in E$ to $a \in A$ over the partial assignment $P = \langle P_1, \ldots, P_m \rangle$ is defined as the value:*

$$\boldsymbol{V}\big(\langle P_1, \ldots, P_a \cup \{e\}, \ldots, P_m \rangle\big) - \boldsymbol{V}(P).$$

To generate a set of such triples from a partial assignment P, all one needs is an arbitrary permutation of the element set (which can be generated in $O(n)$), together with some basic, efficient computations to compute the triples' different gains. This approach thus allows us to make use of every element's precedent contribution. (Ideally, one would perhaps instead like to use all possible permutations of the element set and average over them—this is however, in general, an extremely costly computation, and involves what corresponds to computing all elements' different *Shapley values* [44].)

Gradient Boosting Decision Trees. Our third approximator consists of the gradient boosting decision trees method *XGBoost* [8]. XGBoost and similar methods have shown great promise on certain domains, in particular on tabular data [17, 18, 45], where these methods often outperform deep neural networks. XGBoost has also achieved state-of-the-art results on a large number of different machine learning challenges [8].

We use an *agent assignment vector* (Definition 5), together with a scalar value (in the same manner as for the FNN), as input for XGBoost. This input characterization made XGBoost perform much better than when using a binary assignment matrix in early experiments, which was not the case for the FNN.

Definition 5. *The* agent assignment vector *of a partial assignment $\langle B_1, \ldots, B_m \rangle$ over $\{e_1, \ldots, e_n\}$ is the n-sized vector $[x_j]$, where $x_j = i$ if $e_j \in B_i$, and $x_j = 0$ otherwise.*

Training Paradigm

Our training procedure incorporates generating a data set $\mathcal{D}$, split into training/validation sets (we use a 90%/10% split in our experiments), that consists of pairs $\langle P, \boldsymbol{V}^*(P)\rangle$, with every P being a size-m partial assignment over E. Each such pair's partial assignment is randomly chosen from $\Pi^m_{E_i}$, where $E_i \subset E$ is a uniformly drawn subset from $\{X \subseteq E : |X| = i\}$, for $i = n-1, \ldots, n-\kappa$, where $\kappa \in \{1, \ldots, n-1\}$ is a freely chosen training data parameter. In our experiments, $\mathcal{D}$ consists of exactly 10^4 such pairs for every i, so $|\mathcal{D}| = 10^4\kappa$. Note that it is only tractable to compute $\boldsymbol{V}^*$ if κ is kept small, since in such cases we only have to search a tree with depth κ and branching factor m to compute $\boldsymbol{V}^*$. For this reason, we constrain ourselves to $\kappa = 10$ in this work. We obtain the optimal values with the state-of-the-art optimal algorithm [33], which solves problems of small sizes (e.g., with $n \leq 10$) in milliseconds. This way of generating training data directly from the problem itself—by exploiting the problem's optimal substructure property—allows us to decide ourselves how much data we want to use for training, thus mitigating the machine learning data sparsity problem (i.e., the lack of sufficient training data).

For each function approximator $\boldsymbol{f}_\theta$'s parameters θ are then optimized over the training data to minimize:

$$\mathbb{E}_{\langle P, \boldsymbol{V}^*(P)\rangle \sim \mathcal{D}}\left[\left(\boldsymbol{V}^*(P) - \boldsymbol{f}_\theta(P)\right)^2\right].$$

5 EVALUATION AND EXPERIMENTS

The main goals with our benchmarks are to investigate how different input sizes, value distributions, and function approximators affect our heuristic generation method, and how the generated heuristics compare to the state of the art. To accomplish these goals, and in accordance with established practices in combinatorial optimization, we benchmark our method with standardized problem distributions that generate difficult problem instances for combinatorial assignment and weighted partitioning. We also introduce three application-focused benchmarks based on the three major domains for combinatorial assignment that we outlined in Section 2.

The first five distributions that we use for benchmarking are *NPD* (normal probability distribution), *UPD* (uniform probability distribution), *SNPD* (sparse NPD), *SUPD* (sparse UPD), and *NRD* (normal relational distribution). These generate some of the most difficult known problems for the state-of-the-art UCA solvers (see e.g., [31, 33]). Variations of them have also been used extensively to benchmark various approaches for solving a number of different problems related to UCA in operations research, algorithmic game theory, and multi-agent systems, including coalition formation and combinatorial auctions [27, 31–34, 37, 41, 50]. In more detail, they generate problem instances as follows:

