Ecole: A Gym-like Library for Machine Learning in Combinatorial Optimization Solvers

Antoine Prouvost
Mila, Polytechnique Montréal

Justin Dumouchelle
Polytechnique Montréal

Lara Scavuzzo
Technische Universiteit Delft

Maxime Gasse
Mila, Polytechnique Montréal

Didier Chételat
Polytechnique Montréal

Andrea Lodi
Mila, Polytechnique Montréal

Abstract

We present Ecole, a new library to simplify machine learning research for combinatorial optimization. Ecole exposes several key decision tasks arising in general-purpose combinatorial optimization solvers as control problems over Markov decision processes. Its interface mimics the popular OpenAI Gym library and is both extensible and intuitive to use. We aim at making this library a standardized platform that will lower the bar of entry and accelerate innovation in the field. Documentation and code can be found at `https://www.ecole.ai`.

1 Introduction

In many industrial applications, such as day-to-day lot sizing and production planning [33], it is common to repeatedly solve similar NP-hard combinatorial optimization (CO) problems. In practice those are typically fed into an off-the-shelf, general-purpose mathematical solver, which processes each new problem independently and retains no memory of the past. Yet, it is very likely that there exist strong statistical similarities between each of those sequentially solved problems, which could potentially be exploited to solve future problems more efficiently. This observation has motivated two growing lines of machine learning research: 1) pure machine learning (ML) approaches, where CO solvers are entirely replaced by an ML model trained to produce (near)-optimal solutions [6; 11; 29]; and 2) joint approaches, where hand-designed decision criteria within classical CO solvers are replaced by machine learning models trained to optimize a particular metric of the solver [7]. The latter approach is particularly attractive as it allows for exact solving, or at least for mathematical guarantees on the optimality gap (lower and upper bounds), which are often of high value in practice.

Leading general-purpose CO solvers such as Gurobi [20], IBM CPLEX [10], FICO Xpress [3] or SCIP [15] are all based on the branch-and-cut algorithm [1; 18]. This algorithm iteratively divides the feasible space and prunes away sections of that space that cannot contain the optimum using bounds derived from linear programming (LP) relaxations strengthened with cuts [31]. At many points during the algorithm, decisions must be taken that greatly impact the solving performance, traditionally by following a series of hand-crafted rules designed by operations research (OR) experts. Thus, a natural direction to improve the performance of these solvers is to replace the hand-crafted decision rules by machine learning models, trained on representative problems. This promising line of research has already shown improvement on several of these decision tasks, including variable selection [13; 27; 2; 4; 21; 16; 43], node selection [22; 36; 34], cut generation [37], column generation (a.k.a. pricing) [32], primal heuristic selection [28; 23], or formulation selection [8].

All these works have in common that learning can be formulated as a control problem over a Markov decision process (MDP), where a branch-and-cut solver constitutes the environment. Such a formulation opens the door to reinforcement learning (RL) algorithms, which have been successful in

1st Workshop on Learning Meets Combinatorial Algorithms @ NeurIPS 2020, Vancouver, Canada.

solving extremely complex tasks in other fields [35; 39]. These data-driven policies may hopefully improve upon the expert heuristics currently implemented in commercial solvers, and by doing so highlight new research directions for the combinatorial optimization community.

2 Motivation

Although the idea of using ML for decision-making within CO solvers is receiving increasing attention, research in this area also suffers from several unfortunate technical obstacles, which hinders scientific progress and innovation.

First, reproducibility is currently a major issue. The variety of solvers, problem benchmarks, hand-crafted features, and evaluation metrics used in existing studies impedes reproducibility and comparison to previous works. Those same issues have driven the ML and RL communities to adopt standardized evaluation benchmarks, such as ImageNet [12] or the Arcade Learning Environment [5]. We believe that adopting standard feature sets, problem benchmarks and evaluation metrics for several identified key problems (e.g., branching, node selection, cutting plane generation) will be highly beneficial to this research area as well.

Second, there is a high bar of entry to the field. Modern solvers are complex pieces of software whose implementation always deviates from the vanilla textbook algorithm, and which were not specifically designed for direct customization through machine learning. Implementing a new research idea often requires months of digging in the technical intricacies of low-level C solver code, even for OR experts, and requires ML experts joining the field to be very familiar with the inner working of a CO solver. On the other hand, abstracting away a proper MDP formulation using a solver API is no trivial task either for OR experts, and requires a clear understanding of statistical learning concepts and their significance. We believe that exposing several decision tasks of interest through a unified ML-compatible API will help attract interest from both the traditional ML and OR communities.

Finally, at this time the field hardly benefits from the latest advances in both ML and OR. ML experts typically employ very simplified CO solvers or no solver at all [37; 29], raising criticism among the OR community, while OR experts typically employ basic ML models and algorithms [2; 8; 23], thereby missing potential improvements. We believe that a plug-and-play API between a state-of-the-art CO solver and ML algorithms, in the form of a Gym-compatible interface, will allow for closing this gap and let the field benefit from the latest advances from both sides.

3 Proposed solution

To address these practical challenges, we propose a novel open-source library that could serve as a universal platform for research and development in the ML within CO. This new platform, the *Extensible Combinatorial Optimization Learning Environments* (Ecole) library, is designed as an interface between a CO solver and ML algorithms. It provides a collection of key decision tasks, such as variable selection or cut selection, as partially-observable (PO)-MDP environments in a way that closely mimics OpenAI Gym [9], a widely popular library among the RL community.

3.1 Design

The design of the library was guided to achieve the following objectives.

Modularity An environment in Ecole is defined by a composition of a task, an observation function and a reward function. For example, in Figure 1, a branching environment is defined with a node bipartite graph observation and the negative number of new nodes created as a reward. Users can define their own observation or reward function to fulfill their specific needs, or even define new environments and simply reuse existing observation and reward functions. These new modules can be defined either directly in C++ for speed, or in Python for flexibility.

Scalability Ecole was designed to add as little overhead as possible on top of the solver. In addition, care was taken to ensure that the library is thread-safe, and in particular Ecole was designed to be free from the Python Global Interpreter Lock (GIL). This allows for straightforward parallelism in Python with multi-threading, which simplifies data collection and policy evaluation in RL algorithms.

```python
import ecole

# set up an MDP environment
env = ecole.environment.Branching(
    # use the features from Gasse et al., 2019
    observation_function=ecole.observation.NodeBipartite(),
    # minimize the B&B tree size
    reward_function=-ecole.reward.NNodes())

# set up an instance generator
instances = ecole.instance.CombinatorialAuctionGenerator(n_items=100, n_bids=100)

# generate ten MDP episodes
for _ in range(10):
    # new instances are generated on-the-fly
    instance = next(instances)
    # save instance to disk if desired
    instance.write_problem(f"path/to/problem_{i}.lp")
    # start a new episode
    obs, action_set, reward, done = env.reset(instance)
    # unroll the control loop until the instance is solved
    while not done:
        action = ...  # decide on the next action here
        obs, action_set, reward, done, info = env.step(action)
```

Figure 1: Example code snippet, using Ecole for branching on combinatorial auction problems.

Speed The Ecole core is written in C++, interacts directly with the low-level solver API and provides a thin Python API returning Numpy arrays [38] to interface directly with ML libraries. The initial release of Ecole supports the state-of-the-art open-source solver SCIP [15] as a backend, due to its open code that gives complete access to the solver, and its widespread usage in the literature [22; 21; 16; 14; 42; 19]. We hope in future versions to expand the library to other commercial solvers as well, such as Gurobi [20], CPLEX [10] or Xpress [3] if the developers of these solvers are interested.

Flexibility Ecole can read instance files in any format understood by SCIP and can therefore be used with existing benchmark collections such as MIPLIB [17]. In addition, the library also provides out-of-the box instance generators for classical CO problems, which can be used to quickly test ideas or to offer standard benchmarks. The generated instances can be saved to disk or passed directly to Ecole environments from memory, as illustrated in Figure 1. Four instance generators are currently implemented (combinatorial auctions, maximum independent set, capacitated facility location and set covering problems) with default parameters chosen to yield solving times on the order of a minute.

Openness To encourage widespread usage of the library, we chose to distribute Ecole under an open-source BSD-3 license [26]. In addition, care was taken that the library could be installed with the popular `conda` package manager [25], which is widely used in the ML community.

3.2 Supported features

The library currently supports two control tasks. The first is hyperparameter tuning (`ecole.environment.Configuring`), the task of selecting the best solver hyperparameters before solving. The second is variable selection (`ecole.environment.Branching`), the task of deciding sequentially on the next variable to branch on during the construction of the branch-and-bound tree. The library also includes an empty "baseline" environment that can be used to benchmark against the solver in its default settings. We are actively working on expanding the library with several other environments that correspond to key research questions in the field, such as node selection and cut selection.

In addition, the library currently supports two observation functions for the state of the solving process. The first is the finite-dimensional variable-aggregated representation from Khalil et al. [27], and the second is the bipartite graphical representation from Gasse et al. [16]. Finally, the library currently supports two standard reward functions, namely the number of branch-and-bound nodes, and the number of LP iterations added since the last decision. Several standard metrics will be added, such as the primal integral, the dual integral, the primal-dual integral, and the solving time.

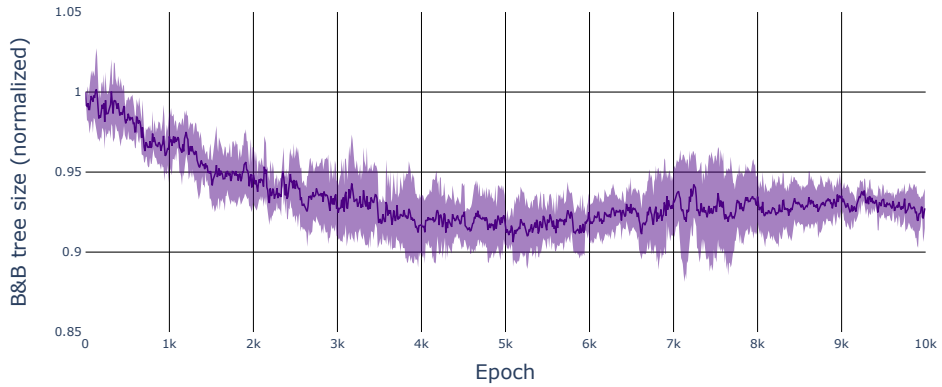


Figure 2: Training curve of a branching environment on randomly generated Combinatorial Auction instances. We report the normalized performance on validation instances, the lower the better.

3.3 Example use case

We now demonstrate a typical use case, namely reinforcement learning for variable selection in branch and bound. We used the negative number of nodes created between two decisions as reward function, and the policy has the same architecture as the GNN model of Gasse et al. [16]. We pretrained the weights by imitation learning of strong branching, as in the cited paper, and further trained the policy using REINFORCE [40] on samples of transitions encountered during rollouts. Figure 2 shows a 5-10% decrease in the branch-and-bound tree size after 10k episodes. Most interestingly for this article, the environment was defined with only a few lines of code using Ecole, namely a combination of `ecole.environment.Branching`, `ecole.observation.NodeBipartite` and `ecole.reward.NNodes`, and the code was parallelized on 8 threads using the native `threading` Python library. Equivalent code in PySCIPOpt [30], the SCIP Python API, would have been substantially more complex to write.

4 Related work

Other open-source libraries have been recently proposed to simplify research at the intersection of machine learning and combinatorial optimization. MIPLearn [41] is a customizable library for machine-learning-based solver configuration currently supporting Gurobi and CPLEX. It offers similar functionalities to the configuration environment in Ecole, which is a (potentially contextual) bandit problem, and can be framed as a borderline case of our MDP framework. In addition, ORGym [24] and OpenGraphGym [44] are Gym-like libraries for learning heuristics for a collection of combinatorial optimization problems that are formulated as sequential decision making problems. Thus, in those libraries there is an explicit MDP formulation like in Ecole, although in those the goal is to replace CO solvers entirely, while Ecole aims at improving existing CO solvers. As such, none of these libraries has the ambitious objective of Ecole, which is to serve as a standardized platform for ML within CO solvers.

5 Conclusions

In this paper, we proposed a new open-source library that offers Gym-like Markov decision process interfaces to key decision tasks in combinatorial optimization solvers. This library was designed to be fast, modular, scalable, and flexible, with easy installation. Such a library is intended to improve reproducibility, lower the bar of entry and simplify integration of recent advances from both fields, in this growing area at the intersection of machine learning and combinatorial optimization.

Acknowledgements

This work was supported by the Canada Excellence Research Chair (CERC) in Data Science for Real-Time Decision Making and IVADO.

References

[1] Tobias Achterberg and Roland Wunderling. *Mixed Integer Programming: Analyzing 12 Years of Progress*, pages 449–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-38189-8. doi: 10.1007/978-3-642-38189-8_18.

[2] Alejandro M. Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29:185–195, 2017.

[3] Robert Ashford. Mixed integer programming: A historical perspective with xpress-mp. *Annals of Operations Research*, 149(1):5–17, 2007.

