Combining Reinforcement Learning and Constraint Programming for Combinatorial Optimization

Quentin Cappart,[1] **Thierry Moisan,**[2] **Louis-Martin Rousseau**[1]**,**
Isabeau Prémont-Schwarz[2]**, Andre Cire**[3]
[1,2] Ecole Polytechnique de Montréal, Montreal, Canada
[2] Element AI, Montreal, Canada
[3] University of Toronto Scarborough, Toronto, Canada
{quentin.cappart, louis-martin.rousseau}@polymtl.ca
thierry.moisan@elementai.com
isabeau@cryptolab.net
andre.cire@utoronto.ca

Abstract

Combinatorial optimization has found applications in numerous fields, from aerospace to transportation planning and economics. The goal is to find an optimal solution among a finite set of possibilities. The well-known challenge one faces with combinatorial optimization is the state-space explosion problem: the number of possibilities grows exponentially with the problem size, which makes solving intractable for large problems. In the last years, deep reinforcement learning (DRL) has shown its promise for designing good heuristics dedicated to solve NP-hard combinatorial optimization problems. However, current approaches have an important shortcoming: they only provide an approximate solution with no systematic ways to improve it or to prove optimality. In another context, constraint programming (CP) is a generic tool to solve combinatorial optimization problems. Based on a complete search procedure, it will always find the optimal solution if we allow an execution time large enough. A critical design choice, that makes CP non-trivial to use in practice, is the branching decision, directing how the search space is explored. In this work, we propose a general and hybrid approach, based on DRL and CP, for solving combinatorial optimization problems. The core of our approach is based on a dynamic programming formulation, that acts as a bridge between both techniques. We experimentally show that our solver is efficient to solve three challenging problems: the traveling salesman problem with time windows, the 4-moments portfolio optimization problem, and the 0-1 knapsack problem. Results obtained show that the framework introduced outperforms the stand-alone RL and CP solutions, while being competitive with industrial solvers.

Introduction

The design of efficient algorithms for solving NP-hard problems, such as *combinatorial optimization problems* (COPs), has long been an active field of research (Wolsey and Nemhauser 1999). Broadly speaking, there exist two main families of approaches for solving COPs, each of them having pros and cons. On the one hand, *exact algorithms* are based on a complete and clever enumeration of the solutions space (Lawler and Wood 1966; Rossi, Van Beek, and Walsh 2006). Such algorithms will eventually find the optimal solution, but they may be prohibitive for solving large instances because of the exponential increase of the execution time. That being said, well-designed exact algorithms can nevertheless be used to obtain sub-optimal solutions by interrupting the search before its termination. This flexibility makes exact methods appealing and practical, and as such they constitute the core of modern optimization solvers as CPLEX (Cplex 2009), Gurobi (Optimization 2014), or Gecode (Team 2008). It is the case of *constraint programming* (CP) (Rossi, Van Beek, and Walsh 2006), which has the additional asset to be a generic tool that can be used to solve a large variety of COPs, whereas *mixed integer programming* (MIP) (Bénichou et al. 1971) solvers only deal with linear problems and limited non-linear cases. A critical design choice in CP is the *branching strategy*, i.e., directing how the search space must be explored. Naturally, well-designed heuristics are more likely to discover promising solutions, whereas bad heuristics may bring the search into a fruitless subpart of the solution space. In general, the choice of an appropriate branching strategy is non-trivial and their design is a hot topic in the CP community (Palmieri, Régin, and Schaus 2016; Fages and Prud'Homme 2017; Laborie 2018).

On the other hand, *heuristic algorithms* (Aarts and Lenstra 2003; Gendreau and Potvin 2005) are incomplete methods that can compute solutions efficiently, but are not able to prove the optimality of a solution. They also often require substantial problem-specific knowledge for building them. In the last years, *deep reinforcement learning* (DRL) (Sutton, Barto et al. 1998; Arulkumaran et al. 2017) has shown its promise to obtain high-quality approximate solutions to some NP-hard COPs (Bello et al. 2016; Khalil et al. 2017; Deudon et al. 2018; Kool, Van Hoof, and Welling 2018). Once a model has been trained, the execution time is typically negligible in practice. The good results obtained suggest that DRL is a promising new tool for finding efficiently good approximate solutions to NP-hard problems, provided that (1) we know the distribution of problem instances and (2) that we have enough instances sampled from this distribution for training the model. Nonetheless, current methods have shortcomings. Firstly, they are mainly dedicated to solve a specific problem, as the *travelling salesman problem* (TSP), with the noteworthy exception of (Khalil et al. 2017) that tackle three other graph-based problems, and of (Kool, Van Hoof, and Welling 2018) that target routing problems. Secondly, they are only designed to act as a constructive heuristic, and come with no systematic ways to improve the solutions obtained, unlike complete methods, such as CP.

As both exact approaches and learning-based heuristics have strengths and weaknesses, a natural question arises: *How can we leverage these strengths together in order to build a better tool to solve combinatorial optimization problems ?* In this work, we show that it can be successfully done by the combination of reinforcement learning and constraint programming, using dynamic programming as a bridge between both techniques. *Dynamic programming* (DP) (Bellman 1966), which has found successful applications in many fields (Godfrey and Powell 2002; Topaloglou, Vladimirou, and Zenios 2008; Tang, Mu, and He 2017; Ghasempour and Heydecker 2019), is an important technique for modelling COPs. In its simplest form, DP consists in breaking a problem into sub-problems that are linked together through a recursive formulation (i.e., the well-known Bellman equation). The main issue with exact DP is the so-called *curse of dimensionality*: the number of generated sub-problems grows exponentially, to the point that it becomes infeasible to store all of them in memory.

This paper proposes a generic and complete solver, based on DRL and CP, in order to solve COPs that can be modelled using DP. Our detailed contributions are as follows: (1) A new encoding able to express a DP model of a COP into a RL environment and a CP model; (2) The use of two standard RL training procedures, *deep Q-learning* and *proximal policy optimization*, for learning an appropriate CP branching strategy. The training is done using randomly generated instances sampled from a similar distribution to those we want to solve; (3) The integration of the learned branching strategies on three CP search strategies, namely *branch-and-bound*, *iterative limited discrepancy search* and *restart based search*; (4) Promising results on three challenging COPs, namely the *travelling salesman problem with time windows*, the *4-moments portfolio optimization*, the *0-1 knapsack problem*; (5) The open-source release of our code and models, in order to ease the future research in this field[1].

In general, as there are no underlying hypothesis such as linearity or convexity, a DP cannot be trivially encoded and solved by standard integer programming techniques (Bergman and Cire 2018). It is one of the reasons that drove us to consider CP for the encoding. The next section presents the hybrid solving process that we designed. Then, experiments on the two case studies are carried out. Finally, a discussion on the current limitations of the approach and the next research opportunities are proposed.