- **NPD:** $\boldsymbol{v}(B,a) \sim \boldsymbol{\mathcal{N}}(\mu_1, \sigma^2)$;
- **UPD:** $\boldsymbol{v}(B,a) \sim \boldsymbol{\mathcal{U}}(0,1)$;
- **SNPD:** $\boldsymbol{v}(B,a) \sim \boldsymbol{\mathcal{N}}(\mu_1, \sigma^2)$ with probability 0.01, otherwise draw $\boldsymbol{v}(B,a) \sim \boldsymbol{\mathcal{N}}(\mu_2, \sigma^2)$;
- **SUPD:** $\boldsymbol{v}(B,a) \sim \boldsymbol{\mathcal{U}}(0,1)$ with probability 0.01, otherwise draw $\boldsymbol{v}(B,a) \sim \boldsymbol{\mathcal{U}}(0,0.1)$; and
- **NRD:** $\boldsymbol{v}(B,a) = \sum_{\{e_1,e_2\}\in\binom{B}{2}} \boldsymbol{r}\big(\{e_1,e_2\},a\big)$;

for all $B \subseteq E$, and $a \in A$, where $\sigma = 0.1$, $\mu_1 = 1$, $\mu_2 = 0.1$, and $\boldsymbol{r}\big(\{e_1,e_2\},a\big) \sim \boldsymbol{\mathcal{N}}(\mu_r, \sigma_r^2)$, with $\mu_r = 0$, $\sigma_r = 0.1$, for all $e \in E$, $\{e_1,e_2\} \in \binom{E}{2}$, and $a \in A$. Note that, out of these distributions, NRD is the only one that generates problem instances that can be represented in a size that is polynomial in the number of elements. More precisely, this requires $O(mn^2)$ memory per problem instance. Also, the class of problems with value functions that are representable concisely in this form is NP-hard [9]. Intuitively, if the elements are viewed as agents, and the alternatives as tasks, an interpretation of $\boldsymbol{r}\big(\{e_1,e_2\},a\big)$ is that it represents how well the agent $e_1 \in E$ can collaborate with $e_2 \in E$ when performing task $a \in A$; NRD simply assumes that this relation is normally distributed.

To make our experiments as exhaustive as possible, and to show that our approach works for several different types of applications, we also introduce three additional difficult distributions that emulate the GAP, the WDP, and the CSG problem (that we discussed thoroughly in Section 2). As a corollary, they illustrate how some important variations of these problems are related and reduces to each other via the UCA problem's value function. We define these distributions, denoted *GAPU* (GAP uniform), *WDPR* (WDP random), and *CSGU* (CSG uniform), as follows:

- **GAPU:** $\boldsymbol{v}(B,a) = \sum_{e\in B}\big\{\boldsymbol{p}(e,a)\big\}$ if $\sum_{e\in B}\big\{\boldsymbol{w}(e,a)\big\} \leq \boldsymbol{c}(a)$, $\boldsymbol{v}(B,a) = 0$ otherwise (exceeds capacity);
- **WDPR:** $\boldsymbol{v}(B,a) = \boldsymbol{b}(B,a)$; and
- **CSGU:** $\boldsymbol{v}(B,a) = \boldsymbol{u}(B)$;

for all $B \subseteq E$, and $a \in A$, where:

- (Capacity) $\boldsymbol{c}(a) \sim \boldsymbol{\mathcal{U}}(0, \frac{1}{m})$;
- (Profit) $\boldsymbol{p}(e,a) \sim \boldsymbol{\mathcal{U}}(0,1)$;
- (Weight) $\boldsymbol{w}(e,a) \sim \boldsymbol{\mathcal{U}}(0, \frac{1}{n})$;
- (Bid price) $\boldsymbol{b}(B,a) \sim \boldsymbol{\mathcal{U}}(0,1)$ for 1000 (uniformly) random bundle-to-alternative draws (representing 1000 bids), else $\boldsymbol{v}(B,a) = 0$ (representing that the bid was not given); and
- (Alternative invariance) $\boldsymbol{u}(B) \sim \boldsymbol{\mathcal{U}}(0,1)$;

for all $e \in E$, $B \subseteq E$, and $a \in A$. Problem instances generated with GAPU and WDPR can be stored compactly—we do not need an exponential number of values in the number of elements to define them—and they correspond to highly difficult GAP and WDP instances that allow *free disposal* (i.e., elements can freely be thrown away). Note that WDPR is equivalent to the problem benchmark called *random* that was first proposed in [39] for combinatorial auctions. Finally, CSGU follows the reduction given in [31], thus corresponding to one of the more difficult CSG benchmark distributions proposed by [21], which was subsequently used in e.g., [35, 37] to benchmark some of the current state-of-the-art CSG algorithms.