[4] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik. Learning to branch. In Jennifer G. Dy and Andreas Krause, editors, *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 353–362. PMLR, 2018.

[5] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[6] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *Proceedings of the Fifth International Conference on Learning Representations*, 2017.

[7] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020.

[8] Pierre Bonami, Andrea Lodi, and Giulia Zarpellon. Learning a classification of mixed-integer quadratic programming problems. In Willem-Jan van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 595–604, Cham, 2018. Springer International Publishing. ISBN 978-3-319-93031-2.

[9] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[10] IBM CPLEX. CPLEX Optimizer User Manual, 2020. URL `https://www.ibm.com/analytics/cplex-optimizer`.

[11] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[13] Giovanni Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. Dash: Dynamic approach for switching heuristics. *European Journal of Operational Research*, 248:943–953, 2016.

[14] Jian-Ya Ding, Chao Zhang, Lei Shen, Shengyin Li, Bing Wang, Yinghui Xu, and Le Song. Optimal solution predictions for mixed integer programs. *arXiv preprint arXiv:1906.09575*, 2019.

[15] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe

Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. ZIB-Report 20-10, Zuse Institute Berlin, March 2020.

[16] Maxime Gasse, Didier Chetelat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 15580–15592. Curran Associates, Inc., 2019.

[17] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. Technical report, Optimization Online, August 2019.

[18] Martin Grötschel. *The sharpest cut: The impact of Manfred Padberg and his work*. SIAM, 2004.

[19] Prateek Gupta, Maxime Gasse, Elias B Khalil, M Pawan Kumar, Andrea Lodi, and Yoshua Bengio. Hybrid models for learning to branch. In *Advances in neural information processing systems*, 2020.

[20] Gurobi Optimization LLC. Gurobi Optimizer Reference Manual, 2020. URL `http://www.gurobi.com`.

[21] Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. arXiv:1805.01415, 2018.

[22] He He, Hal Daume III, and Jason M Eisner. Learning to search in branch and bound algorithms. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3293–3301. Curran Associates, Inc., 2014.

[23] Gregor Hendel, Matthias Miltenberger, and Jakob Witzig. Adaptive algorithmic behavior for solving mixed integer programs using bandit algorithms. In *Operations Research Proceedings*, pages 513–519. Springer, 2018.

[24] Christian D. Hubbs, Hector D. Perez, Owais Sarwar, Nikolaos V. Sahinidis, Ignacio E. Grossmann, and John M. Wassick. Or-gym: A reinforcement learning library for operations research problems, 2020.

[25] Anaconda Inc. Conda Package Manager, 2020. URL `https://conda.io`.

[26] Open Source Initiative et al. The bsd 3-clause license. *URL: http://opensource. org/licenses/BSD-2-Clause*, 2015.

[27] Elias B. Khalil, Pierre Le Bodic, Le Song, George L. Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In Dale Schuurmans and Michael P. Wellman, editors, *AAAI*, pages 724–731. AAAI Press, 2016. ISBN 978-1-57735-760-5.

[28] Elias B. Khalil, Bistra Dilkina, George L. Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 659–666, 2017.

[29] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *International Conference on Learning Representations*, 2019.

[30] Stephen Maher, Matthias Miltenberger, João Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. PySCIPOpt: Mathematical programming in python with the SCIP optimization suite. In *Mathematical Software – ICMS 2016*, pages 301–307. Springer International Publishing, 2016. doi: 10.1007/978-3-319-42432-3_37.

[31] John E Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of applied optimization*, 1:65–77, 2002.

[32] Mouad Morabit, Guy Desaulniers, and Andrea Lodi. Machine-learning-based column selection for column generation. Les Cahiers du GERAD G-2020-29, GERAD, HEC Montréal, Canada, 2020.

[33] Yves Pochet and Laurence A Wolsey. *Production planning by mixed integer programming*. Springer Science & Business Media, 2006.

[34] Ashish Sabharwal, Horst Samulowitz, and Chandra Reddy. Guiding combinatorial optimization with uct. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 356–361. Springer, 2012.

[35] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. ISSN 0036-8075. doi: 10.1126/science.aar6404. URL `https://science.sciencemag.org/content/362/6419/1140`.

[36] Jialin Song, Ravi Lanka, Albert Zhao, Yisong Yue, and Masahiro Ono. Learning to search via retrospective imitation. arXiv:1804.00846, 2018.

[37] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. In *Proceedings of the 37th International Conference on Machine Learning*, pages 1483–1492, 2020.

[38] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[39] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019.

[40] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[41] Alinson S Xavier and Feng Qiu. MIPLearn, 2020. URL `https://anl-ceeesa.github.io/MIPLearn`.

[42] Kaan Yilmaz and Neil Yorke-Smith. Learning efficient search approximation in mixed integer branch and bound. *arXiv preprint arXiv:2007.03948*, 2020.

[43] Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. Parameterizing branch-and-bound search trees to learn branching policies. *arXiv preprint arXiv:2002.05120*, 2020.

[44] Weijian Zheng, Dali Wang, and Fengguang Song. OpenGraphGym: A parallel reinforcement learning framework for graph optimization problems. In *International Conference on Computational Science*, pages 439–452. Springer, 2020.

Combining Reinforcement Learning and Constraint Programming for Combinatorial Optimization

Quentin Cappart,[1, 2] Thierry Moisan,[2] Louis-Martin Rousseau[1], Isabeau Prémont-Schwarz[2], Andre A. Cire[3]

[1] 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 (Schulte, Lagerkvist, and Tack 2006). 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.

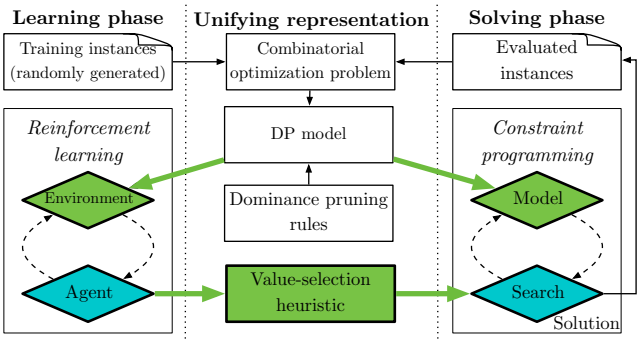


Figure 1: Overview of our framework for solving COPs.

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 re-

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

cursively using *Bellman Equation*, where $g_i : X \to \mathbb{R}$ is a *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 $T(s_i, x_i) \neq \bot$. 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 $\mathsf{a_i} \in \mathsf{A}$ at a state s_i has a one-to-one relationship with the control x_i. The action $\mathsf{a_i}$ 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 $\mathtt{X}$ is the *set of variables*, $\mathtt{D}(\mathtt{X})$ is the *set of domains*, $\mathtt{C}$ is the *set of constraints*, and $\mathtt{O}$ 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.

$$\mathtt{x}_1^s = \epsilon \tag{4}$$

$$\forall i \in \{1, \dots, n\} : \mathtt{x}_{i+1}^s = T(\mathtt{x}_i^s, \mathtt{x}_i^a) \tag{5}$$

$$\forall i \in \{1, \dots, n\} : \texttt{validityCondition}(\mathtt{x}_i^s, \mathtt{x}_i^a) \tag{6}$$

$$\forall i \in \{1, \dots, n\} : \texttt{dominanceCondition}(\mathtt{x}_i^s, \mathtt{x}_i^a) \tag{7}$$

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 ($R : S \times A \to \mathbb{R}$) during the n stages: $\max_{\mathtt{x}^a} \left(\sum_{i=1}^{n} R(\mathtt{x}_i^s, \mathtt{x}_i^a) \right)$. 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.
$\langle \mathtt{X}, \mathtt{D}, \mathtt{C}, \mathtt{O} \rangle := \mathtt{CPEncoding}(\mathcal{Q}_p)$
$\mathcal{K} = \emptyset$
$\Psi := \mathtt{BaB\text{-}search}(\langle \mathtt{X}, \mathtt{D}, \mathtt{C}, \mathtt{O} \rangle)$
while Ψ **is not completed do**
  $\mathsf{s} := \mathtt{encodeStateRL}(\Psi)$
  $\mathtt{x} := \mathtt{takeFirstNonAssignedVar}(\mathtt{X})$
  if $\mathsf{s} \in \mathcal{K}$ **then**
    $v := \mathtt{peek}(\mathcal{K}, \mathsf{s})$
  else
    $v := \text{argmax}_{u \in D(\mathtt{x})} \hat{Q}(\mathsf{s}, u, \mathbf{w})$
  end
  $\mathcal{K} := \mathcal{K} \cup \{\langle \mathsf{s}, v \rangle\}$
  $\mathtt{branchAndUpdate}(\Psi, \mathtt{x}, v)$
end
return $\mathtt{bestSolution}(\Psi)$

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.

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

imizing 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:** σ is the Luby scale factor.
▷ **Pre:** τ 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 Ψ **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
  $c^\star := \max\left(c^\star, \mathtt{bestSolution}(\Psi)\right)$
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` (Schulte, Lagerkvist, and Tack 2006), 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 supplemen-

tary 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 Tables 2 and 3. When optimality is not proved, hybrid methods are run until the timeout. Let us first consider the continuous case (Table 2). 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 (Table 3), 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 4. 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 *failure-driven explanation-based learning* (Kambhampati 1998), 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

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 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 items			50 items			100 items		
Type	Name	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
	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.
Reinforcement learning	DQN	306.71	0	< 1	879.68	0	< 1	2568.31	0	< 1
	PPO	344.95	0	< 1	1123.18	0	< 1	2662.88	0	< 1
Hybrid (with cache)	BaB-DQN*	**356.49**	**100**	< 1	1047.13	0	t.o.	2634.33	0	t.o.
	ILDS-DQN*	**356.49**	**100**	< 1	1067.20	0	t.o.	2639.18	0	t.o.
	RBS-PPO*	356.35	0	t.o.	1126.09	0	t.o.	2674.96	0	t.o.

Table 2: Results for PORT (continuous coefficients). 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		20 items			50 items			100 items		
Type	Name	Sol.	Opt.	Time	Sol.	Opt.	Time	Sol.	Opt.	Time
Non-linear solver	KNITRO	211.60	0	< 1	1039.25	0	< 1	2635.15	0	< 1
	APOPT	-	-	-	-	-	-	-	-	-
Constraint programming	CP-model	**359.81**	**100**	t.o.	1040.30	0	t.o.	2575.64	0	t.o.
Reinforcement learning	DQN	309.17	0	< 1	882.17	0	< 1	2570.81	0	< 1
	PPO	347.85	0	< 1	1126.06	0	< 1	2665.68	0	< 1
Hybrid (with cache)	BaB-DQN*	**359.81**	**100**	< 1	1067.37	0	t.o.	2641.22	0	t.o.
	ILDS-DQN*	**359.81**	**100**	< 1	1084.21	0	t.o.	2652.53	0	t.o.
	RBS-PPO*	359.69	0	t.o.	**1129.53**	0	t.o.	**2679.57**	0	t.o.

Table 3: Results for PORT (discrete coefficients). 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).

transportation problem. In our work, 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

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

Table 4: 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.

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 that 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 and important 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 assess 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.

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.

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

Kambhampati, S. 1998. On the relations between intelligent backtracking and failure-driven explanation-based learning in constraint satisfaction and planning. *Artificial Intelligence* 105(1-2): 161–208.

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.

Schulte, C.; Lagerkvist, M.; and Tack, G. 2006. Gecode. *Software download and online material at the website: http://www. gecode. org* 11–13.

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.

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* .

Deep Reinforcement Learning for Exact Combinatorial Optimization: Learning to Branch

Tianyu Zhang[§]
University of Alberta
Edmonton, Canada
tianyu.zhang@ualberta.ca

Amin Banitalebi-Dehkordi
Huawei Technologies Canada Co., Ltd.
Vancouver, Canada
amin.banitalebi@huawei.com

Yong Zhang
Huawei Technologies Canada Co., Ltd.
Vancouver, Canada
yong.zhang3@huawei.com

***Abstract*—Branch-and-bound is a systematic enumerative method for combinatorial optimization, where the performance highly relies on the variable selection strategy. State-of-the-art handcrafted heuristic strategies suffer from relatively slow inference time for each selection, while the current machine learning methods require a significant amount of labeled data. We propose a new approach for solving the data labeling and inference latency issues in combinatorial optimization based on the use of the reinforcement learning (RL) paradigm. We use imitation learning to bootstrap an RL agent and then use Proximal Policy Optimization (PPO) to further explore global optimal actions. Then, a value network is used to run Monte-Carlo tree search (MCTS) to enhance the policy network. We evaluate the performance of our method on four different categories of combinatorial optimization problems and show that our approach performs strongly compared to the state-of-the-art machine learning and heuristics based methods.**

I. Introduction

Combinatorial optimization is a broad topic covering several areas of computer science, operations research, and artificial intelligence. The fundamental goal of combinatorial optimization is to find optimal configurations from a finite discrete set that satisfy all given conditions, which involves enormous discrete search spaces. Examples include internet routing [1], scheduling [2], protein structure prediction [3], combinatorial auctions [4]. Many real-life problems can also be formalized as combinatorial optimization problems, including the travelling salesman [5], the vertex colouring [6], and the vehicle routing problems [7], [8]. As combinatorial optimization includes various NP-hard problems, there is a significant demand for efficient combinatorial optimization algorithms.