A Unifying Representation Combining Learning and Searching

Because of the state-space explosion, solving NP-hard COPs remains a challenge. In this paper, we propose a generic and complete solver, based on DRL and CP, in order to solve COPs that can be modelled using DP. This section describes the complete architecture of the framework we propose. A high-level picture of the architecture is shown in Figure 1. It is divided into three parts: the *learning phase*, the *solving phase* and the *unifying representation*, acting as a bridge between the two phases. Each part contains several components. Green blocks and arrows represent the original contributions of this work and blue blocks corresponds to known algorithms that we adapted for our framework.

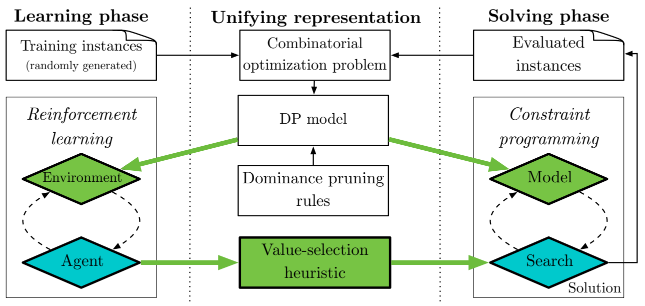

Figure 1: Overview of our framework for solving COPs.

Dynamic Programming Model

Dynamic programming (DP) (Bellman 1966) is a technique combining both mathematical modeling and computer programming for solving complex optimization problems, such as NP-hard problems. In its simplest form, it consists in breaking a problem into sub-problems and to link them through a *recursive formulation*. The initial problem is then solved recursively, and the optimal values of the decision variables are recovered successively by tracking back the information already computed. Let us consider a general COP $\mathcal{Q} : \{\max f(x) : x \in X \subseteq \mathbb{Z}^n\}$, where x_i with $i \in \{1..n\}$ are n discrete variables that must be assigned in order to maximize a function $f(x)$. In the DP terminology, and assuming a fixed-variable ordering where a decision has to been taken at each stage, the decision variables of $\mathcal{Q}$ are referred to as the *controls* (x_i). They take value from their *domain* $D(x_i)$, and enforce a *transition* $(T : S \times X \to S)$ from a *state* (s_i) to another one (s_{i+1}) where S is the set of states. The initial state (s_1) is known and a transition is done at each *stage* $(i \in \{1, \dots, n\})$ until all the variables have been assigned. Besides, a *reward* $(R : S \times X \to \mathbb{R})$ is induced after each transition. Finally, a DP model can also contain *validity conditions* $(V : S \times X \to \{0, 1\})$ and *dominance rules* $(P : S \times X \to \{0, 1\})$ that restrict the set of feasible actions. The difference between both is that validity conditions are mandatory to ensure the correctness of the DP model $(V(s, x) = 0 \Leftrightarrow T(s, x) = \bot)$ whereas the dominance rules are only used for efficiency purposes $(P(s, x) = 0 \Rightarrow T(s, x) = \bot)$, where $\Leftrightarrow$, $\Rightarrow$, and $\bot$ represent the equivalence, the implication, and the unfeasible state, respectively. A DP model for a COP can then be modelled as a tuple $\langle S, X, T, R, V, P\rangle$. The problem can be solved recursively using *Bellman Equation*, where $g_i : X \to \mathbb{R}$ is a

[1]https://github.com/qcappart/hybrid-cp-rl-solver

state-value function representing the optimal reward of being at state s_i at stage i:

$$g_i(s_i) = \max\Big\{R(s_i, x_i) + g_{i+1}\big(T(s_i, x_i)\big)\Big\} \quad (1)$$

This applies $\forall i \in \{1..n\}$ and such that $T(s_i, x_i) \neq \perp$. The reward is equal to zero for the final state ($g_{n+1}(s_{n+1}) = 0$) and is backtracked until $g_1(s_1)$ has been computed. This last value gives the optimal cost of $\mathcal{Q}$. Then, by tracing the values assigned to the variables x_i, the optimal solution is recovered. Unfortunately, DP suffers from the well-known curse of dimensionality, which prevents its use when dealing with problems involving large state/control spaces. A partial solution to this problem is to prune dominated actions ($P(s, x) = 0$). An action is dominated if it is valid according to the recursive formulation, but is (1) either strictly worse than another action, or (2) it cannot lead to a feasible solution. In practice, pruning such dominated actions can have a huge impact on the size of the search space, but identifying them is not trivial as assessing those two conditions precisely is problem-dependent. Besides, even after pruning the dominated actions, the size of the state-action space may still be too large to be completely explored in practice.

RL Encoding

An introduction to reinforcement learning is proposed in appendices. Note that all the sets used to define an RL environment are written using a larger size font. Encoding the DP formulation into a RL environment requires to define, adequately, the *set of states*, the *set of actions*, the *transition function*, and the *reward function*, as the tuple $\langle \mathsf{S}, \mathsf{A}, \mathsf{T}, \mathsf{R}\rangle$ from the DP model $\langle S, X, T, R, V, P\rangle$ and a specific instance $\mathcal{Q}_p$ of the COP that we are considering. The initial state of the RL environment corresponds to the first stage of the DP model, where no variable has been assigned yet.

State For each stage i of the DP model, we define the RL state s_i as the pair $(\mathcal{Q}_p, s_i)$, where $s_i \in S$ is the DP state at the same stage i, and $\mathcal{Q}_p$ is the problem instance we are considering. Note that the second part of the state (s_i) is *dynamic*, as it depends on the current stage i in the DP model, or similarly, to the current time-step of the RL episode, whereas the first part ($\mathcal{Q}_p$) is *static* as it remains the same for the whole episode. In practice, each state is embedded into a tensor of features, as it serves as input of a neural network.

Action Given a state s_i from the DP model at stage i and its control x_i, an action $\mathsf{a}_\mathsf{i} \in \mathsf{A}$ at a state s_i has a one-to-one relationship with the control x_i. The action a_i can be done if and only if x_i is valid under the DP model. The idea is to allow only actions that are consistent with regards to the DP model, the validity conditions, and the eventual dominance conditions. Formally, the set of feasible actions A at stage i are as follows:

$$\mathsf{A}_i = \Big\{v_i \,\Big|\, v_i \in D(x_i) \wedge V(s_i, v_i) = 1 \wedge P(s_i, v_i) = 1\Big\} \quad (2)$$

Transition The RL transition T gives the state s_{i+1} from s_i and a_i in the same way as the transition function T of the DP model gives a state s_{i+1} from a previous state s_i and a control value v_i. Formally, we have the deterministic transition:

$$\mathsf{s}_{i+1} = \mathsf{T}(\mathsf{s}_i, \mathsf{a}_i) = \big(\mathcal{Q}_p, T(s_i, \mathsf{a}_i)\big) = \big(\mathcal{Q}_p, T(s_i, v_i)\big) \quad (3)$$