Benchmark Setup

We generate a number of new test sets used solely for benchmarking. For each problem, we generate test sets $\mathcal{T}_i$, consisting of 20 random partial assignments over E with i unassigned elements, for $i = 1, \ldots, n-1$. This enables us to benchmark how well our learned heuristic functions generalize to predicting the gain of unseen partial assignments, for which some even have fewer assigned elements than those that exist in the training/validation sets.

Note that while it would be interesting to gauge our learned approximators' performances against other heuristics when the optimal cannot be obtained, this is not possible to do without first integrating the heuristics in a full-fledged solver. The reason for this is that there is no value to compare the heuristics' prediction quality

to unless the ground truth can be computed (which is not possible for large n). In other words, in such cases we cannot know which heuristic is best unless we use them to find a "complete" combinatorial assignment first. This would naturally require benchmarking the different heuristics with various algorithms, which we hope to do as future work. Also, for the same reason, it is at the moment not possible to benchmark against industry-grade solvers like for example Gurobi and CPLEX (which have already been shown to perform subpar compared to the state-of-the-art algorithm [33]). Note that another reason to not benchmark heuristics by integrating them in a solver is to avoid *algorithm bias*, i.e., the phenomenon that certain heuristics perform better with some algorithms—a well-known problem in heuristic design.

The heuristics that we benchmark against are based on the state-of-the-art UCA heuristics [31], namely a *local search heuristic* (abbreviated *LS*), and a hybrid *local search/greedy heuristic* (*LSG* for short). Both of them are myopic, and they outperform more advanced heuristics in all previously tried benchmarks. We also introduce and use two baseline (naïve) heuristics: *PV*, which uses a partial solution's value as an approximation for (1), and *RR*, which assigns unassigned elements randomly and uses the value of the generated solution for the same purpose. Monte Carlo methods are often based on similar approaches. The worst-case execution times for single LS, LSG, PV, and RR predictions are $O\big(mn\boldsymbol{g}(v)\big)$, $O\big(mn\boldsymbol{g}(v)\big)$, $O(m)$, $O(n+m)$, respectively. The function $\boldsymbol{g}(v)$ represents the number of local improvements that are required for LS and LSG to reach local optima. In worst-case, $\boldsymbol{g}(v) \in O(m^n)$. However, in practice, $\boldsymbol{g}(v)$ is typically small. FNN, RNN, and XGBoost generate predictions in time $O(mn)$, $O(n)$, and $O(n)$, respectively. Moreover, our feedforward and recurrent neural networks, and the XGBoost model, are denoted FNN, RNN, and XGB in the graphs, respectively.

We re-train the function approximators for every new problem distribution. The approximators' hyperparameters are optimized using *Hyperopt* [4] and 100 evaluations per problem instance. For XGBoost we use the same hyperparameter search space as the one used in [46]. A batch size of 32 was used for the neural networks, together with 100 epochs, and finding their hyperparameters took us 4 days, running 44 experiments concurrently. Our hyperparameter search space together with the final hyperparameters are presented in this paper's Appendix.

The result of each experiment was produced by evaluating all heuristics on the different test sets' partial assignments. To make sure that the heuristics competed on a similar computation budget, we allowed RR, LS, and LSG to run with random restarts until they had finished a number of iterations and consumed at least as much computation time as the FNN, the most promising network. The highest-valued prediction of these runs was then chosen as the final prediction. PV, the FNN/RNN, and XGB always make the same predictions for a given partial assignment, and thus cannot be used with random restarts in this fashion. Furthermore, unlike the other heuristics [31], their predictions are not a lower bound on the optimal value, and choosing the highest-valued prediction would therefore not be motivated.

The final result of each experiment was computed by taking the average of the resulting values from 10 runs. We plot the 95% confidence interval in all graphs. The neural networks were trained and benchmarked with the *PyTorch 1.4.0* library (*Python 3.6*), while the other methods were implemented in *C++17*. All values for the synthetic problem sets were generated with methods from the *C++ Standard Library*. The *gcc 9.3.0* compiler was used to compile all C++ code. The training and hyperparameter optimization ran on two Nvidia RTX 2080 Ti GPUs, while the experiments were conducted with an AMD 3950X 3.5GHz CPU, and 32GB memory. Our networks were evaluated on the CPU to compete on equal terms when computing run times.