Several exact combinatorial optimization algorithms have been proposed to provide theoretical guarantees on finding optimal solutions or determining the non-existence of a solution. The core idea is to prune the candidate solution set by confidently introducing new conditions. Branch-and-bound (B&B) [9] is an example of an exact method to solve the combinatorial problem, which recursively divides the candidate solution set into disjoint subsets and rules out subsets that cannot have any candidate solutions satisfying all conditions. It has shown a reliable performance in the domain of mixed-integer linear programs (MILPs) to which many combinatorial problems can be reduced [10]. Several commercial optimization solvers (e.g. CPLEX, Gurobi) use a B&B algorithm to solve MILP instances. However, two decisions must be made at each iteration of B&B: *node selection* and *variable selection*, which determine the next solution set to be partitioned, and the variable to be used as the partition rule, respectively. Most state-of-the-art optimizers use heuristics hard-coded by domain experts to improve the performance [11]. However, such heuristics are hard to develop and require adjustment for different problems [12].

In recent years, an increasing number of studies have been focusing on training machine learning (ML) algorithms to solve MILP problems. The idea is that some procedural parts of the solvers may be replaced by ML models that are trained with historical data. However, most ML models are trained through supervised learning, which requires the mapping between training inputs and outputs. Since the optimal labels are typically not accessible, supervised learning is not capable for most MILP problems [13]. In contrast, reinforcement learning (RL) algorithms show a potential benefit to the B&B decision-making, thanks to the fact that the B&B decision-making process can be modelled as a Markov decision process (MDP) [12], [14]. This offers an opportunity to use statistical learning for decision-making.

In this work, we provide an RL-based approach to learn a variable selection strategy, which is the core of the B&B method. Our agent is trained to maximize the improvement of dual bound integral with respect to time in the B&B method. We adopt the design of Proximal Policy Optimization (PPO) [15], combining the idea of imitation learning to improve the sample efficiency and advance imitated policy. We imitate the Full Strong Branching (FSB) [16] variable selection rule to discourage the exploration of unpromising directions. We also introduce a Monte Carlo Tree Search (MCTS) like approach [17] to encourage exploration during the training phase and reinforce the action selection strategy.

We evaluate our RL agent with four kinds of widely adopted combinatorial optimization problems. The experiments show that our approach can outperform state-of-the-art methods under multiple metrics. In summary, our contribution is threefold:

- We implement and evaluate an RL-based agent training framework for B&B variable selection problem and achieve comparable performance with the state-of-the-art GCNN approach using supervised learning.
- To facilitate the decision quality, we propose a new MDP formulation that is more suitable for the B&B method.

[§]Work done during an internship at Huawei Technologies Canada Co., Ltd.

- We use imitation learning to accelerate the training process of our PPO agent and propose an MCTS policy optimization method to refine the learned policy.

II. Related Work

B&B [9] is one of the most general approaches for global optimization in nonconvex and combinatorial problems, which combines partial enumeration strategy with relaxation techniques. B&B maintains a provable upper and lower (primal and dual) bound on the optimal objective value and, hence, provides more reliable results than heuristic approaches. However, the B&B method can be slow depending on the selection of branching rules, which may grow the computational cost exponentially with the size of the problem [18].

Several attempts have been made to derive good branching strategies. Current branching strategies can be categorized into hand-designed approaches that make selections based on heuristic scoring functions; and statistical approaches that use machine learning models to approximate the scoring functions. Most modern MILP solvers use hand-designed branching strategies, including most infeasible branching [19], pseudocost branching (PC) [20], strong branching (SB) [21], [16], reliability branching (RB) [19], and more. Strong branching provides the local optimal solution with the highest computational cost by experimenting with all possible outcomes. Pseudocost branching keeps a history of the success of performed branchings to evaluate the quality of candidate variables, which provides a less accurate but computationally cheaper solution. Reliability branching integrates both strong and pseudocost branching to balance the selection quality and time.

Given the fact that strong branching decisions provide a minimum number of explored nodes among all other hand-designed branching strategies but have a high computational cost, several studies have come up with the idea of approximating and speeding up strong branching strategies using statistical approaches. In [22], a regressor is learned to predict estimated strong branching scores using offline data collected from similar instances. Similarly, a learning-to-rank algorithm that estimates the rank of variables can also provide reliable result [23], [24], which is more reliable than mimicking the score function. However, these statistical approaches suffer from extensive feature engineering.

One common approach to reducing the feature engineering effort is to use the graph convolutional neural network (GCNN). Reference [25] first proposed a GCNN model to solve combinatorial optimization problems, and reference [12] extended the structure to the context of B&B variable selection, which is the closest line of work to ours. In [12], authors show the GCNN can provide accurate estimation of strong branching with the shortest solving time in most of the considered instances.

However, most recent statistical approaches for variable selection in B&B use supervised learning techniques, which require a mapping between training inputs and expected labels. The quality of the model highly depends on the quality of training labels. As mentioned earlier, recent studies use strong branching scores or selections as training labels, which provides the local optimal solution, but is not guaranteed to be the global optimal solution. In general, we do not have access to optimal labels for most combinatorial optimization problems, and thus the supervised learning paradigm is not suitable in most cases [13]. Another approach is to learn through the interactions with an uncertain environment and provide some reward feedbacks to a learning algorithm. This is also known as the reinforcement learning (RL) paradigm. The RL algorithm makes a sequence of decisions and learns the decision-making policy through trial and error to maximize the long-term reward. Previous studies have shown that the combinatorial optimization problem can be solved using RL algorithms, such as the travelling salesman problem [13], [26], [27], maximum cut problem [28], [29], [30], and more. This study proposes a deep reinforcement learning framework to learn the global optimal variable selection strategy. We adopt the structure of GCNN as the design of our policy and value network.

III. Background

In this section, we describe the fundamental concepts related to the paper, and provide formal definitions to various terms.

A. Mixed integer linear program (MILP)

A MILP is a mathematical optimization problem that has a set of linear constraints, a linear objective function, and several decision variables that are continuous or integral with the form:

$$\arg\min_{\mathbf{x}} \mathbf{c}^T\mathbf{x}, \qquad \text{s.t.} \qquad \mathbf{A}\mathbf{x} \leq \mathbf{b}, \qquad \mathbf{l} \leq \mathbf{x} \leq \mathbf{u},$$
$$x_i \in \mathbb{Z} \text{ where } i \in \mathcal{I}, \qquad |\mathcal{I}| \leq n,$$

where $\mathbf{c}$ is the objective coefficient matrix, $\mathbf{x}$ is the variable vector, $\mathbf{A} \in \mathbb{R}^{m\times n}$ denotes the constraint coefficient matrix, $\mathbf{b} \in \mathbb{R}^m$ represents the constraint constant term vector, $\mathbf{l} \in (\mathbb{R} \cup \{-\infty\})^n$ and $\mathbf{u} \in (\mathbb{R} \cup \{\infty\})^n$ indicate the lower and upper variable bound vectors, respectively. Here n, m, and $\mathcal{I}$ respectively denote the number of variables, number of constraints, and index set of integer variables where $|\mathcal{I}| \leq n$. If a variable has no lower or upper bound, then we set the associated l and u to infinite values respectively. A *candidate* solution is any assignment of $\mathbf{x}$ that satisfy the variable bounds. A *feasible* solution is a candidate solution that satisfies all constraints in the MILP instance, and an *optimal* solution is a feasible solution that minimize the objective function.

A MILP can be relaxed to a linear program (LP) by ignoring the integer constraints in the MILP; this is also called *LP relaxation*. LP is convex and therefore can be solved efficiently using various algorithms, such as the simplex algorithm. Since removing the integer constraints expands the feasible set, the optimal solution for LP is then used as the lower bound for the corresponding MILP, namely the *dual bound*.

B. Branch-and-bound (B&B) algorithm

The B&B algorithm constructs a search tree recursively. Each node in the search tree is a MILP. The B&B algorithm can be described as follows. The original MILP is treated as the root node in the search tree. The algorithm then recursively picks a node from the search tree by a given node selection

rule, picks a variable to decompose the selected node, and adds two children to the selected node that are produced by the decomposition. The dual bound of these two children are then being used to update the dual bound of the root node, and the algorithm selects the next node to expand. To decompose a MILP on variable x_i, we first find the optimal solution $\mathbf{x}^*$ to the LP relaxation. Then, if $\mathbf{x}_i^*$ does not meet the integrality constraint, we can decompose the MILP into two sub-problems with additional constraints $x_i \leq \lfloor x_i^* \rfloor$ and $x_i \geq \lceil x_i^* \rceil$. The variable x_i is called the *branching variable*, and all variables that can be selected are called *branching candidates*.

1) Strong branching (SB): SB is one of the most powerful state-of-the-art variable selection rules. The idea of SB is to test which branching candidate can provides the best improvement measured in children nodes. This method is a greedy method that selects the locally best variable to branch on, which usually works well in terms of the number of nodes visited to solve the problem. However, it requires to branch on every branching candidates to calculate the score, which is computationally expensive. Moreover, this greedy approach cannot guarantee to provide the global optimal selection.

C. Markov decision process (MDP) formulation

We can formulate the sequential decision making of variable selection as a MDP. Each node in the search tree can be encoded as a state. The agent exerts a branching variable from all branching candidates to decompose the current node. This action causes a transition to a child node. Through interactions with the MDP, the algorithm learns an optimal *policy* π, that is a sequence of control actions starting from the root node.

1) State: The state s_t at node t, can be represented as:

$$s_t = \{(X, E, C)_t, J_t\},$$

where the first tuple is the bipartite graph representation $(X, E, C)_t$ of the current node MILP, as done in [12], and index set J_t is the index set of branching candidates. Two sets of nodes in the bipartite graph correspond to the n variable to be optimized and m constraints to meet. The edge $e_{i,j}$ is added if the variable x_i has a non-zero coefficient $A_{i,j}$ in the constraint c_j, where d_e features form the constraints constant term. $E \in \mathbb{R}^{m \times n \times d_e}$ represents the sparse edge feature matrix. $X \in \mathbb{R}^{n \times d_x}$ is a feature matrix for all variable nodes, including the features extracted from the objective function and variable constraints. Similarly, $C \in \mathbb{R}^{m \times d_c}$ represents the feature matrix for all constraint nodes, where each constraint is encoded into d_c features. Figure 1 illustrates the bipartite representation of a general MILP instance. We calculate the optimal solution of the current node's LP relaxation and mark variables with integer constraints and having a non-integer solution as the branching variable to get J_t.

2) Action and transition: The action at node t, denoted by a_t, determines the branching variable from the branching candidates: $a_t \in J_t$. After an action is performed, the search tree will add two children nodes to the current node and then prunes the search tree if needed, as described in Section III-B. All children share the same $p(s_{t+1}|s_t, a_t)$ in this study.

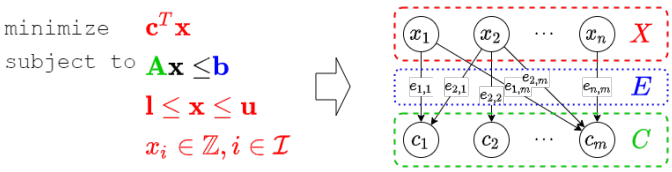


Fig. 1: Bipartite graph representation (X, E, C) of a MILP.

3) Reward: The reward function is designed to encourage the agent increase the dual bound quickly with as less branching operations as possible. Because we do not control the selection of state, and the global dual bound is highly related to the search tree constructed based on the selection of branching node at each step, using the improvement of global dual bound is not valid as the selected branching node might not be able to improve the global dual bound by any action. Therefore, we calculate the reward based on the improvement of the local dual bound:

$$r(s_t, a_t) = \min\{\mathbf{c}^T\mathbf{x}_{\lfloor \mathbf{t}' \rfloor}^*{}_{LP}, \mathbf{c}^T\mathbf{x}_{\lceil \mathbf{t}' \rceil}^*{}_{LP}\} - \mathbf{c}^T\mathbf{x}_{\mathbf{t}LP}^*,$$

where $\mathbf{x}_{\mathbf{t}LP}^*$ is the dual bound of the current node s_t, $\mathbf{x}_{\lfloor \mathbf{t}' \rfloor}^*{}_{LP}$ and $\mathbf{x}_{\lceil \mathbf{t}' \rceil}^*{}_{LP}$ are respectively the dual bound of children nodes after adding constraint $x_{a_t} \leq \lfloor x_{a_t}^* \rfloor$ and $x_{a_t} \geq \lceil x_{a_t}^* \rceil$ to s_t.