Reward An initial idea for designing the RL reward function R is to use the reward function R of the DP model using the current state s_i and the action a_i that has been selected. However, performing a sequence of actions in a DP subject to validity conditions can lead to a state with no solutions, which must be avoided. Such a situation happens when a state with no action is reached whereas at least one control $x \in X$ has not been assigned to a value v. Finding first a feasible solution must then be prioritized over maximizing the DP reward and is not considered with this simple form of the RL reward. Based on this, two properties must be satisfied in order to ensure that the reward will drive the RL agent to the optimal solution of the COP: (1) the reward collected through an episode e_1 must be lesser than the reward of an episode e_2 if the COP solution of e_1 is worse than the one obtained with e_2, and (2) the total reward collected through an episode giving an unfeasible solution must be lesser than the reward of any episode giving a feasible solution. A formal definition of these properties is proposed in the supplementary material. By doing so, we ensure that the RL agent has incentive to find, first, feasible solutions (i.e., maximizing the first term is more rewarding), and, then, finding the best ones (i.e., then, maximizing the second term). The reward we designed is as follows : $\mathsf{R}(\mathsf{s}, \mathsf{a}) = \rho \times \big(1 + |\mathtt{UB}(\mathcal{Q}_p)| + R(\mathsf{s}, \mathsf{a})\big)$; where $\mathtt{UB}(\mathcal{Q}_p)$ corresponds to an upper bound of the objective value that can be reached for the COP $\mathcal{Q}_p$. The term $1 + |\mathtt{UB}(\mathcal{Q}_p)|$ is a constant factor that gives a strict upper bound on the reward of any solution of the DP and drives the agent to progress into a feasible solution first. For the *travelling salesman problem with time windows*, this bound can be, for instance, the maximum distance that can be traveled in a complete tour (computed in $\mathcal{O}(1)$). This term is required in order to prioritize the fact that we want first a feasible solution. The absolute value ensures that the term is positive and is used to negate the effect of negative rewards that may lead the agent to stop the episode as soon as possible. The second term $R(\mathsf{s}, \mathsf{a})$ forces then the agent to find the best feasible solution. Finally, a scaling factor $\rho \in \mathbb{R}$ can also be added in order to compress the space of rewards into a smaller interval value near zero. Note that for DP models having only feasible solutions, the first term can be omitted.

Learning Algorithm

We implemented two different agents, one based on a value-based method (DQN) and a second one based on policy gradient (PPO). In both cases, the agent is used to parametrize the weight vector ($\mathbf{w}$) of a neural network giving either the Q-values (DQN), or the policy probabilities (PPO). The training is done using randomly generated instances sampled from a similar distribution to those we want to solve. It is important to mention that this learning procedure makes the assumption that we have a generator able to create random instances

($\mathcal{Q}_p$) that follows the same distribution that the ones we want to tackle, or a sufficient number of similar instances from past data. Such an assumption is common in the vast majority of works tackling NP-hard problems using ML (Khalil et al. 2017; Kool, Van Hoof, and Welling 2018; Cappart et al. 2019), and, despite being strong, has nonetheless a practical interest when repeatedly solving similar instances of the same problem (e.g., package shipping by retailers)

Neural Network Architecture

In order to ensure the genericity and the efficiency of the framework, we have two requirements for designing the neural network architecture: (1) be able to handle instances of the same COPs, but that have a different number of variables (i.e., able to operate on non-fixed dimensional feature vectors) and (2) be invariant to input permutations. In other words, encoding variables x_1, x_2, and x_3 should produce the same prediction as encoding x_3, x_1, and x_2. A first option is to embed the variables into a *set transformer* architecture (Lee et al. 2018), that ensures these two requirements. Besides, many COPs also have a natural graph structure that can be exploited. For such a reason, we also considered another embedding based on *graph attention network* (GAT) (Veličković et al. 2017). The embedding, either obtained using GAT or *set transformer*, can then be used as an input of a feed-forward network to get a prediction. Case studies will show a practical application of both architectures. For the DQN network, the dimension of the last layer output corresponds to the total number of actions for the COP and output an estimation of the Q-values for each of them. The output is then masked in order to remove the unfeasible actions. Concerning PPO, distinct networks for the actor and the critic are built. The last layer on the critic output only a single value. Concerning the actor, it is similar as the DQN case but a softmax selection is used after the last layer in order to obtain the probability to select each action.

CP Encoding

An introduction to constraint programming is proposed in appendices. Note that the `teletype font` is used to refer to CP notations. This section describes how a DP formulation can be encoded in a CP model. Modeling a problem using CP consists in defining the tuple $\langle \mathtt{X}, \mathtt{D}, \mathtt{C}, \mathtt{O} \rangle$ where `X` is the *set of variables*, `D(X)` is the *set of domains*, `C` is the *set of constraints*, and `O` is the *objective function*. Let us consider the DP formulation $\langle S, X, T, R, V, P \rangle$ with also n the number of stages.

Variables and domains We make a distinction between the *decision variables*, on which the search is performed, and the *auxiliary variables* that are linked to the decision variables, but that are not branched on during the search. The encoding involves two variables per stage: (1) $\mathtt{x}_i^s \in \mathtt{X}$ is an auxiliary variable representing the current state at stage i whereas (2) $\mathtt{x}_i^a \in \mathtt{X}$ is a decision variable representing the action done at this state, similarly to the `regular` decomposition (Pesant 2004). Besides, a last auxiliary variable is considered for the stage $n + 1$, which represents the final state of the system. In the optimal solution, the variables thus indicate the best state that can be reached at each stage, and the best action to select as well.

Constraints The constraints of our encoding have two purposes. Firstly, they must ensure the consistency of the DP formulation. It is done (1) by setting the initial state to a value (e.g., ϵ), (2) by linking the state of each stage to the previous one through the transition function (T), and finally (3) by enforcing each transition to be *valid*, in the sense that they can only generate a feasible state of the system. Secondly, other constraints are added in order to remove dominated actions and the subsequent states. In the CP terminology, such constraints are called *redundant constraint*, they do not change the semantic of the model, but speed-up the search. The constraints inferred by our encoding are as follows, where `validityCondition` and `dominanceCondition` are both Boolean functions detecting non-valid transitions and dominated actions, respectively.

$$\mathtt{x}_1^s = \epsilon \tag{4}$$
$$\forall i \in \{1, \dots, n\} : \mathtt{x}_{i+1}^s = T(\mathtt{x}_i^s, \mathtt{x}_i^a) \tag{5}$$
$$\forall i \in \{1, \dots, n\} : \mathtt{validityCondition}(\mathtt{x}_i^s, \mathtt{x}_i^a) \tag{6}$$
$$\forall i \in \{1, \dots, n\} : \mathtt{dominanceCondition}(\mathtt{x}_i^s, \mathtt{x}_i^a) \tag{7}$$

Setting the initial state is done in Eq. (4), enforcing the transition function in Eq. (5), keeping only the valid transitions in Eq. (6), and pruning the dominated states in Eq. (7)

Objective function The goal is to maximize the accumulated sum of rewards generated through the transition ($R : S \times A \rightarrow \mathbb{R}$) during the n stages: $\max_{\mathtt{x}^a} \big(\sum_{i=1}^{n} R(\mathtt{x}_i^s, \mathtt{x}_i^a) \big)$. Note that the optimization and branching selection is done only on the decision variables ($\mathtt{x}^a$).