Results

Figure 1 shows the results of our main benchmarks. In these benchmarks, we computed the mean squared error of the heuristics' predictions to the ground truth on the different test sets. First, we can see a clear general trend that two of our learned heuristics, namely XGB and the FNN, outperform all other heuristics. This indicates that their execution times are motivated, since they provide significantly higher-quality predictions in equal or lower time than their competition. A notable exception is NRD, for which the state-of-the-art conventional heuristics outperform the learned ones, and CSGU, for which there is almost a tie. An important observation is that there is no distribution for which the FNN and XGB perform much worse than the state-of-the-art conventional heuristics—indicating that our heuristic generation paradigm is potentially suitable for many different combinatorial assignment problems, including those that can be represented concisely, such as the WDP and the GAP. Moreover, the sparse distributions are clearly much more difficult for all heuristics, which is not surprising due to their sparsity of high-valued bundles. A potential way to mitigate this for the learned heuristics is by calibrating their training data so that its partial assignments contain more high-valued bundles. Finally, we sum and take the average of all the mean squared errors for each benchmark in Table 1, where we see that the FNN and XGB greatly outperform their competition in a majority of the tests, most notably on problem instances generated with NPD, SNPD, and WDPR. LS and LSG however outperform our learned heuristics on NRD and CSGU.

When the number of unassigned elements is similar to those used in the training data, the RNN exhibits similar performance as the FNN. However, outside this manifold, the RNN's predictions quickly diverge from the ground truth. Seen from another perspective, the FNN generalizes surprisingly well outside of the training data distribution, while the RNN does not. This indicates that our matrix representation is more suitable (better) for our experiments than the RNN's learned representation. A potential reason for this is that UCA is inherently invariant to the order for which the elements are assigned to an alternative, making it necessary for the RNN to learn order invariance to generalize well, effectively making the sequential nature of the RNN redundant at best.

Furthermore, our results show that the time required for optimally solving a problem far exceeds generating our training sets. We can see this by e.g., looking at the experiments that we ran on the GAPU distribution for $n = 25$—here, our generated heuristics greatly outperform the conventional heuristics, while the best optimal solvers are not able to solve such instances in feasible time.

Finally, to gauge the heuristics' efficiency, we recorded their execution time for different distributions and numbers of unassigned