IV. Methodology

In this section, we discuss the design of the RL agent, techniques to address the cold-start problem, the training algorithm, and how to exploit the knowledge of a trained RL agent to select branching variables from a given set of branching candidates. Figure 2 shows an overview of our approach, which entails (1) designing the RL agent; (2) using imitation learning to pre-train the RL agent; (3) training the RL agent with PPO; (4) finally, selecting reliable branching variables for test environments using RL agent based on the search result of Monte-Carlo tree search (MCTS); We describe each of these tasks below.

A. Designing the RL agent

Reinforcement learning methods can find a policy that maximizes the total reward, especially when the MDP is identified. In this study, we use a policy gradient-based method called proximal policy optimization (PPO) to find the optimal policy π using the actor-critic framework. PPO has shown a strong performance in nearly all reinforcement learning tasks, thanks to the clipping method that limits the update of the behaviour policy within a trust region.

To evaluate the step size of the policy gradient method, PPO keeps tracking two policies, current policy π_θ and old policy $\pi_{\theta_{old}}$. Each policy contains two networks: a policy network that estimates the action distribution of a given state and a value network that estimates the state value. The state value $V(s_t)$ in this study is defined as follow:

$$V(s_t) = \sum_a p(a|s_t) \left(r(s_t, a) + \gamma \frac{V(\lfloor s_t' \rfloor) + V(\lceil s_t' \rceil)}{2} \right),$$

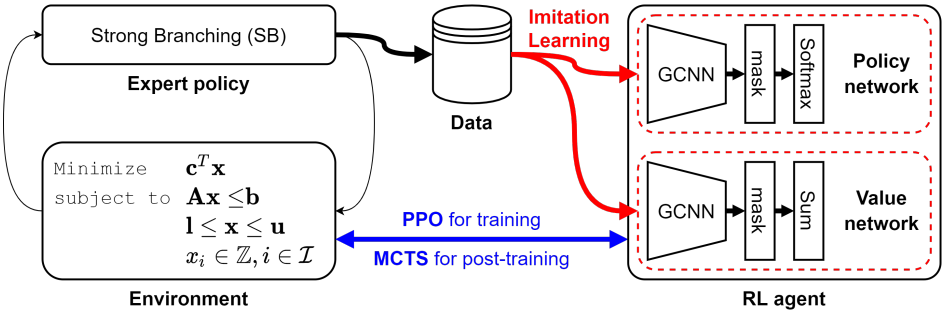


Fig. 2: The flow-diagram of the proposed approach.

where γ is a discount factor with value of 0.99 to encourage immediate reward, and states $\lfloor s'_t \rfloor$ and $\lceil s'_t \rceil$ are two children after branching on state s_t with variable x_a. This is the different from the next state we obtained through the interaction with the environment. That is being said, state s_t and s_{t+1} might not have an edge in the search tree, because the node selection rule pick the next state from all leaf nodes in the search tree. In the calculation of the state value, the next state must be the child of the current state, to therefore correctly represent the state value in the search tree. If s_t is a leaf node in the search tree, then the state value $V(s_t)$ is set to 0.

Because the state consists of a bipartite graph, we use graph convolutional neural network (GCNN) as our policy and value network. Previous studies also proved that GCNNs can effectively capture structural characteristics of combinatorial optimization problems. We adopt the similar GCNN design from [12], which use two successive passes to perform a single graph convolution. These passes are

$$c'_p \leftarrow f_{\mathrm{c}}\left(c_p, \sum_{q}^{(p,q)\in E} g_{\mathrm{c}}\left(\mathrm{emb}_{\mathrm{x}}(x_q), e_{p,q}, \mathrm{emb}_{\mathrm{c}}(c_p)\right)\right),$$

$$x'_q \leftarrow f_{\mathrm{x}}\left(x_q, \sum_{p}^{(p,q)\in E} g_{\mathrm{c}}\left(\mathrm{emb}_{\mathrm{x}}(x_q), e_{p,q}, c'_p\right)\right),$$

for all $p \in C, q \in X$. Next, the value of x' is sent to a 2-layer perceptron. For the policy network, we apply masked softmax activation to estimate the action distribution. For the value network, we compute masked sum to predict the state value.

B. Imitating the Strong Branching (SB)

Theoretically, the RL agent can find the optimal policy π from scratch after training for enough episodes. However, as the search tree is huge, with a branching factor usually over 1,000, training an RL agent from scratch becomes time-consuming and therefore impractical. To avoid the initial aimless exploration of the RL agent, we use the imitation learning approach to pretrain the RL agent policy and value network, paving the way for learning sophisticated policy. We select SB as our expert policy to generate offline training data, including the state, corresponding SB score for each branching candidate, as well as the reward. Then we reconstruct the state value $V(s_t)$ from the offline data and pretrain the policy and value network by minimizing the following loss:

$$L^{policy}(\theta) = -\frac{1}{N}\sum_{s,a\in\mathcal{D}} \log\pi_\theta(a|s)$$

$$L^{value}(\theta) = \frac{1}{N}\sum_{s\in\mathcal{D}} (V_\theta(s) - V(s))^2$$

C. Training the RL agent

Once the RL agent is pretrained using offline data, it is necessary to learn an advanced policy by interacting with the environment directly. To update the policy parameter θ with some trajectories generated through the interaction with the environment, we first save the parameters θ into θ_{old}, and then calculate the loss as follows:

$$A_t = V(s_t) - V_\theta(s_t), \qquad r = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)},$$

$$L_t^{policy}(\theta) = \mathbb{E}_t\left[\min\left(rA_t, \mathrm{clip}(r, 1-\epsilon, 1+\epsilon)A_t\right)\right],$$

$$L_t(\theta) = \mathbb{E}_t\left[L_t^{policy}(\theta) - c_1A_t^2 - c_2\sum_a \pi_\theta(a|s_t)\log\pi_\theta(a|s_t)\right],$$

where $V(s_t)$ is the state value calculated from the experiences, $V_\theta(s_t)$ is the value network estimated state value for s_t, and ϵ, c_1, c_2 are hyperparameters of the model. In this study, we use $\epsilon = 0.1, c_1 = 0.5, c_2 = 0.01$.

D. Enhancing policy with Monte-Carlo Tree Search (MCTS)

After we obtain the stable policy π^*_θ and V^*_θ, it is essential to make reliable selections for a given state s_t. One common and straightforward approach is to take action with the highest estimated action probability, $\mathrm{argmax}_a \pi^*_\theta(a|s_t)$. However, this result could be biased when the policy is not optimized or has a significant variance. Since we have a tree-like search space, it is possible to adopt the idea of MCTS to update the policy π^*_θ further. In MCTS, we generate multiple trajectories starting from the current node. Then, we expand trajectories by taking action based on some probability distribution until a certain number of steps or the final state is reached. Finally, we pick the best action based on these trajectories. This is similar to the SB, except MCTS does not explore all branching candidates.

To run MCTS efficiently, we incorporate the knowledge learned by our RL agent. In the action selection step for MCTS,

we use a modified version of upper confidence bound (UCB), which selects the action that maximizes the following equation:

$$\arg\max_{a\in\mathcal{A}(s)}(s,a)+c\pi_\theta^*(a|s)\sqrt{\frac{\log(1+\sum_a N(s,a))}{N(s,a)+1}}.$$

Here, the $Q(s,a)$ represents the action value based on the trajectories done previously, and $N(s,a)$ keeps tracking the number of times a has been selected on state s. We introduce trained policy π_θ^* to encourage the algorithm to search for promising directions. To minimize the size of the search tree, we further limit the branching candidates on each state s to $\mathcal{A}(s)$, which only contains the top k actions based on the $\pi_\theta^*(s)$. In this study, we use $k=10$.

Also, to reduce the simulation time, we do not perform the branching operation when we run MCTS. Instead, we directly modify the constraint feature matrix and edge feature matrix to simulate the next state s' based on the action, with half chance to reach the left child and half chance to reach the right child. We then set the reward of all actions to 0 and use the value network V_θ^* trained by the RL agent to calculate the value of $V_\theta^*(s')$, and use it to calculate the $Q(s,a)$. We initialize the action value and the visit count as follow:

$$Q(s,a)=\gamma V_\theta^*(s_t'),\qquad N(s,a)=1.$$

The $Q(s,a)$ and $N(s,a)$ are then updated when the agent reaches the leaf of the search tree or the maximum number of steps is reached. We apply the following update rule for each state $\{s_t,\cdots,s_0\}$:

$$Q(s_\tau,a_\tau)\leftarrow Q(s_\tau,a_\tau)+\frac{-Q(s_\tau,a_\tau)+\sum_{t'=\tau}^{t}\gamma^{t-\tau}V_\theta^*(s_t')}{N(s_\tau,a_\tau)+1}.$$

After all MCTS simulations are finished, we identify the action a with the highest $Q(s,a)$ as the best branching variable for each state s that has been visited at least ten times and use this to train the policy network by minimizing the cross-entropy loss. In this study, we limit the maximum depth to 3 and run 1,000 simulations of the MCTS for each state.

V. Evaluation

In this section, we study the efficacy of different variable selection strategies. We adopt the average solving time, average number of resulting B&B nodes, and average dual integral as our evaluation metrics. All experiments are repeated five times with different random seeds to eliminate randomness. All numbers are the averaged value across all five runs.

A. Data sets

To test the generalizability of our framework, we evaluate our approach on four different types of NP-hard problems. The first problem is called set covering problem proposed in [31]. Our instances contain 1,000 columns and 500 rows per instance. The second problem is generated following the arbitrary relationships procedure of [32]. This problem is also known as the combinatorial auction problem. In our experiment, we generate instances with 100 items for 500 bids. Our third data set is called capacitated facility location described in [33]. We collect instances with 100 facilities and 100 customers. The last data set we used in this study is proposed in [34], which is called the maximum independent set problem. The affinity is set to 4, and the graph size is set to 500 in this study.

These problems are selected based on the previous works and the hardness of the problem itself. According to [12], these problems are the most representative of the types of integer programming problems encountered in practice. We use SCIP 7.0.3 [35] as the backend solver throughout the study, with ecole 0.7.3 [36] as the environment interface. All SCIP parameters are kept to default in this study.

B. Baselines

In the rest of this paper, we use PPO-MCTS to refer to our proposed reinforcement learning framework. We compare our approach with three different variable selection baseline strategies. The first naive baseline strategy is the pure random strategy, in which we select the branching variable from a set of candidate variables uniformly. We use the full strong branching (FSB) strategy as our second baseline, which we use the default parameter defined in SCIP in this study. We also re-implemented the GCNN model from [12] as our third baseline. Based on the ML4CO NeurIPS 2021 competition result, the GCNN model yields the best performance among all other competing methods [37], [38], [39]. The performance of PPO agents that have no MCTS learning afterwards (PPO) is also reported for ablation study.

C. Evaluation metrics

We evaluate the performance of each approach using three metrics, including the average solving time for each problem instance, the average number of B&B search tree nodes visited before the problem is solved, and a reward score that takes into account both the solving time and the improvement of the dual bound. Solving time and the number of nodes visited measure the computational cost of each algorithm. Solving time is evaluated based on the wall clock time, including feature extraction time, model inference time, branching time, and more. Therefore, a shorter solving time does not guarantee to optimize the number of nodes visited during the B&B method. To optimize the branching variable selection strategy, we expect to minimize the number of nodes visited during the branching and the total solving time to select optimal branching variables with minimum computational cost. The score is calculated by:

$$-T\mathbf{c}^T\mathbf{x}^*+\int_{t=0}^{T}\mathbf{z}_t^*\partial t, \tag{1}$$

where $\mathbf{x}^*$ is the optimal solution of the MILP instance, T is the time budget to solve the problem, and $\mathbf{z}_t^*$ is the best dual bound at time t. This score is to be maximized, representing a fast improvement of the dual bound. This reward metric was first introduced in the ML4CO NeurIPS 2021 competition [37] and is expected to be adopted further by the community.

TABLE I: Number of resulting B&B nodes on the test data sets

	Set Covering	Independent Set	Combinatorial Auction	Capacitated Facility Location
Random	2225.20	257.60	25543.26	2292.24
FSB	47.42	103.85	***193.83***	***47.9***
GCNN	44.07	**88.66**	201.82	886.66
PPO-MCTS	***43.90***	90.23	194.25	863.21

TABLE II: MILP instance solving time (in seconds) on the test data sets

	Set Covering	Independent Set	Combinatorial Auction	Capacitated Facility Location
Random	19.2	15.36	99.08	92.28
FSB	95.8	240.65	113.58	864.75
GCNN	3.28	***6.54***	4.15	80.68
PPO-MCTS	***3.13***	6.60	***3.87***	***77.24***

TABLE III: Evaluation score on the test data sets

	Set Covering	Independent Set	Combinatorial Auction	Capacitated Facility Location
FSB	149930	-191876	-7093620	16119158
GCNN	**150654**	**-191123**	-7077028	16159789
PPO-MCTS	150652	-191139	**-7076023**	**16160324**

TABLE IV: Performance comparison between PPO and PPO-MCTS

	Nodes visit		Solving time		Evaluation score	
	PPO	PPO-MCTS	PPO	PPO-MCTS	PPO	PPO-MCTS
Set Covering	57.68	**43.90**	3.52	**3.13**	150651	**150652**
Independent Set	**88.18**	90.23	8.34	**6.60**	-203328	**-191139**
Combinatorial Auction	270.97	**194.25**	5.28	**3.87**	-7077229	**-7076023**
Capacitated Facility Location	2202.46	**863.21**	139.65	**77.24**	16155103	**16160324**

D. Experiment result

Table I shows the average number of nodes visited before solving the instances for each approach. We noticed both GCNN and our PPO-MCTS have more nodes visited in complex problems, namely the combinatorial auction and capacitated facility location problems, compared to FSB. We conclude this to the fact that the number of branching candidates in these two problems are more significant than the other two problem types, and therefore leads to the approximate function getting more complex, which lowers the performance of the GCNN model. Similarly, as the search tree branching factor grows, RL agents become more challenging to learn the environment thoroughly. The agent can potentially struggle in local optimal as the maximum depth is set to three for our PPO-MCTS agent. It is worth noting that number of nodes on its own is not enough of a measure to judge different approaches with. There reason is that a method may visit a larger number of nodes, but may in fact be faster on each visit and result a better overall reward value for convergence.