Search Strategy

From a single DP formulation, we are able to (1) build a RL environment dedicated to learn the best actions to perform, and (2) state a CP model of the same problem (Figure 1). This consistency is at the heart of the framework. This section shows how the knowledge learned during the training phase can be transferred into the CP search. We considered three standard CP specific search strategy: *depth-first branch-and-bound search* (BaB), and *iterative limited discrepancy search* (ILDS), that are able to leverage knowledge learned with a value-based method as DQN, and *restart based search* (RBS), working together with policy gradient methods. The remaining of this section presents how to plug a learned heuristics inside these three search strategies.

Depth-First Branch-and-Bound Search with DQN This search works in a depth-first fashion. When a feasible solution has been found, a new constraint ensuring that the next solution has to be better than the current one is added. In case of an unfeasible solution due to an empty domain reached, the search is backtracked to the previous decision. With this procedure, and provided that the whole search space has been explored, the last solution found is then proven to be optimal.

This search requires a good heuristic for the value-selection. This can be achieved by a value-based RL agent, such as DQN. After the training, the agent gives a parametrized state-action value function $\hat{Q}(\mathsf{s}, \mathsf{a}, \mathbf{w})$, and a greedy policy can be used for the value-selection heuristic, which is intended to be of a high, albeit non-optimal, quality. The variable ordering must follow the same order as the DP model in order to keep the consistency with both encoding. As highlighted in other works (Cappart et al. 2019), an appropriate variable ordering has an important impact when solving DPs. However, such an analysis goes beyond the scope of this work.

Algorithm 1: `BaB-DQN` Search Procedure.

```
▷ Pre: $\mathcal{Q}_p$ is a COP having a DP formulation.
▷ Pre: $\mathbf{w}$ is a trained weight vector.
$\langle$X, D, C, O$\rangle$ := CPEncoding($\mathcal{Q}_p$)
$\mathcal{K} = \emptyset$
$\Psi$ := BaB-search($\langle$X, D, C, O$\rangle$)
while $\Psi$ is not completed do
    s := encodeStateRL($\Psi$)
    x := takeFirstNonAssignedVar(X)
    if s $\in \mathcal{K}$ then
        $v$ := peek($\mathcal{K}$, s)
    else
        $v := \text{argmax}_{u \in D(\mathrm{x})} \hat{Q}(\mathsf{s}, u, \mathbf{w})$
    end
    $\mathcal{K} := \mathcal{K} \cup \{\langle \mathsf{s}, v \rangle\}$
    branchAndUpdate($\Psi$, x, $v$)
end
return bestSolution($\Psi$)
```

The complete search procedure (`BaB-DQN`) is presented in Algorithm 1, taking as input a COP $\mathcal{Q}_p$, and a pre-trained model with the weight vector $\mathbf{w}$. First, the optimization problem $\mathcal{Q}_p$ in encoded into a CP model. Then, a new BaB-search Ψ is initialized and executed on the generated CP model. Until the search is not completed, a RL state s is obtained from the current CP state (`encodeStateRL`). The first non-assigned variable x_i of the DP is selected and is assigned to the value maximizing the state-action value function $\hat{Q}(\mathsf{s}, \mathsf{a}, \mathbf{w})$. All the search mechanisms inherent of a CP solver but not related to our contribution (propagation, backtracking, etc.), are abstracted in the `branchAndUpdate` function. Finally, the best solution found during the search is returned. We enriched this procedure with a *cache mechanism* ($\mathcal{K}$). During the search, it happens that similar states are reached more than once (Chu, de La Banda, and Stuckey 2010). In order to avoid recomputing the Q-values, one can store the Q-values related to a state already computed and reuse them if the state is reached again. In the worst-case, all the action-value combinations have to be tested. This gives the upper bound $\mathcal{O}(d^m)$, where m is the number of actions of the DP model and d the maximal domain size. Note that this bound is standard in a CP solver. As the algorithm is based on DFS, the worst-case space complexity is $\mathcal{O}(d \times m + |\mathcal{K}|)$, where $|\mathcal{K}|$ is the cache size.

Iterative Limited Discrepancy Search with DQN

Iterative limited discrepancy search (ILDS) (Harvey and Ginsberg 1995) is a search strategy commonly used when we have a good prior on the quality of the value selection heuristic used for driving the search. The idea is to restrict the number of decisions deviating from the heuristic choices (i.e., a discrepancy) by a threshold. By doing so, the search will explore a subset of solutions that are likely to be good according to the heuristic while giving a chance to reconsider the heuristic selection which may be sub-optimal. This mechanism is often enriched with a procedure that iteratively increases the number of discrepancies allowed once a level has been fully explored.

As ILDS requires a good heuristic for the value-selection, it is complementary with a value-based RL agent, such as DQN. After the training, the agent gives a parametrized state-action value function $\hat{Q}(\mathsf{s}, \mathsf{a}, \mathbf{w})$, and the greedy policy $\text{argmax}_a \hat{Q}(\mathsf{s}, \mathsf{a}, \mathbf{w})$ can be used for the value-selection heuristic, which is intended to be of a high, albeit non-optimal, quality. The variable ordering must follow the same order as the DP model in order to keep the consistency with both encoding.

Algorithm 2: `ILDS-DQN` Search Procedure.

```
▷ Pre: $\mathcal{Q}_p$ is a COP having a DP formulation.
▷ Pre: $\mathbf{w}$ is a trained weight vector.
▷ Pre: $I$ is the threshold of the iterative LDS.

$\langle$X, D, C, O$\rangle$ := CPEncoding($\mathcal{Q}_p$)
$c^\star = -\infty$, $\mathcal{K} = \emptyset$
for $i$ from $0$ to $I$ do
    $\Psi$ := LDS-search($\langle$X, D, C, O$\rangle$, $i$)
    while $\Psi$ is not completed do
        s := encodeStateRL($\Psi$)
        x := takeFirstNonAssignedVar(X)
        if s $\in \mathcal{K}$ then
            $v$ := peek($\mathcal{K}$, s)
        else
            $v := \text{argmax}_{u \in D(\mathrm{x})} \hat{Q}(\mathsf{s}, u, \mathbf{w})$
        end
        $\mathcal{K} := \mathcal{K} \cup \{\langle \mathsf{s}, v \rangle\}$
        branchAndUpdate($\Psi$, x, $v$)
    end
    $c^\star$ := max($c^\star$, bestSolution($\Psi$))
end
return $c^\star$
```