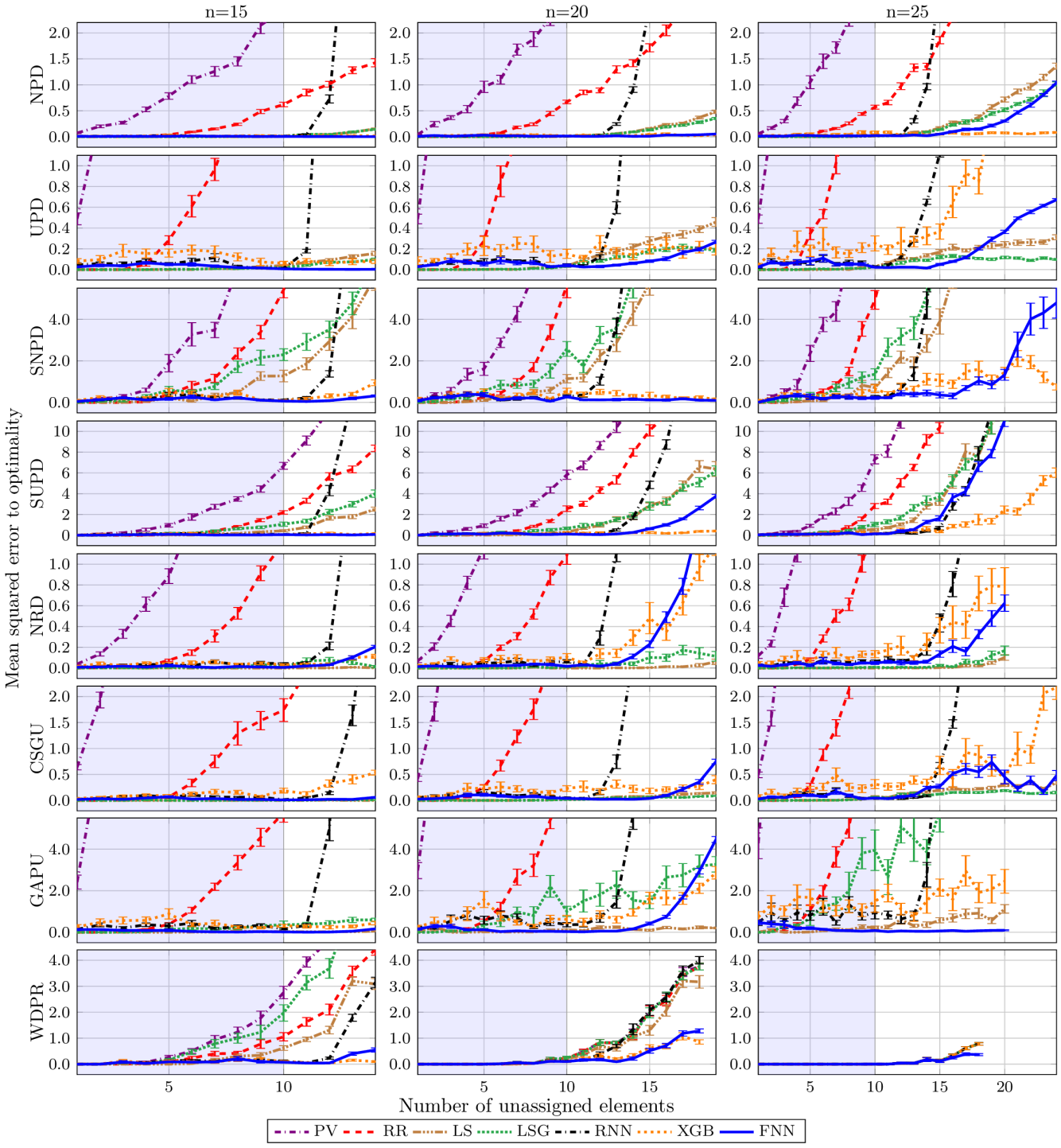

Figure 1: Mean squared error of heuristic predictions to optimality against 1 to $n-1$ unassigned elements for different problems with 10 alternatives. A lower value is better. An optimal heuristic has value zero and always yields an optimal solution if followed greedily. The columns represent tests with $n = 15$ (left), $n = 20$ (middle), and $n = 25$ (right), and the rows represent different problem instance distributions. A low mean squared error to the right of $\kappa = 10$ (the shaded areas) shows generalization outside of the training distribution. The LSG heuristic is the current state of the art. (Note that for some distributions, we only use partial assignments with fewer unassigned elements than that of the value of n, making the graphs appear incomplete. Our training paradigm works for much larger n, but the state-of-the-art solver's computational cost for optimally solving partial assignments past this point becomes prohibitively high, thus making it difficult for us to compute a mean squared error to optimality against ground-truth values.)

Table 1: Average (aggregated) mean squared error for all predictions to optimality for the various benchmarks. A lower value is better. The best value for each benchmark is marked in bold.

	NPD			UPD			SNPD			SUPD			NRD			CSGU			GAPU			WDPR		
n =	15	20	25	15	20	25	15	20	25	15	20	25	15	20	25	15	20	25	15	20	25	15	20	25
PV	1.61	2.86	4.33	10.3	20.60	22.39	8.29	17.38	33.69	5.04	8.32	18.80	3.13	6.84	9.50	26.93	35.41	37.10	67.87	130.20	173.04	2.30	0.88	0.17
LS	0.024	0.09	0.29	**0.03**	0.13	0.12	1.34	3.06	6.39	0.56	1.56	5.48	**0.001**	**0.007**	**0.011**	**0.001**	0.04	0.13	**0.01**	**0.12**	0.36	0.71	0.71	0.12
LSG	0.026	0.07	0.22	**0.03**	**0.08**	**0.06**	1.90	3.79	6.31	1.00	1.57	5.51	0.02	0.042	0.025	0.004	**0.03**	**0.09**	0.24	1.39	4.60	1.74	0.86	0.17
RR	0.44	0.95	1.67	1.34	2.97	3.35	5.06	10.32	22.30	2.06	5.40	12.44	1.03	2.78	3.79	1.28	3.08	4.10	4.77	10.97	18.15	1.05	0.85	0.12
RNN	2.23	2.30	13.90	3.34	8.91	5.65	3.13	13.07	29.07	2.58	5.03	8.82	0.52	4.50	1.31	0.53	7.29	8.64	8.68	22.35	18.94	0.44	0.84	0.12
FNN	**0.007**	**0.02**	0.17	**0.03**	**0.08**	0.17	**0.15**	**0.18**	1.01	**0.04**	0.59	4.70	0.03	0.27	0.13	0.03	0.12	0.23	0.06	0.62	**0.14**	0.13	**0.26**	**0.08**
XGB	0.017	0.03	**0.06**	0.11	0.18	0.73	0.32	0.33	**0.83**	0.11	**0.20**	**1.17**	0.05	0.26	0.26	0.18	0.21	0.57	0.40	1.02	1.55	**0.09**	**0.26**	0.12

elements. Results for our largest problems with $n = 25$ are shown in Figure 2. Note that PV is not included, as its execution time is negligible. The networks are slower than the other heuristics by roughly one or two orders of magnitude. This was expected since all our experiments were made without any attempts to improve the computational efficiency of the networks—for example by minimizing the networks' sizes, or removing redundant nodes. We also see that the execution times for RR, LS, and LSG increase with the number of unassigned elements. The RNN exhibits the inverse behaviour, since fewer assigned elements results in a shorter input sequence, while the FNN is unaffected. Furthermore, the neural networks differ between distributions due to changing hyperparameters. Our results were almost identical for the smaller (easier) problems.