On the other hand, it is readily seen from Table II that FSB takes a significant amount of time to select a variable for each node in these two problems, and therefore the total solving time for FSB is the longest compared to all other approaches in all four problems. The GCNN and PPO-MCTS are having similar inference time as the network designs are similar. As the PPO-MCTS has a lower average number of visited nodes in all but independent set problems, our method provides the shortest solving time in all problems except the independent set problem. However, the performance differences between GCNN and PPO-MCTS on independent set problem in all three metrics are negligible, which proves the effectiveness of our proposed framework. In addition, when the problem is easy, such as the set covering and independent set problems, both GCNN and PPO-MCTS can find better branching variables with fewer nodes visited than FSB. In general, PPO-MCTS has a slightly better performance across different data sets and metrics than GCNN, with a trade-off on the training expenses.

The average evaluation score for each approach is shown in Table III. The GCNN and PPO-MCTS approaches keep dominating the score in all problems, whereas the PPO-MCTS has a higher score on challenging problems, and GCNN performs better with easy problems.

E. Ablation study

We present an ablation study of the proposed PPO-MCTS model to evaluate the importance of having post MCTS retraining. Table IV demonstrates the performance of PPO without post MCTS retraining on all metrics across all data sets, denoted by PPO, comparing with the proposed PPO-MCTS. It is observed that the PPO-MCTS perform better than PPO in all cases, except the number of nodes visited in the independent set problem. This empirical result suggests that the post MCTS retraining offers a better performing RL agent.

VI. Conclusion

We proposed a reinforcement learning framework to learn the variable selection policy for the B&B method. We formulated a reward function that helps the agent learn optimal policies without generating labels. We used imitation learning and MCTS to deal with sample inadequacy challenges by initializing the policy to a relatively good policy and enhancing it with multiple steps look-ahead. We demonstrated the performance of the proposed framework with three baseline approaches on four NP-hard problems and showed that the proposed method yields a strong performance in most problems.

REFERENCES

[1] João CN Clímaco, Marta MB Pascoal, José MF Craveirinha, and M Eugénia V Captivo. Internet packet routing: Application of a k-quickest path algorithm. *European Journal of Operational Research*, 181(3):1045–1054, 2007.

[2] Yves Crama. Combinatorial optimization models for production scheduling in automated manufacturing systems. *European Journal of Operational Research*, 99(1):136–153, 1997.

[3] Mehul M Khimasia and Peter V Coveney. Protein structure prediction as a hard optimization problem: the genetic algorithm approach. *Molecular Simulation*, 19(4):205–226, 1997.

[4] Tuomas Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artificial intelligence*, 135(1-2):1–54, 2002.

[5] Pedro Larranaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial intelligence review*, 13(2):129–170, 1999.

[6] Moustapha Diaby. Linear programming formulation of the vertex colouring problem. *International Journal of Mathematics in Operational Research*, 2(3):259–289, 2010.

[7] Paolo Toth and Daniele Vigo. *The vehicle routing problem*. SIAM, 2002.

[8] Bruce L Golden, Subramanian Raghavan, and Edward A Wasil. *The vehicle routing problem: latest advances and new challenges*, volume 43. Springer Science & Business Media, 2008.

[9] Ailsa H Land and Alison G Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[10] Eugene L Lawler and David E Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[11] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, et al. Miplib 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, pages 1–48, 2021.

[12] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *arXiv preprint arXiv:1906.01629*, 2019.

[13] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[14] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, page 105400, 2021.

[15] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[16] Jeff T Linderoth and Martin WP Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2):173–187, 1999.

[17] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[18] Stephen Boyd and Jacob Mattingley. Branch and bound methods. *Notes for EE364b, Stanford University*, pages 2006–07, 2007.

[19] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.

[20] Michel Bénichou, Jean-Michel Gauthier, Paul Girodet, Gerard Hentges, Gerard Ribière, and O Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1):76–94, 1971.

[21] Jan Karel Lenstra and David Shmoys. The traveling salesman problem: A computational study, 2009.

[22] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195, 2017.

[23] Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[24] Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. *arXiv preprint arXiv:1805.01415*, 2018.

[25] Hanjun Dai, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*, 2017.

[26] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711. PMLR, 2016.

[27] Hao Lu, Xingwen Zhang, and Shuang Yang. A learning-based iterative method for solving vehicle routing problems. In *International Conference on Learning Representations*, 2019.

[28] Thomas Barrett, William Clements, Jakob Foerster, and Alex Lvovsky. Exploratory combinatorial optimization with reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3243–3250, 2020.

[29] Quentin Cappart, Emmanuel Goutierre, David Bergman, and Louis-Martin Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1443–1451, 2019.

[30] Kenshin Abe, Zijian Xu, Issei Sato, and Masashi Sugiyama. Solving np-hard problems on graphs with extended alphago zero. *arXiv preprint arXiv:1905.11623*, 2019.

[31] Egon Balas and Andrew Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. In *Combinatorial Optimization*, pages 37–60. Springer, 1980.

[32] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76, 2000.

[33] Gérard Cornuéjols, Ranjani Sridharan, and Jean-Michel Thizy. A comparison of heuristics and relaxations for the capacitated plant location problem. *European journal of operational research*, 50(3):280–297, 1991.

[34] David Bergman, Andre A Cire, Willem-Jan Van Hoeve, and John Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[35] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, et al. The scip optimization suite 7.0. 2020.

[36] Antoine Prouvost, Justin Dumouchelle, Lara Scavuzzo, Maxime Gasse, Didier Chételat, and Andrea Lodi. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

[37] ecole.ai. ML4CO: 2021 neurips competition on machine learning for combinatorial optimization. *https://www.ecole.ai/2021/ml4co-competition/*.

[38] Zixuan Cao, Yang Xu, Zhewei Huang, and Shuchang Zhou. Ml4co-kida: Knowledge inheritance in dataset aggregation. *arXiv preprint arXiv:2201.10328*, 2022.

[39] Amin Banitalebi-Dehkordi and Yong Zhang. Ml4co: Is gcnn all you need? graph convolutional neural networks produce strong baselines for combinatorial optimization problems, if tuned and trained properly, on appropriate data. *arXiv preprint arXiv:2112.12251*, 2021.

Learning Optimal Decision Trees Using Caching Branch-and-Bound Search

Gaël Aglin, Siegfried Nijssen, Pierre Schaus
firstname.lastname@uclouvain.be
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium

Abstract

Several recent publications have studied the use of Mixed Integer Programming (MIP) for finding an optimal decision tree, that is, the best decision tree under formal requirements on accuracy, fairness or interpretability of the predictive model. These publications used MIP to deal with the hard computational challenge of finding such trees. In this paper, we introduce a new efficient algorithm, DL8.5, for finding optimal decision trees, based on the use of itemset mining techniques. We show that this new approach outperforms earlier approaches with several orders of magnitude, for both numerical and discrete data, and is generic as well. The key idea underlying this new approach is the use of a cache of itemsets in combination with branch-and-bound search; this new type of cache also stores results for parts of the search space that have been traversed partially.

Introduction

Decision trees are among the most widely used machine learning models. Their success is due to the fact that they are simple to interpret and that there are efficient algorithms for learning trees of acceptable quality.

The most well-known algorithms for learning decision trees, such as CART (Breiman et al. 1984) and C4.5 (Quinlan 1993), are greedy in nature: they grow the decision tree top-down, iteratively splitting the data into subsets.

While in general these algorithms learn models of good accuracy, their greedy nature, in combination with the NP-hardness of the learning problem (Laurent and Rivest 1976), implies that the trees that are found are not necessarily optimal. As a result, these algorithms do not ensure that:

- the trees found are the most accurate for a given limit on the depth of the tree; as a result, the paths towards decisions may be longer and harder to interpret than necessary;
- the trees found are the most accurate for a given lower bound on the number of training examples used to determine class labels in the leaves of the tree;
- the trees found are accurate while satisfying additional constraints such as on the *fairness* of the trees: in their predictions, the trees may favor one group of individuals over another.

With the increasing interest in explainable and fair models in machine learning, recent years have witnessed a renewed interest in alternative algorithms for learning decision trees that can provide such optimality guarantees.

Most attention has been given in recent years and in prominent venues to approaches based on mixed integer programming (Bertsimas and Dunn 2017; Verwer and Zhang 2019; Aghaei, Azizi, and Vayanos 2019). In these approaches, a limit is imposed on the depth of the trees that can be learned and a MIP solver is used to find the optimal tree under well-defined constraints.

However, earlier algorithms for finding optimal decision trees under constraints have been studied in the literature, of which we consider the DL8 algorithm of particular interest (Nijssen and Fromont 2007; 2010). The existence of this earlier work does not appear to have been known to the authors of the more recent MIP-based approaches, and hence, no comparison with this earlier work was carried out.

DL8 is based on a different set of ideas than the MIP-based approaches: it treats the paths of a decision tree as *itemsets*, and uses ideas from the itemset mining literature (Agrawal et al. 1996) to search through the space of possible paths efficiently, performing dynamic programming over the itemsets to construct an optimal decision tree. Compared to the MIP-based approaches, which most prominently rely on a constraint on depth, DL8 stresses the use of a minimum support constraint to limit the size of the search space. It was shown to support a number of different optimization criteria and constraints that do not necessarily have to be linear.

In this paper, we present a number of contributions. We will demonstrate that DL8 can also be applied in settings in which MIP-based approaches have been used; we will show that, despite its age, it outperforms the more modern MIP-based approaches significantly, and is hence an interesting starting point for future algorithms.

Subsequently, we will present DL8.5, an improved version of DL8 that outperforms DL8 by orders of magnitude. Compared to DL8, DL8.5 adds a number of novel ideas:

- it uses branch-and-bound search to cut large additional parts of the search space;
- it uses a novel caching approach, in which we store also store information for itemsets for which the search space has been cut; this allows us to avoid redundant computation later on as well;
- we consider a range of different branching heuristics to find good trees more rapidly;
- the algorithm has been made any-time, i.e. it can be stopped at any time to report the best tree it has found so far.

In our experiments we focus our attention on traditional decision tree learning problems with little other constraints, as we consider these learning problems to be the hardest. However, we will show that DL8.5 remains sufficiently close to DL8 that the addition of other constraints or optimization criteria is straightforward.

In a recent MIP-based study, significant attention was given to the distinction between binary and numerical data (Verwer and Zhang 2019). We will show that DL8.5 outperforms this method on both types of data.

Our implementation is publicly available at https://github.com/aglingael/dl8.5 and can easily be used from Python.

This paper is organized as follows. The next section presents the state of the art of optimal decision trees induction. Then we present the background on which our work relies, before presenting our approach and our results.

Related work

In our discussion of related work, we will focus our attention on alternative methods for finding *optimal* decision trees, that is, decision trees that achieve the best possible score under a given set of constraints.

Most attention has been given in recent years to MIP-based approaches. Bertsimas and Dunn (2017) developed an approach for finding decision trees of a maximum depth K that optimize misclassification error. They use K to model the problem in a MIP model with a fixed number of variables; a MIP solver is then used to find the optimal tree.

Verwer and Zhang (2019) proposed *BinOCT*, an optimization of this approach, focused on how to deal with numerical data. To deal with numerical data, decision trees need to identify thresholds that are used to separate examples from each other. A MIP model was proposed in which fewer variables are needed to find high-quality thresholds; consequently, it was shown to work better on numerical data.

A benefit of MIP-based approaches is that it is relatively easy from a modeling perspective to add linear constraints or additional linear optimization criteria. Aghaei, Azizi, and Vayanos (2019) exploit this to formalize a learning problem that also takes into account the *fairness* of a prediction.

Verhaeghe et al. (2019) recently proposed a Constraint Programming (CP) approach to solve the same problem. It supports a maximum depth constraint and a minimum support constraint, but only works for binary classification tasks. It also relies on branch-and-bound search and caching, but uses a less efficient caching strategy. The approach in the present paper is easily implemented and understood without relying on CP systems.