The complete search procedure we designed (`ILDS-DQN`) is presented in Algorithm 2, taking as input a COP $\mathcal{Q}$, a pre-trained model with the weight vector $\mathbf{w}$, and an iteration threshold I for the ILDS. First, the optimization problem $\mathcal{Q}$ in encoded into a CP model. Then, for each number $i \in \{1, \ldots, I\}$ of discrepancies allowed, a new search Ψ is initialized and executed on $\mathcal{Q}$. Until the search is not completed, a RL state s is obtained from the current CP state (`encodeStateRL`). The first non-assigned variable x_i of the DP is selected and is assigned to the value maximizing the state-action value function $\hat{Q}(\mathsf{s},\mathsf{a},\mathbf{w})$. All the search mechanisms inherent of a CP solver but not related to our contribution (propagation, backtracking, etc.), are abstracted in the `branchAndUpdate` function. Finally, the best solution found during the search is returned. The cache mechanism ($\mathcal{K}$) introduced for the BaB search is reused. The worst-case bounds are the same as `BaB-DQN` presented in the main manuscript: $\mathcal{O}(d^m)$ for the time complexity, and $\mathcal{O}(d \times m + |\mathcal{K}|)$ for the space complexity, where m is the number of actions of the DP model, d is the maximal domain size, and $|\mathcal{K}|$ is the cache size.

Restart-Based Search with PPO

Restart-based search (RBS) is another search strategy, which involves multiple restarts to enforce a suitable level of exploration. The idea is to execute the search, to stop it when a given threshold is reached (i.e., execution time, number of nodes explored, number of failures, etc.), and to restart it. Such a procedure works only if the search has some randomness in it, or if new information is added along the search runs. Otherwise, the exploration will only consider similar sub-trees. A popular design choice is to schedule the restart on the Luby sequence (Luby, Sinclair, and Zuckerman 1993), using the number of failures for the threshold, and *branch-and-bound* for creating the search tree.

The sequence starts with a threshold of 1. Each next parts of the sequence is the entire previous sequence with the last value of the previous sequence doubled. The sequence can also be scaled with a factor σ, multiplying each element. As a controlled randomness is a key component of this search, it can naturally be used with a policy $\pi(\mathsf{s},\mathbf{w})$ parametrized with a policy gradient algorithm. By doing so, the heuristic randomly selects a value among the feasible ones, and according to the probability distribution of the policy through a softmax function. It is also possible to control the exploration level by tuning the softmax function with a standard Boltzmann temperature τ. The complete search process is depicted in Algorithm 3. Note that the cache mechanism is reused in order to store the vector of action probabilities for a given state. The worst-case bounds are the same as `BaB-DQN` presented in the main manuscript: $\mathcal{O}(d^m)$ for the time complexity, and $\mathcal{O}(d \times m + |\mathcal{K}|)$ for the space complexity, where m is the number of actions of the DP model, d is the maximal domain size and, $|\mathcal{K}|$ is the cache size.

Algorithm 3: `RBS-PPO` Search Procedure.

```
▷ Pre: $\mathcal{Q}_p$ is a COP having a DP formulation.
▷ Pre: $\mathbf{w}$ is a trained weight vector.
▷ Pre: $I$ is the number of restarts to do.
▷ Pre: $\sigma$ is the Luby scale factor.
▷ Pre: $\tau$ is the softmax temperature.

$\langle \mathtt{X},\mathtt{D},\mathtt{C},\mathtt{O}\rangle := \mathtt{CPEncoding}(\mathcal{Q}_p)$
$c^\star = -\infty,\ \mathcal{K} = \emptyset$
for $i$ from $0$ to $I$ do
    $\mathcal{L} = \mathtt{Luby}(\sigma, i)$
    $\Psi := \mathtt{BaB\text{-}search}(\langle \mathtt{X},\mathtt{D},\mathtt{C},\mathtt{O}\rangle, \mathcal{L})$
    while $\Psi$ is not completed do
        $\mathsf{s} := \mathtt{encodeStateRL}(\Psi)$
        $\mathtt{x} := \mathtt{takeFirstNonAssignedVar}(\mathtt{X})$
        if $\mathsf{s} \in \mathcal{K}$ then
            $p := \mathtt{peek}(\mathcal{K}, \mathsf{s})$
        else
            $p := \pi(\mathsf{s}, \mathbf{w})$
        end
        $\mathcal{K} := \mathcal{K} \cup \{\langle \mathsf{s}, p\rangle\}$
        $v \sim_{D(x)} \mathtt{softmaxSelection}(p, \tau)$
        $\mathtt{branchAndUpdate}(\Psi, \mathtt{x}, v)$
    end
    $c^\star := \max\big(c^\star, \mathtt{bestSolution}(\Psi)\big)$
end
return $c^\star$
```

Experimental Results

The goal of the experiments is to evaluate the efficiency of the framework for computing solutions of challenging COPs having a DP formulation. To do so, comparisons of our three learning-based search procedures (`BaB-DQN`, `ILDS-DQN`, `RBS-PPO`) with a standard CP formulation (`CP-model`), stand-alone RL algorithms (`DQN`, `PPO`), and industrial solvers are performed. Three NP-hard problems are considered in the main manuscript: the *travelling salesman problem with time windows* (TSPTW), involving non-linear constraints, and the *4-moments portfolio optimization problem* (PORT), which has a non-linear objective, and the *0-1 knapsack problem* (KNAP). In order to ease the future research in this field and to ensure reproducibility, the implementation, the models, the results, and the hyper-parameters used are released with the permissive MIT open-source license. Algorithms used for training have been implemented in `Python` and `Pytorch` (Paszke et al. 2019) is used for designing the neural networks. Library `DGL` (Wang et al. 2019) is used for implementing graph embedding, and `SetTransformer` (Lee et al. 2018) for set embedding. The CP solver used is `Gecode` (Team 2008), which has the benefit to be open-source and to offer a lot of freedom for designing new search procedures. As `Gecode` is implemented in `C++`, an operability interface with `Python` code is required. It is done using `Pybind11` (Jakob, Rhinelander, and Moldovan 2017). Training time is limited to 48 hours, memory consumption to 32 GB and 1 GPU (Tesla V100-SXM2-32GB) is used per model. Models are trained with a single run. A new model is recorded after each 100 episodes of the RL algorithm and the model achieving the best average reward on a validation set of 100 instances generated in the same way as for the training is selected. The final evaluation is done on 100 other instances (still randomly generated in the same manner) using Intel Xeon E5-2650 CPU with 32GB of RAM and a time limit of 60 minutes. Detailed information about the hyper-parameters tested and selected are proposed in the supplementary material.