6 SUMMARY AND CONCLUSIONS

We developed a general-purpose learning-based method leveraged by generating and optimally solving subproblems to produce heuristics that can be used for solving many combinatorial assignment problems. We ran benchmarks on a variety of different problem distributions, and showed that the heuristics our method generates outperform the state of the art. Our experiments also show that our training paradigm manages to make a neural network and a state-of-the-art ensemble method (i.e., XGBoost) generalize surprisingly well outside of the training set distribution. In addition, our method is highly scalable. It can for example be used on problem instances much larger than those that are used in our experiments, and it can run on relatively cheap hardware. It is also compatible with a large number of important problems, such as the generalized assignment problem, coalition structure generation, and the combinatorial auction winner determination problem.

For future work, we would like to investigate other machine learning methods, approximators, and architectures, including more specialized ones. An open question is what search algorithms our heuristics are best coupled with to solve full combinatorial assignment problems. Also, while there are synthetic problem sets that can be used for benchmarking like the standardized ones that we used, and for example the synthetic tests for combinatorial auctions provided by [27], there is a lack of real-world data sets that have been published. While [32] benchmarked their solver on a real-world strategy game, the data sets that they used are proprietary and not openly available. We therefore believe that it is important to publish real-world instances that the community can use for benchmarking with the aim to improve our understanding of the challenges that real-world combinatorial assignment problems entail. Finally, it would be interesting to apply our method to related combinatorial assignment problems, e.g., competitive instances such as [30].

ACKNOWLEDGMENTS

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and by grants from the National Graduate School in Computer Science (CUGS), Sweden, the Excellence Center at Linköping-Lund for Information Technology (ELLIIT), the TAILOR Project funded by EU Horizon 2020 research and innovation programme GA No 952215, and Knut and Alice Wallenberg Foundation (KAW 2019.0350).

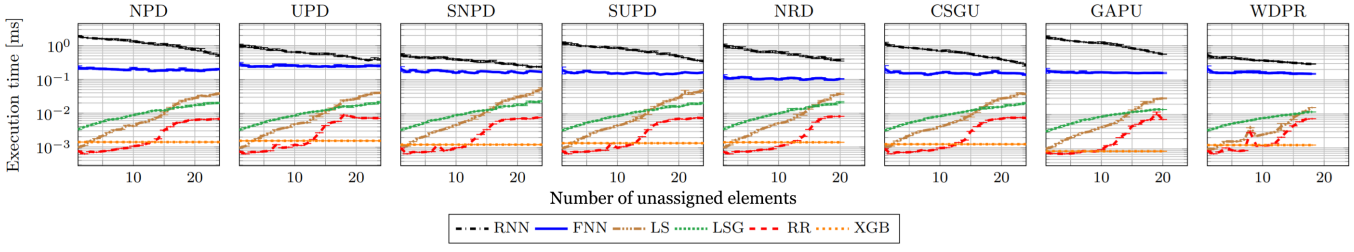

Figure 2: Time (in milliseconds) required to evaluate each heuristic on a single partial assignment with varying numbers of unassigned elements on problems with $n = 25$ and $m = 10$. PV constantly takes approximately 10^{-4} milliseconds.

REFERENCES

[1] Haris Aziz, Hau Chan, Ágnes Cseh, Bo Li, Fahimeh Ramezani, and Chenhao Wang. 2021. Multi-Robot Task Allocation-Complexity and Approximation. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

[2] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. 2021. Machine Learning for Combinatorial Optimization: a Methodological Tour D'horizon. *European Journal of Operational Research (EJOR)* 290, 2 (2021), 405–421.

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum Learning. In *International Conference on Machine Learning (ICML)*.

[4] James Bergstra, Dan Yamins, David D Cox, et al. 2013. Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms. In *Python in Science Conference (SciPy)*.

[5] Dalila Boughaci. 2013. Metaheuristic Approaches for the Winner Determination Problem in Combinatorial Auction. In *Artificial Intelligence, Evolutionary Computing and Metaheuristics (AIECM)*. Springer.

[6] Dirk G Cattrysse and Luk N Van Wassenhove. 1992. A Survey of Algorithms for the Generalized Assignment Problem. *European Journal of Operational Research (EJOR)* 60, 3 (1992).

[7] Omar Cheikhrouhou and Ines Khoufi. 2021. A comprehensive Survey on the Multiple Traveling Salesman Problem: Applications, Approaches and Taxonomy. *Computer Science Review* 40 (2021), 100369.