Another class of methods for learning optimal decision trees is that based on SAT Solvers (Narodytska et al. 2018; Bessiere, Hebrard, and O'Sullivan 2009). SAT-based studies, however, focus on a different type of decision tree learning problem than the MIP-based approaches: finding a decision tree of limited size that performs 100% accurate predictions on training data. These approaches solve this problem by creating a formula in conjunctive normal form, for which a satisfying assignment would represent a 100% accurate decision tree. We believe there is a need for algorithms that minimize the error, and hence we focus on this setting.

Most related to this work is the work of Nijssen and Fromont (2007; 2010) on DL8, which relies on a link between learning decision trees and itemset mining. Similarly to MIP-based approaches, DL8 allows to find optimal decision trees minimizing misclassification error. DL8 does not require a depth constraint; it does however assume the presence of a minimum support constraint, that is, a constraint on the minimum number of examples falling in each leaf. In the next section we will discuss this approach in more detail. This discussion will show that DL8 can easily be used in settings identical to those in which MIP and CP solvers have been used. Subsequently, we will propose a number of significant improvements, allowing the itemset-based approach to outperform MIP-based and CP-based approaches.

Background

Itemset mining for decision trees

We limit our discussion to the key ideas behind DL8, and assume that the reader is already familiar with the general concepts behind learning decision trees.

DL8 operates on Boolean data. As running example of such data, we will use the dataset of Table 1a, which consists of three Boolean features and eleven examples. The optimal decision tree for this database can be found in Figure 1a.

While it may seem a limitation that DL8 only operates on Boolean data, there are straightforward ways to transform any tabular database in a Boolean database: for categorical attributes, we can create a Boolean column for each possible value of that attribute; for numerical attributes, we can create Boolean columns for possible thresholds for that attribute.

DL8 takes an itemset mining perspective on learning decision trees. In this perspective, the binary matrix of Table 1a is transformed into the transactional database of Table 1b. Each transaction of the dataset contains an itemset describing the presence or absence of each feature in the dataset. More formally, the database $\mathcal{D}$ can be thought of as a collection $\mathcal{D} = \{(t, I, c) \,|\, t \in \mathcal{T}, I \subseteq \mathcal{I}, c \in \mathcal{C}\}$, where $\mathcal{T}$ represents the transaction or rows identifiers, $\mathcal{I}$ is the set of possible items, and $\mathcal{C}$ is the set of class labels; within $\mathcal{I}$ there are two items (one *positive*, the other *negative*) for each original Boolean feature, and each itemset I contains either a positive or a negative item for every feature.

Using this representation, every path in a decision tree can be mapped to an itemset $I \subseteq \mathcal{I}$, as shown in Figure 1b. For instance, the last path in this tree corresponds to the itemset

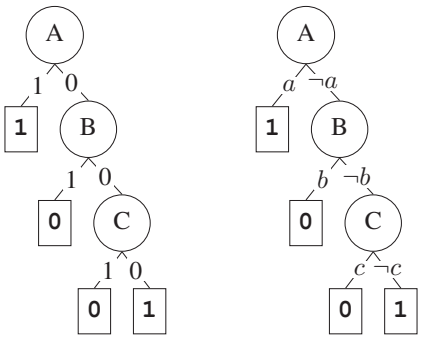
A	B	C	class
0	1	1	0
1	0	1	1
0	0	1	1
0	1	0	0
1	0	0	1
0	0	0	0
0	0	1	0
1	1	0	1
0	0	0	1
0	0	1	0
0	0	0	1

(a) Binary matrix

rowId	items	class
1	$\neg a, b, c$	0
2	$a, \neg b, c$	1
3	$\neg a, \neg b, c$	1
4	$\neg a, b, \neg c$	0
5	$a, \neg b, \neg c$	1
6	$\neg a, \neg b, \neg c$	0
7	$\neg a, \neg b, c$	0
8	$a, b, \neg c$	1
9	$\neg a, \neg b, \neg c$	1
10	$\neg a, \neg b, c$	0
11	$\neg a, \neg b, \neg c$	1

(b) Transactional database

Table 1: Example database



(a) Common view (b) Itemset view

Figure 1: Optimal tree corresponding to database of Table 1. Max depth = 3 and minimum examples per leaf = 1

$\{\neg a, \neg b, \neg c\}$. Please note that multiple paths can be mapped to the same itemset.

For every itemset I, we define its cover to be $cover(I) = \{(t', I', c') \in \mathcal{D} \mid I \subseteq I'\}$: the set of transactions in which the itemset is contained; the class-based support of an itemset is defined as $Sup(I, c) = |\{(t', I', c') \in cover(I) \mid c' = c\}|$, and can be used to identify the number of examples for a given class c in a leaf. Based on class-based supports, the error of an itemset is defined as:

$$leaf\_error(I) = \mid cover(I) \mid - \max_{c \in \mathcal{C}} \{Sup(I, c)\} \qquad (1)$$

Unlike the CP-based approach, our error function is also valid for classification tasks involving more than 2 classes.

The canonical decision tree learning problem that we study in this work can now be defined as follows using itemset mining notation. Given a database $\mathcal{D}$, we wish to identify a collection $\mathcal{DT} \subseteq \mathcal{I}$ of itemsets such that

- the itemsets in $\mathcal{DT}$ represent a decision tree;
- $\sum_{I \in \mathcal{DT}} leaf\_error(I)$ is minimal;
- for all $I \in \mathcal{DT}$: $|I| \leq maxdepth$, where $maxdepth$ is the maximum depth of the tree;
- for all $I \in \mathcal{DT}$: $|cover(I)| \geq minsup$, where $minsup$ is a minimum support threshold.

As stated earlier, in DL8, the maximum depth constraint is not required; MIP-based approaches have ignored the minimum support constraint.

Algorithm 1: $DL8(maxdepth, minsup)$

```
1  struct BestTree{ tree : Tree; error : float }
2  cache ← HashSet < Itemset, BestTree >
3  (τ, b) ← DL8 − Recurse(∅)
4  return τ
5  Procedure DL8 − Recurse(I)
6      solution ← cache.get(I)
7      if solution was found then
8          return (solution.tree, solution.error)
9      if leaf_error(I) = 0 or |I| = maxdepth then
10         return (make_leaf(I), leaf_error(I))
11     (τ, b) ← (make_leaf(I), leaf_error(I))
12     for all attributes i do
13         if |cover(I ∪ {i})| ≥ minsup and
             |cover(I ∪ {¬i})| ≥ minsup then
14             (τ1, e1) ← DL8 − Recurse(I ∪ {¬i})
15             if e1 ≤ b then
16                 (τ2, e2) ← DL8 − Recurse(I ∪ {i})
17                 if e1 + e2 ≤ b then
18                     (τ, b) ← (make_tree(i, τ1, τ2),
                         e1 + e2)
19     cache.store(I, BestTree(τ, b))
20     return (τ, b)
```

DL8 Algorithm

A high-level perspective of the DL8 algorithm is given in Algorithm 1. Essentially, the algorithm recursively enumerates itemsets using the $\texttt{DL8} - \texttt{Recurse}(I)$ function. The post-condition of this function is that it returns the optimal decision tree for the transactions covered by itemset I, together with the quality of that tree. This optimal tree is calculated recursively using the observation that the best decision tree for a set of transactions can be obtained by considering all possible ways of partitioning the set of transactions into two, and determining the best tree for each partition recursively.

Figure 2 illustrates the search space of itemsets for the dataset of Table 1, where all the possible itemsets are represented. Intuitively, DL8 starts at the top node of this search space, and calculates the optimal decision tree for the root based on its children.

A distinguishing feature of DL8 is its use of a cache. The idea behind this cache is to store the result of a call to $\texttt{DL8} - \texttt{Recurse}$. Doing so is effective as the same itemset can be reached by multiple paths in the search space: itemset ab can be constructed by adding b to itemset a, or by adding a to itemset b. By storing the result, we can reuse the same result for both paths.

Note that the optimal decision tree for the root can only be calculated after all its children have been considered; hence, the algorithm will only produce a solution once the entire search space of itemsets has been considered.

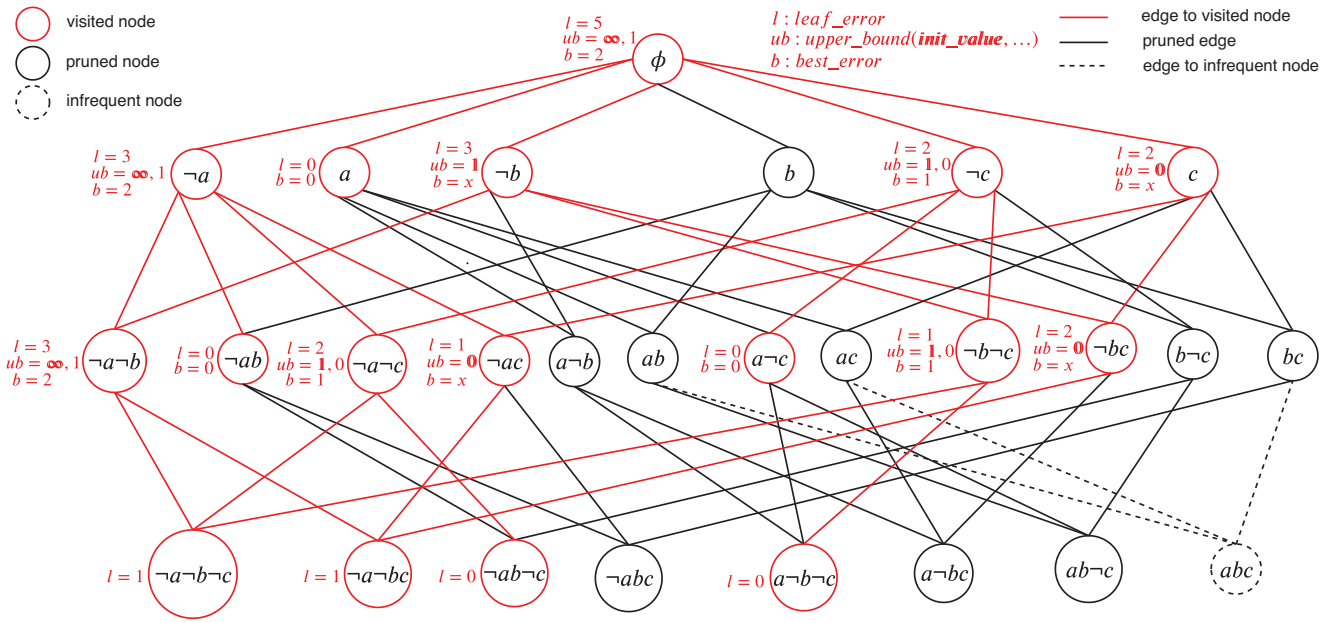


Figure 2: Complete itemset lattice for introduction database and DL8.5 search execution

In our pseudocode, we use the following other functions. Function $make\_leaf(I)$ returns a decision tree with one node, representing a leaf that predicts the majority class for the examples covered by I. The function $make\_tree(i, \tau_1, \tau_2)$ returns a tree with a test on attribute i, and subtrees τ_1 and τ_2.

The code illustrates a number of optimizations implemented in DL8:

Maximum depth pruning In line 9 the search is stopped as soon as the itemsets considered are too long;

Minimum support pruning In line 13 an attribute is not considered if one of its branches has insufficient support; in our running example, the itemset $\{a, b, c\}$ is not considered due to this optimization;

Purity pruning In line 9 the search is stopped if the error for the current itemset is already 0;

Quality bounds In the loop of lines 12–18, the best solution found among the children is maintained, and used to prune the second branch for an attribute if the first branch is already worse than the best solution found so far.

We omit a number of optimizations in this pseudo-code that can be found in the original publication, in particular, optimizations that concern the incremental maintenance of data structures. While we will use most of these optimizations in our implementation as well, we do not discuss these in detail here for reasons of simplicity.

The most important optimization in DL8 that we do not use in this study is the *closed itemset mining* optimization. The reason for this choice is that this optimization is hard to combine with a constraint on the depth of a decision tree. Similarly, while DL8 can be applied to other scoring functions than error, as long as the scoring function is *additive*, we prioritize accuracy and the depth constraint here as we focus on solving the same problem as in recent MIP-based studies.

Our approach: DL8.5

As identified in the introduction, DL8 has a number of weaknesses, which we will address in this section.

The most prominent of these weaknesses is that the size of the search tree considered by DL8 is unnecessarily large. Reconsider the example of Figure 2, in which DL8's pruning approach does not prune any node except from one infrequent itemset (abc). We will see in this section that a new type of caching brand-and-bound search can reduce the number of itemsets considered significantly.

The pseudo-code of our new algorithm, DL8.5, is presented in Algorithm 2. DL8.5 inherits a number of ideas from DL8, including the use of a cache, the recursive traversal of the space of itemsets, and the use of depth and support constraints to prune the search space.

The main distinguishing feature of DL8.5 concerns its use of bounds during the search.