Travelling Salesman Problem with Time Windows

Detailed information about this case study (TSPTW) and the baselines used for comparison is proposed in supplementary material. In short, OR-Tools is an industrial solver developed by Google, PPO uses a beam-search decoding of width 64, and CP-nearest solves the DP formulation with CP, but without the learning part. A nearest insertion heuristic is used for the value-selection instead. Results are summarized in Table 1. First of all, we can observe that OR-Tools, CP-model, and DQN are significantly outperformed by the hybrid approaches. Good results are nevertheless achieved by CP-nearest, and PPO. We observe that the former is better to prove optimality, whereas the latter is better to discover feasible solutions. However, when the size of instances increases, both methods have more difficulties to solve the problem and are also outperformed by the hybrid methods, which are both efficient to find solutions and to prove optimality. Among the hybrid approaches, we observe that DQN-based searches give the best results, both in finding solutions and in proving optimality.

We also note that *caching* the predictions is useful. Indeed, the learned heuristics are costly to use, as the execution time to finish the search is larger when the cache is disabled. For comparison, the average execution time of a value-selection without caching is 34 milliseconds for BaB-DQN (100 cities), and goes down to 0.16 milliseconds when caching is enabled. For CP-nearest, the average time is 0.004 milliseconds. It is interesting to see that, even being significantly slower than the heuristic, the hybrid approach is able to give the best results.

4-Moments Portfolio Optimization (PORT)

Detailed information about this case study (Atamtürk and Narayanan 2008; Bergman and Cire 2018) is proposed in supplementary material. In short, Knitro and APOPT are two general non-linear solvers. Given that the problem is non-convex, these solvers are not able to prove optimality as they may be blocked in local optima. The results are summarized in Table 2. When optimality is not proved, hybrid methods are run until the timeout. Let us first consider the continuous case. For the smallest instances, we observe that BaB-DQN*, ILDS-DQN*, and CP-model achieve the best results, although only BaB-DQN* has been able to prove optimality for all the instances. For larger continuous instances, the non-linear solvers achieve the best results, but are nevertheless closely followed by RBS-PPO*. When the coefficients of variables are floored (discrete case), the objective function is not continuous anymore, making the problem harder for non-linear solvers, which often exploit information from derivatives for the solving process. Such a variant is not supported by APOPT. Interestingly, the hybrid approaches do not suffer from this limitation, as no assumption on the DP formulation is done beforehand. Indeed, ILDS-DQN* and BaB-DQN* achieve the best results for the smallest instances and RBS-PPO* for the larger ones.

0-1 Knapsack Problem (KNAP)

Detailed information about this case study is proposed in supplementary material. In short, COIN-OR is a integer programming solver, and three types of instances, that differ from the correlation between the weight and the profit of each item, are considered (Pisinger 2005). The results (size of 50, 100 and 200 with three kinds of weight/profit correlations - easy, medium, and hard) are summarized in Table 3. For each approach, the optimality gap (i.e., the ratio with the optimal solution) is proposed. First, it is important to note that an integer programming solver, as COIN-OR (Saltzman 2002), is far more efficient than CP for solving the knapsack problem, which was already known. For all the instances tested, COIN-OR has been able to find the optimal solution and to prove it. No other methods have been able to prove optimality for all of the instances of any configuration. We observe that RBS-PPO* has good performances, and outperforms the RL and CP approaches. Methods based on DQN seems to have more difficulties to handle large instances, unless they are strongly correlated.

Discussion and Limitations

First of all, let us highlight that this work is not the first one attempting to use ML for guiding the decision process of combinatorial optimization solvers (He, Daume III, and Eisner 2014). According to the survey and taxonomy of (Bengio, Lodi, and Prouvost 2018), this kind of approach belongs to the third class (*Machine learning alongside optimization algorithms*) of ML approaches for solving COPs. It is for instance the case of (Gasse et al. 2019), which propose to augment branch-and-bound procedures using imitation learning. However, their approach requires supervised learning and is only limited to (integer) linear problems. The differences we have with this work are that (1) we focus on COPs modelled as a DP, and (2) the training is entirely based on RL. Thanks to CP, the framework can solve a large range of problems, as the TSPTW, involving non-linear combinatorial constraints, or the portfolio optimization problem, involving a non-linear objective function. Another limitation of imitation learning is that it requires the solver to be able to find a least a feasible solution for collecting data, which can be challenging for some problems as the TSPTW. Thanks to the use of reinforcement learning, our framework does not suffer from this restriction.

Besides its expressiveness, and in contrast to most of the related works solving the problem end-to-end (Bello et al. 2016; Kool, Van Hoof, and Welling 2018; Deudon et al. 2018; Joshi, Laurent, and Bresson 2019), our approach is able to deal with problems where finding a feasible solution is difficult and is able to provide optimality proofs. This was considered by (Bengio, Lodi, and Prouvost 2018) as an important challenge in learning-based methods for combinatorial optimization. Note also that compared to *speedup learning* (Fern 2010), hybridation with *ant colony optimization* (Meyer 2008; Khichane, Albert, and Solnon 2010; Di Gaspero, Rendl, and Urli 2013), and related mechanisms (Katsirelos and Bacchus 2005; Xia and Yap 2018), where learning is used to improve the search of the solving process for a specific instance, the knowledge learned by our approach can be used to solve new instances. The closest related work we identified is the approach of (Antuori et al. 2020) that has been developed in parallel by another team independently. Reinforcement learning is also leveraged for directing the search of a constraint programming solver. However, this last approach is restricted to a realistic transportation problem. In our work,

Table 1: Results for TSPTW. Methods with $\star$ indicate that caching is used, *Success* reports the number of instances where at least a solution has been found (among 100), *Opt.* reports the number of instances where the optimality has been proven (among 100), *Gap* reports the average gap with the best solution found by any method (in %, and only including the instances having only successes) and *Time* reports the average execution time to complete the search (in minutes, and only including the instances where the search has been completed; when the search has been completed for no instance *t.o.* (timeout) is indicated.

Approaches		20 cities				50 cities				100 cities			
Type	Name	Success	Opt.	Gap	Time	Success	Opt.	Gap	Time	Success	Opt.	Gap	Time
Constraint programming	`OR-Tools`	100	0	0	< 1	0	0	-	t.o.	0	0	-	t.o.
	`CP-model`	100	100	0	< 1	0	0	-	t.o.	0	0	-	t.o.
	`CP-nearest`	100	100	0	< 1	99	99	-	6	0	0	-	t.o.
Reinforcement learning	`DQN`	100	0	1.91	< 1	0	0	-	< 1	0	0	-	< 1
	`PPO`	100	0	0.13	< 1	100	0	0.86	5	21	0	-	46
Hybrid (no cache)	`BaB-DQN`	100	100	0	< 1	100	99	0	2	100	52	0.06	20
	`ILDS-DQN`	100	100	0	< 1	100	100	0	2	100	53	0.06	39
	`RBS-PPO`	100	100	0	< 1	100	80	0.02	12	100	0	0.18	t.o.
Hybrid (with cache)	`BaB-DQN`*	100	100	0	< 1	100	100	0	< 1	100	91	0	15
	`ILDS-DQN`*	100	100	0	< 1	100	100	0	1	100	90	0	15
	`RBS-PPO`*	100	100	0	< 1	100	99	0	2	100	11	0.04	32

Table 2: Results for PORT. Best results are highlighted, *Sol.* reports the best average objective profit reached, *Opt.* reports the number of instances where the optimality has been proven (among 100), and *Time* reports the average execution time to complete the search (in minutes, and only including the instances where the search has been completed; when the search has been completed for no instance *t.o.* -timeout- is indicated).