[8] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable Tree Boosting System. In *International Conference on Knowledge Discovery and Data Mining (KDD)*.

[9] Yann Chevaleyre, Ulle Endriss, Sylvia Estivie, and Nicolas Maudet. 2008. Multiagent Resource Allocation in k-Additive Domains: Preference Representation and Complexity. *Annals of Operations Research* 163, 1 (2008), 49–62.

[10] Viet Dung Dang, Rajdeep K Dash, Alex Rogers, and Nicholas R Jennings. 2006. Overlapping Coalition Formation for Efficient Data Fusion in Multi-Sensor Networks. In *AAAI Conference on Artificial Intelligence (AAAI)*.

[11] Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. 2018. Learning Heuristics for the Tsp by Policy Gradient. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*. Springer.

[12] Maxime Gasse, Didier Chetelat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. 2019. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[13] Johan Håstad. 1999. Clique is Hard to Approximate Within 1- ε. *Acta Mathematica* 182, 1 (1999), 105–142.

[14] Elad Hazan, Shmuel Safra, and Oded Schwartz. 2006. On the Complexity of Approximating k-Set Packing. *Computational Complexity* 15 (2006).

[15] He He, Hal Daume III, and Jason M Eisner. 2014. Learning to Search in Branch and Bound Algorithms. *Advances in Neural Information Processing Systems (NeurIPS)* 27 (2014), 3293–3301.

[16] André Hottung, Shunji Tanaka, and Kevin Tierney. 2020. Deep Learning Assisted Heuristic Tree Search for the Container Pre-marshalling Problem. *Computers & Operations Research* 113 (2020).

[17] Arlind Kadra, Marius Lindauer, Frank Hutter, and Josif Grabocka. 2021. Regularization Is All You Need: Simple Neural Nets can Excel on Tabular Data. *arXiv preprint arXiv:2106.11189* (2021).

[18] Liran Katzir, Gal Elidan, and Ran El-Yaniv. 2020. Net-DNF: Effective Deep Modeling of Tabular Data. In *International Conference on Learning Representations (ICLR)*.

[19] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning Combinatorial Optimization Algorithms Over Graphs. *Advances in Neural Information Processing Systems (NeurIPS)*.

[20] Wouter Kool, Herke van Hoof, and Max Welling. 2018. Attention, Learn to Solve Routing Problems!. In *International Conference on Learning Representations (ICLR)*.

[21] Kate S Larson and Tuomas W Sandholm. 2000. Anytime Coalition Structure Generation: An Average Case Study. *Journal of Experimental & Theoretical Artificial Intelligence (JETAI)* 12, 1 (2000), 23–42.

[22] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521, 7553 (2015).

[23] Mengyuan Lee, Seyyedali Hosseinalipour, Christopher Greg Brinton, Guanding Yu, and Huaiyu Dai. 2020. A Fast Graph Neural Network-Based Method for Winner Determination in Multi-Unit Combinatorial Auctions. *IEEE Transactions on Cloud Computing (TCC)* (2020).

[24] Mengyuan Lee, Guanding Yu, and Geoffrey Ye Li. 2019. Learning To Branch: Accelerating Resource Allocation in Wireless Networks. *IEEE Transactions on Vehicular Technology (TVT)* 69, 1 (2019), 958–970.

[25] Daniel Lehmann, Rudolf Müller, and Tuomas Sandholm. 2006. The Winner Determination Problem. *Combinatorial Auctions* (2006).

[26] Daniel Lehmann, Liadan Ita Oćallaghan, and Yoav Shoham. 2002. Truth Revelation in Approximately Efficient Combinatorial Auctions. *Journal of the ACM (JACM)* 49, 5 (2002), 577–602.

[27] Kevin Leyton-Brown and Yoav Shoham. 2006. A Test Suite for Combinatorial Auctions. *Combinatorial Auctions* 18 (2006), 451–478.

[28] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. 2018. Combinatorial Optimization With Graph Convolutional Networks and Guided Tree Search. *Advances in Neural Information Processing Systems (NeurIPS)*.

[29] Neil Newman, Alexandre Fréchette, and Kevin Leyton-Brown. 2017. Deep Optimization for Spectrum Repacking. *Commun. ACM* 61, 1 (2017), 97–104.

[30] Abraham Othman, Tuomas Sandholm, and Eric Budish. 2010. Finding Approximate Competitive Equilibria: Efficient and Fair Course Allocation. In *International Conference On Autonomous Agents and Multi-Agent Systems (AAMAS)*.

[31] Fredrik Präntare, Herman Appelgren, and Fredrik Heintz. 2021. Anytime Heuristic and Monte Carlo Methods for Large-Scale Simultaneous Coalition Structure Generation and Assignment. In *AAAI Conference on Artificial Intelligence (AAAI)*.