In DL8.5, the recursive procedure `DL8.5 − Recurse` has an additional parameter, $init\_ub$, which represents an upper-bound on the quality of the decision trees that the recursive procedure is expected to find. If no sufficiently good tree can be found, the procedure returns a tree of type `NO_TREE`.

Initially, the upper-bound that is used is $+\infty$ (line 3). However, as soon as the recursive algorithm has found one decision tree, or has found a better tree than earlier known, the quality of this decision tree, calculated in line 21, is used as upper-bound for future decision trees and is communicated to the children in the search tree (line 25, line 14, 18).

The upper-bound is used to prune the search space using a test in line 17; intuitively, as soon as we have traversed one

Algorithm 2: $DL8.5(maxdepth, minsup)$

```
1  struct BestTree {init_ub : float; tree : Tree;
     error : float}
2  cache ← HashSet < Itemset, BestTree >
3  bestSolution ← DL8.5 − Recurse(∅, +∞)
4  return bestSolution.tree
5  Procedure DL8.5 − Recurse (I, init_ub)
6      if leaf_error(I) = 0 or |I| = maxdepth or
        time-out is reached then
7          return BestTree(init_ub, make_leaf(I),
            leaf_error(I))
8      solution ← cache.get(I)
9      if solution was found and ((solution.tree ≠
        NO_TREE) or (init_ub ≤ solution.init_ub)) then
10         return solution
11     (τ, b, ub) ← (NO_TREE, +∞, init_ub)
12     for all attributes i in a well-chosen order do
13         if |cover(I ∪ {i})| ≥ minsup and
            |cover(I ∪ {¬i})| ≥ minsup then
14             sol_1 ← DL8.5 − Recurse(I ∪ {¬i}, ub)
15             if sol_1.tree = NO_TREE then
16                 continue
17             if sol_1.error ≤ ub then
18                 sol_2 ← DL8.5 − Recurse(I ∪ {i},
                    ub − sol_1.error)
19                 if sol_2.tree = NO_TREE then
20                     continue
21                 feature_error ←
                    sol_1.error + sol_2.error
22                 if feature_error ≤ ub then
23                     τ ← make_tree(i, sol_1.tree,
                        sol_2.tree)
24                     b ← feature_error
25                     ub ← b − 1
26                 if feature_error = 0 then
27                     break
28     solution ← BestTree(init_ub, τ, b)
29     cache.store(I, solution)
30     return solution
```

branch for an attribute, and the quality of that branch is already worse than accepted by the bound, we do not consider the second branch for that attribute.

In line 18 we use the quality of the first branch to bound the required quality of the second branch further.

An important modification involves the interaction of the bounds with the cache. In DL8.5, we store an itemset also if no solution could be found for the given bound (line 29 is still executed even if the earlier loop did not find a tree). In this case, the special value NO_TREE is associated with the itemset in the cache, and the upper-bound used during the last search is stored.

The benefit of doing so is that at a later moment, we may

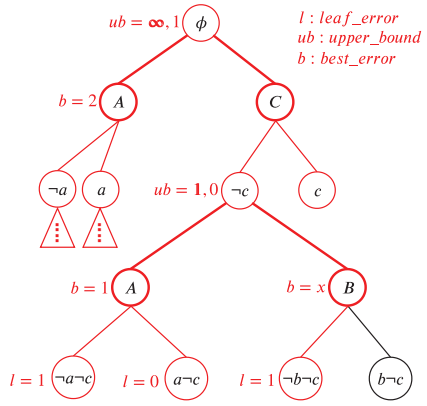


Figure 3: Example of pruning

reuse the fact that for a given itemset and bound, no sufficiently good decision tree can be found. In particular, in line 9, when the current bound ($init\_ub$) is worse than the stored upper-bound for a NO_TREE itemset, we return the NO_TREE indicator immediately.

Other modifications in comparison with DL8 improve the anytime behavior of the algorithm. In line 6 the search can be interrupted when a time-out is reached, and line 12 offers the possibility to consider the attributes in a specific heuristic order to discover good trees more rapidly.

A number of different heuristics could be considered. In our experiments, we consider three: the original order of the attributes in the data, in increasing or in decreasing order of information gain (such as used in C4.5 and CART).

Our modifications of DL8 improve drastically the pruning of the search space. Figure 2 indicates which additional nodes are pruned during the execution of DL8.5 (for an alphabetic order of the attributes). At the end, 17 nodes are visited instead of 27.

Figure 3 shows a part of the execution of DL8.5 in more detail. The initial value of the upper-bound at node ϕ is $+\infty$ (line 11). The attribute A provides an error of 2; the upper-bound value is subsequently updated from $+\infty$ to 1 in line 25. In the first branch for attribute C, the new value of the upper-bound is passed down recursively (line 14). Notice that the initial value of the upper-bound at node $\neg c$ is 1. At this node, the attribute A is first visited and provides an error of 1 by summing errors of $\neg a\neg c$ and $a\neg c$ (line 21). The upper-bound for subsequent attributes is then updated to 0 and passed down recursively to the first branch of attribute B. After visiting the first item $\neg b\neg c$ the obtained error is 1 and greater than the upper-bound of 0. The second item is pruned as the condition of line 18 is not satisfied. So, there is no solution by selecting the attribute B, which leads to storing the value NO_TREE for this itemset. This error value is represented in Figures 2 and 3 by the character x.

The reuse of the cache is illustrated for itemset $\neg ac$ (Figure 2). The first time we encounter this itemset, we do so coming from the itemset $\neg a$ for an upper-bound of zero; after the first branch, we observe that no solution can be found for this bound, and we store NO_TREE for this itemset. The

second time we encounter $\neg ac$, we do so coming from the parent c, again with an upper-bound of 0. From the cache we retrieve the fact that no solution could be found for this bound, and we skip attribute A from further consideration.

Results

In our experiments we answer the following questions:

Q1 How does the performance of DL8.5 compare to DL8, MIP-based and CP-based approaches on binary data?

Q2 What is the impact of different branching heuristics on the performance of DL8.5?

Q3 What is the impact of caching incomplete results in DL8.5 (`NO_TREE` itemsets)?

Q4 How does the performance of DL8.5 compare to DL8, MIP-based and CP-based approaches on continuous data?

As a representative MIP-based approach, we use BinOCT, as it was shown to be the best performing MIP-based approach in a recent study (Verwer and Zhang 2019). The implementations of BinOCT[1], DL8 and the CP-based approach[2] used in our comparison were obtained from their original authors, and use the CPLEX 12.9[3] and OscaR [4] solvers. Experiments were performed on a server with an Intel Xeon E5-2640 CPU, 128GB of memory, running Red Hat 4.8.5-16.

To respect the constraint of the CP-based algorithm all the datasets used in our experiments have binary classes. We compare our algorithms on 24 binary datasets from CP4IM[5], described in the first columns of Table 2.

Similar to Verwer and Zhang (2019), we run the different algorithms for 10 minutes on each dataset and for a maximum depth of 2, 3 and 4. All the tests are run with a minimum support of 1 since this is the setting used in BinOCT.

We do not split our datasets in training and test sets since the focus of this work is on comparing the computational performance of algorithms that should generate decision trees of the same quality. The benefits of optimal decision trees were discussed in (Bertsimas and Dunn 2017).

We compare a number of variants of DL8.5. The following table summarizes the abbreviations used.

Abbreviation	Meaning
d.o.	the original order of the attributes in the data is used as branching heuristic
asc	attributes are sorted in increasing value of information gain
desc	attributes are sorted in decreasing value of information gain
n.p.s.	no partial solutions are stored in the cache

Table 2 shows the results for a maximum depth equal to 4, as we consider deeper decision trees of more interest. If optimality could not be proven within 10 minutes, this is indicated using *TO*; in this case, the objective value of the best tree found so far is shown. Note that we here exploit the ability of DL8.5 to produce a result after a time-out. The best solutions and best times are marked in bold while a star (*) is added to mark solutions proven to be optimal.

BinOCT solved and proved optimality for only 1 instance within the timeout; the older DL8 algorithm solved 7 instances and the CP-based algorithm solved 11 instances. DL8.5 solved 19 (which answers **Q1**). The difference in performance is further illustrated in Figure 4, which gives *cactus plots* for each algorithm, for different depth constraints. In these plots each point (x, y) indicates the number of instances (x) solved within a time limit (y). While for lower depth thresholds, BinOCT does find solutions, the performance of all variants of DL8.5 clearly remains superior to that of DL8, BinOCT and the CP-based algorithm, obtaining orders of magnitude better performance.

Comparing the different branching heuristics in DL8.5, the differences are relatively small; however, for deeper trees, a descending order of information gain gives slightly better results. This confirms the intuition that chosen a split with high information gain is a good heuristic (**Q2**).

If we disable DL8.5's ability to cache incomplete results, we see a significant degradation in performance. In this variant only 12 instances are solved optimally, instead of 19, for a depth of 4. Hence, this optimization is significant (**Q3**).

To answer **Q4**, we repeat these tests on continuous data. For this, we use the same datasets as Verwer and Zhang (2019). These datasets were obtained from the UCI repository[6] and are summarized in the first columns of Table 3. Before running DL8, the CP-based algorithm and DL8.5, we binarize these datasets by creating binary features using the same approach as the one used by Verwer and Zhang (2019). Note that the number of generated features is very high in this case. As a result, for most datasets all algorithms reach a time-out for maximum depths of 3 and 4, as was also shown by Verwer and Zhang (2019). Hence, we focus on results for a depth of 2 in Table 3. Even though BinOCT uses a specialized technique for solving continuous data, the table shows that DL8.5 outperforms DL8, the CP-based algorithm and BinOCT. Note that the differences between the different variations of DL8.5 are small here, which may not be surprising given the shallowness of the search tree considered.

Conclusions

In this paper we presented the DL8.5 algorithm for learning optimal decision trees. DL8.5 is based on a number of ideas: the use of itemsets to represent paths, the use of a cache to store intermediate results (including results for parts of the search tree that have only been traversed partially), the use of bounds to prune the search space, the ability to use heuristics during the search, and the ability to return a result even when a time-out is reached.

Our experiments demonstrated that DL8.5 outperforms existing approaches by orders of magnitude, including approaches presented recently at prominent venues.

In this paper, we focused our experiments on one particular setting: learning maximally accurate trees of lim-

[1] https://github.com/SiccoVerwer/binoct

[2] https://bitbucket.org/helene_verhaeghe/classificationtree/src/default/classificationtree/

[3] https://www.ibm.com/analytics/cplex-optimizer

[4] https://oscarlib.bitbucket.io

[5] https://dtai.cs.kuleuven.be/CP4IM/datasets/

[6] https://archive.ics.uci.edu/ml/index.php

Dataset	nItems	nTrans	BinOCT		DL8		CP-Based		DL8.5							
									d.o.		asc		desc		n.p.s.	
			obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)
anneal	186	812	115	*TO*	∞	*TO*	**91***	450.69	**91***	129.24	**91***	127.45	**91***	**121.87**	**91***	250.64
audiology	296	216	2	*TO*	∞	*TO*	**1**	*TO*	**1***	**180.84**	**1***	204.19	**1***	195.73	**1**	*TO*
australian-credit	250	653	82	*TO*	∞	*TO*	66	*TO*	**56***	**566.71**	**56***	586.39	**56***	593.38	57	*TO*
breast-wisconsin	240	683	12	*TO*	∞	*TO*	8	*TO*	**7***	**305.3**	**7***	325.7	**7***	330.71	**7**	*TO*
diabetes	224	76	170	*TO*	∞	*TO*	140	*TO*	**137***	**553.49**	**137***	562.83	**137***	565.5	**137**	*TO*
german-credit	224	1000	223	*TO*	∞	*TO*	**204**	*TO*	**204***	**558.73**	206	*TO*	**204***	599.87	**204**	*TO*
heart-cleveland	190	296	39	*TO*	∞	*TO*	**25**	*TO*	**25***	**124.1**	**25***	130.3	**25***	132.23	**25***	214.76
hepatitis	136	137	7	*TO*	**3***	66.62	**3***	109.36	**3***	**13.46**	**3***	14.06	**3***	14.88	**3***	27.28
hypothyroid	176	3247	55	*TO*	∞	*TO*	**53**	*TO*	**53***	392.22	**53***	**368.95**	**53***	427.34	**53**	*TO*
ionosphere	890	351	27	*TO*	∞	*TO*	20	*TO*	17	*TO*	**11**	*TO*	13	*TO*	17	*TO*
kr-vs-kp	146	3196	193	*TO*	∞	*TO*	**144***	483.15	**144***	216.11	**144***	**206.18**	**144***	223.85	**144***	528.72
letter	448	20000	813	*TO*	∞	*TO*	574	*TO*	**550**	*TO*	586	*TO*	802	*TO*	**550**	*TO*
lymph	136	148	6	*TO*	**3***	56.29	**3***	112.48	**3***	8.7	**3***	11.03	**3***	**8.47**	**3***	25.04
mushroom	238	8124	278	*TO*	∞	*TO*	**0***	352.18	**0***	331.39	4	*TO*	**0***	**0.11**	4	*TO*
pendigits	432	7494	780	*TO*	∞	*TO*	38	*TO*	32	*TO*	26	*TO*	**14**	*TO*	32	*TO*
primary-tumor	62	336	37	*TO*	**34***	2.79	**34***	8.96	**34***	1.48	**34***	1.51	**34***	**1.38**	**34***	2.43
segment	470	2310	13	*TO*	∞	*TO*	**0***	128.25	**0***	**3.54**	**0***	6.99	**0***	7.05	**0***	**3.54**
soybean	100	630	15	*TO*	**14***	41.59	**14***	40.13	**14***	**5.7**	**14***	6.34	**14***	5.75	**14***	18.41
splice-1	574	319	574	*TO*	∞	*TO*	∞	*TO*	224	*TO*	224	*TO*	**141**	*TO*	224	*TO*
tic-tac-toe	54	958	180	*TO*	**137***	3.76	**137***	9.17	**137***	**1.43**	**137***	1.54	**137***	1.55	**137***	2.12
vehicle	504	846	61	*TO*	∞	*TO*	22	*TO*	16	*TO*	18	*TO*	**13**	*TO*	16	*TO*
vote	96	435	6	*TO*	**5***	29.84	**5***	44.47	**5***	5.48	**5***	**5.33**	**5***	5.58	**5***	12.82
yeast	178	1484	395	*TO*	∞	*TO*	**366**	*TO*	**366***	**318.87**	**366***	326.2	**366***	334.15	**366***	470.88
zoo-1	72	101	**0***	0.52	**0***	1.11	**0***	0.2	**0***	**0.01**	**0***	**0.01**	**0***	**0.01**	**0***	**0.01**