Approaches		Continuous coefficients									Discrete coefficients								
		20 items			50 items			100 items			20 items			50 items			100 items		
Type	Name	Sol.	Opt.	Time	Sol.	Opt.	Time	Sol.	Opt.	Time	Sol.	Opt.	Time	Sol.	Opt.	Time	Sol.	Opt.	Time
Non-linear solver	`KNITRO`	343.79	0	< 1	**1128.92**	0	< 1	**2683.55**	0	< 1	211.60	0	< 1	1039.25	0	< 1	2635.15	0	< 1
	`APOPT`	342.62	0	< 1	1127.71	0	< 1	2678.48	0	< 1	-	-	-	-	-	-	-	-	-
Constraint programming	`CP-model`	**356.49**	98	8	1028.82	0	t.o.	2562.59	0	t.o.	**359.81**	**100**	t	1040.30	0	t.o.	2575.64	0	t.o.
Reinforcement learning	`DQN`	306.71	0	< 1	879.68	0	< 1	2568.31	0	< 1	309.17	0	< 1	882.17	0	< 1	2570.81	0	< 1
	`PPO`	344.95	0	< 1	1123.18	0	< 1	2662.88	0	< 1	347.85	0	< 1	1126.06	0	< 1	2665.68	0	< 1
Hybrid (with cache)	`BaB-DQN`*	**356.49**	**100**	< 1	1047.13	0	t.o.	2634.33	0	t.o.	**359.81**	**100**	< 1	1067.37	0	t.o.	2641.22	0	t.o.
	`ILDS-DQN`*	**356.49**	**100**	< 1	1067.20	0	t.o.	2639.18	0	t.o.	**359.81**	**100**	< 1	1084.21	0	t.o.	2652.53	0	t.o.
	`RBS-PPO`*	356.35	0	t.o.	1126.09	0	t.o.	2674.96	0	t.o.	359.69	0	t.o.	**1129.53**	0	t.o.	**2679.57**	0	t.o.

Table 3: Results for KNAP. The best results after `COIN-OR` are highlighted, and the average optimality gap is reported. A timeout is always reached for the hybrids and the standard CP method.

Approaches		50 items			100 items			200 items		
Type	Name	Easy	Medium	Hard	Easy	Medium	Hard	Easy	Medium	Hard
Integer programming	`COIN-OR`	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Constraint programming	`CP-model`	0.30	2.35	2.77	16.58	5.99	5.44	26.49	7.420	6.19
Reinforcement learning	`DQN`	2.11	1.97	1.36	2.08	4.87	1.32	35.88	8.98	5.99
	`PPO`	0.08	0.27	0.21	0.16	0.42	0.14	0.37	0.80	0.80
Hybrid (with cache)	`BaB-DQN`*	0.02	**0.01**	**0.00**	0.44	1.73	0.60	4.20	7.84	**0.00**
	`ILDS-DQN`*	0.03	0.05	0.01	0.38	2.90	0.35	30.33	7.80	4.91
	`RBS-PPO`*	**0.01**	**0.01**	**0.00**	**0.01**	**0.12**	**0.04**	**0.11**	**0.90**	0.28

we proposed a generic approach that can be used for a larger range of problems thanks to the DP formulation, but without considering realistic instances. Then, we think that the ideas of both works are complementary.

In most situations, experiments show that our approach can obtain more and better solutions than the other methods with a smaller execution time. However, they also highlighted that resorting to a neural network prediction is an expensive operation to perform inside a solver, as it has to be called numerous times during the solving process. It is currently a bottleneck, especially if we would like to consider larger instances. It is why caching, despite being a simple mechanism, is important. Another possibility is to reduce the complexity of the neural network by compressing its knowledge, which can for instance be done using knowledge-distillation (Hinton, Vinyals, and Dean 2015) or by building a more compact equivalent network (Serra, Kumar, and Ramalingam 2020). Note that the *Pybind11* binding between the Python and C++ code is also a source of inefficiency. Another solution would be to implement the whole framework into a single, efficient, and expressive enough, programming language. Although not considered in this paper, it is worth mentioning that variable ordering also plays an important role in the efficiency of CP solvers. Learning a good variable ordering is another promising direction but raises additional challenge, such as a correct design of the reward.

Only three case studies are considered, but the approach proposed can be easily extended to other COPs that can be modeled as a DP. Many COPs have an underlying graph structure, and can then be represented by a GNN (Khalil et al. 2017), and the *set architecture* is also general for modelling COPs as they can take an arbitrary number of variables as input. DP encodings are also pervasive in the optimization literature and, similar to integer programming, have been traditionally used to model a wide range of problem classes (Godfrey and Powell 2002; Topaloglou, Vladimirou, and Zenios 2008; Tang, Mu, and He 2017).

An important assumption that is done is that we need a generator able to create random instances hat follows the same distribution that the ones we want to solve, or enough historical data of the same distribution, in order to train the models. This can be hardly achieved for some real-world problems where the amount of available data may be less important. Analyzing how this assumption can be relaxed is an interesting direction for future work.

Conclusion

The goal of combinatorial optimization is to find an optimal solution among a finite set of possibilities. There are many practical and industrial applications of COPs, and efficiently solving them directly results in a better utilization of resources and a reduction of costs. However, since the number of possibilities grows exponentially with the problem size, solving is often intractable for large instances. In this paper, we propose a hybrid approach, based on both deep reinforcement learning and constraint programming, for solving COPs that can be formulated as a dynamic program. To do so, we introduced an encoding able to express a DP model into a reinforcement learning environment and a constraint programming model. Then, the learning part can be carried out with reinforcement learning, and the solving part with constraint programming. The experiments carried out on the travelling salesman problem with time windows, the 4-moments portfolio optimization, and the 0-1 knapsack problem show that this framework is competitive with standard approaches and industrial solvers for instances up to 100 variables. These results suggest that the framework may be a promising new avenue for solving challenging combinatorial optimization problems. In future work, we plan to tackle industrial problems with realistic instances in order to access the applicability of the approach for real-world problems.

References

Aarts, E.; and Lenstra, J. K. 2003. *Local search in combinatorial optimization*. Princeton University Press.

Antuori, V.; Hébrard, E.; Huguet, M.-J.; Essodaigui, S.; and Nguyen, A. 2020. Leveraging Reinforcement Learning, Constraint Programming and Local Search: A Case Study in Car Manufacturing. In *International Conference on Principles and Practice of Constraint Programming*, 657–672. Springer.