[32] Fredrik Präntare and Fredrik Heintz. 2020. An Anytime Algorithm for Optimal Simultaneous Coalition Structure Generation and Assignment. *Autonomous Agents and Multi-Agent Systems (JAAMAS)* 34, 1 (2020).

[33] Fredrik Präntare and Fredrik Heintz. 2020. Hybrid Dynamic Programming for-Simultaneous Coalition Structure Generation and Assignment. In *International Conference on Principles and Practice of Multi-Agent Systems (PRIMA)*.

[34] Talal Rahwan, Tomasz Michalak, Michael Wooldridge, and Nicholas R Jennings. 2012. Anytime Coalition Structure Generation in Multi-Agent Systems with Positive or Negative Externalities. *Artificial Intelligence (AIJ)* 186 (2012), 95–122.

[35] Talal Rahwan, Tomasz P Michalak, and Nicholas R Jennings. 2012. A Hybrid Algorithm for Coalition Structure Generation. In *AAAI Conference on Artificial Intelligence (AAAI)*.

[36] Talal Rahwan, Tomasz P Michalak, Michael Wooldridge, and Nicholas R Jennings. 2015. Coalition Structure Generation: A Survey. *Artificial Intelligence* (2015).

[37] Talal Rahwan, Sarvapali D Ramchurn, Nicholas R Jennings, and Andrea Giovannucci. 2009. An Anytime Algorithm for Optimal Coalition Structure Generation. *Journal of Artificial Intelligence Research (JAIR)* 34 (2009), 521–567.

[38] John R Rice. 1976. The Algorithm Selection Problem. In *Advances in Computers*. Vol. 15. Elsevier.

[39] Tuomas Sandholm. 2002. Algorithm for Optimal Winner Determination in Combinatorial Auctions. *Artificial Intelligence (AIJ)* 135, 1-2 (2002).

[40] Tuomas Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. 2002. Winner Determination in Combinatorial Auction Generalizations. In *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

[41] Tuomas Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. 2005. CABOB: A Fast Optimal Algorithm for Winner Determination in Combinatorial Auctions. *Management Science* 51, 3 (2005), 374–390.

[42] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. 2020. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *Nature* 588, 7839 (2020), 604–609.

[43] Daniel Selsam, Matthew Lamm, B Benedikt, Percy Liang, Leonardo de Moura, David L Dill, et al. 2018. Learning a SAT Solver from Single-Bit Supervision. In *International Conference on Learning Representations (ICLR)*.

[44] Lloyd S Shapley. 1953. A Value for n-person Games. *Contributions to the Theory of Games (AM-28)* 2, 28 (1953), 307–317.

[45] Ira Shavitt and Eran Segal. 2018. Regularization Learning Networks: Deep Learning for Tabular Datasets. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[46] Ravid Shwartz-Ziv and Amitai Armon. 2022. Tabular Data: Deep Learning is Not All You Need. *Information Fusion* 81 (2022), 84–90.

[47] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the Game of Go With Deep Neural Networks and Tree Search. *Nature* 529, 7587 (2016).

[48] Yuan Sun, Andreas Ernst, Xiaodong Li, and Jake Weiner. 2021. Generalization of Machine Learning for Problem Reduction: A Case Study on Travelling Salesman Problems. *OR Spectrum (ORSP)* 43, 3 (2021), 607–633.

[49] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer Networks. *Advances in Neural Information Processing Systems (NeurIPS)*.

[50] Feng Wu and Sarvapali D Ramchurn. 2020. Monte-Carlo Tree Search for Scalable Coalition Formation. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

[51] Qinghua Wu and Jin-Kao Hao. 2015. Solving the Winner Determination Problem via a Weighted Maximum Clique Heuristic. *Expert Systems with Applications* 42, 1 (2015).

[52] Junchi Yan, Shuang Yang, and Edwin R Hancock. 2020. Learning for Graph Matching and Related Combinatorial Optimization Problems. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

[53] ChiaWei Yeh and Toshiharu Sugawara. 2016. Solving Coalition Structure Generation Problem With Double-Layered Ant Colony Optimization. In *IIAI International Congress on Advanced Applied Informatics (IIAI AAI)*.

[54] Emre Yolcu and Barnabás Póczos. 2019. Learning Local Search Heuristics for Boolean Satisfiability. *Advances in Neural Information Processing Systems (NeurIPS)*.

[55] David Zuckerman. 2006. Linear Degree Extractors and the Inapproximability of Max Clique and Chromatic Number. In *ACM Symposium on Theory of Computing (STOC)*.