Arulkumaran, K.; Deisenroth, M. P.; Brundage, M.; and Bharath, A. A. 2017. A Brief Survey of Deep Reinforcement Learning. *CoRR* abs/1708.05866. URL http://arxiv.org/abs/1708.05866.

Atamtürk, A.; and Narayanan, V. 2008. Polymatroids and mean-risk minimization in discrete optimization. *Operations Research Letters* 36(5): 618–622.

Bellman, R. 1966. Dynamic programming. *Science* 153(3731): 34–37.

Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; and Bengio, S. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940* .

Bengio, Y.; Lodi, A.; and Prouvost, A. 2018. Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon. *arXiv preprint arXiv:1811.06128* .

Bénichou, M.; Gauthier, J.-M.; Girodet, P.; Hentges, G.; Ribière, G.; and Vincent, O. 1971. Experiments in mixed-integer linear programming. *Mathematical Programming* 1(1): 76–94.

Bergman, D.; and Cire, A. A. 2018. Discrete nonlinear optimization by state-space decompositions. *Management Science* 64(10): 4700–4720.

Cappart, Q.; Goutierre, E.; Bergman, D.; and Rousseau, L.-M. 2019. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 1443–1451.

Chu, G.; de La Banda, M. G.; and Stuckey, P. J. 2010. Automatically exploiting subproblem equivalence in constraint programming. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, 71–86. Springer.

Cplex, I. I. 2009. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation* 46(53): 157.

Deudon, M.; Cournut, P.; Lacoste, A.; Adulyasak, Y.; and Rousseau, L.-M. 2018. Learning heuristics for the tsp by policy gradient. In *International conference on the integration of constraint programming, artificial intelligence, and operations research*, 170–181. Springer.

Di Gaspero, L.; Rendl, A.; and Urli, T. 2013. A hybrid ACO+CP for balancing bicycle sharing systems. In *International Workshop on Hybrid Metaheuristics*, 198–212. Springer.

Fages, J.-G.; and Prud'Homme, C. 2017. Making the first solution good! In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, 1073–1077. IEEE.

Fern, A. 2010. Speedup Learning.

Gasse, M.; Chételat, D.; Ferroni, N.; Charlin, L.; and Lodi, A. 2019. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, 15554–15566.

Gendreau, M.; and Potvin, J.-Y. 2005. Metaheuristics in combinatorial optimization. *Annals of Operations Research* 140(1): 189–213.

Ghasempour, T.; and Heydecker, B. 2019. Adaptive railway traffic control using approximate dynamic programming. *Transportation Research Part C: Emerging Technologies* .

Godfrey, G. A.; and Powell, W. B. 2002. An adaptive dynamic programming algorithm for dynamic fleet management, I: Single period travel times. *Transportation Science* 36(1): 21–39.

Harvey, W. D.; and Ginsberg, M. L. 1995. Limited discrepancy search. In *IJCAI (1)*, 607–615.

He, H.; Daume III, H.; and Eisner, J. M. 2014. Learning to Search in Branch and Bound Algorithms. In Ghahramani, Z.; Welling, M.; Cortes, C.; Lawrence, N. D.; and Weinberger, K. Q., eds., *Advances in Neural Information Processing Systems 27*, 3293–3301. Curran Associates, Inc. URL http://papers.nips.cc/paper/5495-learning-to-search-in-branch-and-bound-algorithms.pdf.

Hinton, G.; Vinyals, O.; and Dean, J. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* .

Jakob, W.; Rhinelander, J.; and Moldovan, D. 2017. pybind11–Seamless operability between C++ 11 and Python.

Joshi, C. K.; Laurent, T.; and Bresson, X. 2019. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227* .

Katsirelos, G.; and Bacchus, F. 2005. Generalized nogoods in CSPs. In *AAAI*, volume 5, 390–396.

Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, 6348–6358.

Khichane, M.; Albert, P.; and Solnon, C. 2010. Strong combination of ant colony optimization with constraint programming optimization. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, 232–245. Springer.

Kool, W.; Van Hoof, H.; and Welling, M. 2018. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475* .

Laborie, P. 2018. Objective landscapes for constraint programming. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 387–402. Springer.

Lawler, E. L.; and Wood, D. E. 1966. Branch-and-bound methods: A survey. *Operations research* 14(4): 699–719.

Lee, J.; Lee, Y.; Kim, J.; Kosiorek, A. R.; Choi, S.; and Teh, Y. W. 2018. Set transformer: A framework for attention-based permutation-invariant neural networks. *arXiv preprint arXiv:1810.00825* .

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

Meyer, B. 2008. Hybrids of constructive metaheuristics and constraint programming: A case study with aco. In *Hybrid Metaheuristics*, 151–183. Springer.

Optimization, G. 2014. Inc.,“Gurobi optimizer reference manual,” 2015.

Palmieri, A.; Régin, J.-C.; and Schaus, P. 2016. Parallel strategies selection. In *International Conference on Principles and Practice of Constraint Programming*, 388–404. Springer.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 8024–8035.

Pesant, G. 2004. A regular language membership constraint for finite sequences of variables. In *International conference on principles and practice of constraint programming*, 482–495. Springer.

Pisinger, D. 2005. Where are the hard knapsack problems? *Computers & Operations Research* 32(9): 2271–2284.

Rossi, F.; Van Beek, P.; and Walsh, T. 2006. *Handbook of constraint programming*. Elsevier.

Saltzman, M. J. 2002. COIN-OR: an open-source library for optimization. In *Programming languages and systems in computational economics and finance*, 3–32. Springer.

Serra, T.; Kumar, A.; and Ramalingam, S. 2020. Lossless Compression of Deep Neural Networks. *arXiv preprint arXiv:2001.00218* .

Sutton, R. S.; Barto, A. G.; et al. 1998. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge.

Tang, Y.; Mu, C.; and He, H. 2017. Near-space aerospace vehicles attitude control based on adaptive dynamic programming and sliding mode control. In *2017 International Joint Conference on Neural Networks (IJCNN)*, 1347–1353. IEEE.

Team, G. 2008. Gecode: Generic constraint development environment, 2006.

Topaloglou, N.; Vladimirou, H.; and Zenios, S. A. 2008. A dynamic stochastic programming model for international portfolio management. *European Journal of Operational Research* 185(3): 1501–1524.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; and Bengio, Y. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* .

Wang, M.; Yu, L.; Zheng, D.; Gan, Q.; Gai, Y.; Ye, Z.; Li, M.; Zhou, J.; Huang, Q.; Ma, C.; et al. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315* .

Wolsey, L. A.; and Nemhauser, G. L. 1999. *Integer and combinatorial optimization*, volume 55. John Wiley & Sons.

Xia, W.; and Yap, R. H. 2018. Learning robust search strategies using a bandit-based approach. *arXiv preprint arXiv:1805.03876* .