Table 2: Comparison table for binary datasets with max depth = 4

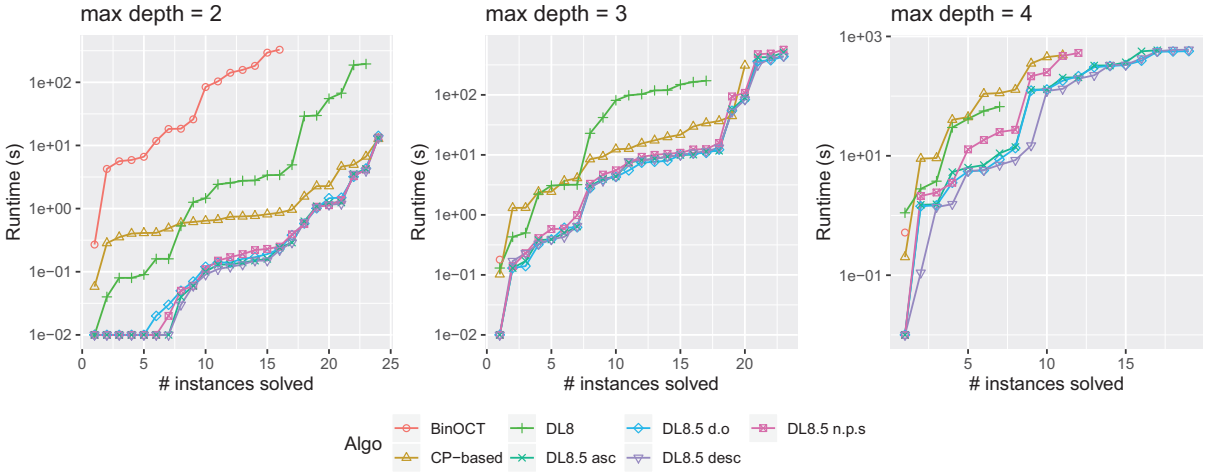


Figure 4: Cumulative number of instances solved over time

Dataset	nTrans	nFeat	nItems	BinOCT		DL8		CP-Based		DL8.5							
										d.o.		asc		desc			
				obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)	obj	time (s)		
balance-scale	625	4	32	**149***	1.2	**149***	0.5	**149***	**0.01**	**149***	**0.01**	**149***	**0.01**	**149***	**0.01**		
banknote	1372	4	3710	101	*TO*	**100***	363.01	∞	*TO*	**100***	**52.81**	**100***	63.41	**100***	58.07		
bank	4521	51	3380	449	*TO*	448	*TO*	∞	*TO*	**446***	253.87	**446***	223.42	**446***	**222.66**		
biodeg	1055	41	8356	212	*TO*	212	*TO*	∞	*TO*	**202***	**341.57**	**202***	365.83	**202***	370.26		
car	1728	6	28	**250***	4.09	**250***	0.32	**250***	0.02	**250***	**0.01**	**250***	**0.01**	**250***	**0.01**		
IndiansDiabetes	768	8	1714	**171**	*TO*	**171***	36.75	∞	*TO*	**171***	**7.43**	**171***	8.46	**171***	8.72		
Ionosphere	351	34	4624	**29**	*TO*	**29***	423.52	∞	*TO*	**29***	**25.68**	**29***	33.76	**29***	33.04		
iris	150	4	52	**0***	0.02	**0***	0.05	**0***	**0.01**	**0***	**0.01**	**0***	**0.01**	**0***	**0.01**		
letter	20000	16	352	625	*TO*	**591***	**5.97**	**591***	392.09	**591***	8.61	**591***	8.26	**591***	8.83		
messidor	1151	19	9460	**383**	*TO*	**383**	*TO*	∞	*TO*	**383***	**533.27**	**383***	563.83	**383***	534.32		
monk1	124	6	22	**22***	0.33	**22***	0.28	**22***	**0.01**	**22***	**0.01**	**22***	**0.01**	**22***	**0.01**		
monk2	169	6	22	**57***	0.79	**57***	0.28	**57***	**0.01**	**57***	**0.01**	**57***	**0.01**	**57***	**0.01**		
monk3	122	6	22	**8***	0.31	**8***	0.28	**8***	**0.01**	**8***	**0.01**	**8***	**0.01**	**8***	**0.01**		
seismic	2584	18	2240	166	*TO*	**164***	117.34	∞	*TO*	**164***	**44.3**	**164***	47.36	**164***	47.17		
spambase	4601	57	16012	660	*TO*	900	*TO*	∞	*TO*	741	*TO*	845	*TO*	**586**	*TO*		
Statlog	4435	36	3274	460	*TO*	**443**	*TO*	∞	*TO*	**443***	205.14	**443***	193.87	**443***	**188.9**		
tic-tac-toe	958	18	36	**282***	7.52	**282***	0.33	**282***	**0.01**	**282***	**0.01**	**282***	**0.01**	**282***	**0.01**		
wine	178	13	1198	**6***	73.1	**6***	7.0	**6***	74.72	**6***	1.17	**6***	1.45	**6***	**1.09**		

Table 3: Comparison table for continuous datasets with max depth = 2

ited depth without support constraints. This was motivated by our desire to compare our new approach with other approaches. However, we believe DL8.5 can be modified for use in other constraint-based decision tree learning problems, using ideas from DL8 (Nijssen and Fromont 2010).

Acknowledgements. This work was supported by Bpost.

References

Aghaei, S.; Azizi, M. J.; and Vayanos, P. 2019. Learning optimal and fair decision trees for non-discriminative decision-making. *arXiv preprint arXiv:1903.10598*.

Agrawal, R.; Mannila, H.; Srikant, R.; Toivonen, H.; Verkamo, A. I.; et al. 1996. Fast discovery of association rules. *Advances in knowledge discovery and data mining* 12(1):307–328.

Bertsimas, D., and Dunn, J. 2017. Optimal classification trees. *Machine Learning* 106(7):1039–1082.

Bessiere, C.; Hebrard, E.; and O'Sullivan, B. 2009. Minimising decision tree size as combinatorial optimisation. In *International Conference on Principles and Practice of Constraint Programming*, 173–187. Springer.

Breiman, L.; Friedman, J.; Olshen, R.; and Stone, C. 1984. *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks.

Laurent, H., and Rivest, R. L. 1976. Constructing optimal binary decision trees is NP-complete. *Information processing letters* 5(1):15–17.

Narodytska, N.; Ignatiev, A.; Pereira, F.; Marques-Silva, J.; and RAS, I. 2018. Learning optimal decision trees with sat. In *IJCAI*, 1362–1368.

Nijssen, S., and Fromont, E. 2007. Mining optimal decision trees from itemset lattices. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 530–539. ACM.

Nijssen, S., and Fromont, E. 2010. Optimal constraint-based decision tree induction from itemset lattices. *Data Mining and Knowledge Discovery* 21(1):9–51.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Verhaeghe, H.; Nijssen, S.; Pesant, G.; Quimper, C.-G.; and Schaus, P. 2019. Learning optimal decision trees using constraint programming. In *The 25th International Conference on Principles and Practice of Constraint Programming (CP2019)*.

Verwer, S., and Zhang, Y. 2019. Learning optimal classification trees using a binary linear program formulation. In *33rd AAAI Conference on Artificial Intelligence*.

European Journal of Operational Research 290 (2021) 405–421

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Invited Review

Machine learning for combinatorial optimization: A methodological tour d'horizon

Yoshua Bengio[c,b], Andrea Lodi[a,b,*], Antoine Prouvost[a,b]

[a] Canada Excellence Research Chair in Data Science for Decision Making, École Polytechnique de Montréal, Pavillon André-Aisenstadt 2920, Chemin de la Tour Montreal, Qc, H3T 1J4 Canada
[b] Mila, Institut Québecois d'Intelligence Artificielle, Pavillon André-Aisenstadt 2920, Chemin de la Tour Montreal, Qc, H3T 1J4 Canada
[c] Université de Montréal, Département d'Informatique et de Recherche Opérationelle, Pavillon André-Aisenstadt 2920, Chemin de la Tour Montreal, Qc, H3T 1J4 Canada

ARTICLE INFO

Article history:
Received 8 November 2018
Accepted 29 July 2020
Available online 8 August 2020

Keywords:
Combinatorial optimization
Machine learning
Branch and bound
Mixed-integer programming solvers

ABSTRACT

This paper surveys the recent attempts, both from the machine learning and operations research communities, at leveraging machine learning to solve combinatorial optimization problems. Given the hard nature of these problems, state-of-the-art algorithms rely on handcrafted heuristics for making decisions that are otherwise too expensive to compute or mathematically not well defined. Thus, machine learning looks like a natural candidate to make such decisions in a more principled and optimized way. We advocate for pushing further the integration of machine learning and combinatorial optimization and detail a methodology to do so. A main point of the paper is seeing generic optimization problems as data points and inquiring what is the relevant distribution of problems to use for learning on a given task.

1. Introduction

Operations research, also referred to as prescriptive analytics, started in the second world war as an initiative to use mathematics and computer science to assist military planners in their decisions (Fortun & Schweber, 1993). Nowadays, it is widely used in the industry, including but not limited to transportation, supply chain, energy, finance, and scheduling. In this paper, we focus on discrete optimization problems formulated as integer constrained optimization, *i.e.*, with integral or binary variables (called decision variables). While not all such problems are hard to solve (*e.g.*, shortest path problems), we concentrate on combinatorial optimization problems (NP-hard). This is bad news, in the sense that, for those problems, it is considered unlikely that an algorithm whose running time is polynomial in the size of the input exists. However, in practice, combinatorial optimization algorithms can solve instances with up to millions of decision variables and constraints.

How is it possible to solve NP-hard problems in practical time? Let us look at the example of the traveling salesman problem, a NP-hard problem defined on a graph where we are searching for a cycle of minimum length visiting once and only once every node. A particular case is that of the *Euclidian* traveling salesman problem. In this version, each node is assigned coordinates in a plane,[1] and the cost on an edge connecting two nodes is the Euclidian distance between them. While theoretically as hard as the general traveling salesman problem, good approximate solution can be found more efficiently in the Euclidian case by leveraging the *structure* of the graph (Larson and Odoni, 1981, Chapter 6.4.7). Likewise, diverse types of problems are solved by leveraging their special structure. Other algorithms, designed to be general, are found in hindsight to be empirically more efficient on particular problems types. The scientific literature covers the rich set of techniques researchers have developed to tackle different combinatorial optimization problems. An expert will know how to further refine algorithm parameters to different behaviors of the optimization process, thus extending this knowledge with unwritten intuition. These techniques, and the parameters controlling them, have been collectively *learned* by the community to perform on the inaccessible distribution of problem instances deemed valuable. The focus of this paper is on combinatorial optimization algorithms that automatically perform learning on a chosen implicit distribution of problems. Incorporating machine learning components in the algorithm can achieve this.

Conversely, machine learning focuses on performing a task given some (finite and usually noisy) data. It is well suited for natural signals for which no clear mathematical formulation emerges

* Corresponding author. Polytechnique de Montréal - Canada Excellence Research Chair, C.P. 6079, Succ. Centre-ville Montréal, Québec, Canada H3C 3A7
E-mail addresses: yoshua.bengio@mila.quebec (Y. Bengio), andrea.lodi@polymtl.ca (A. Lodi), antoine.prouvost@polymtl.ca (A. Prouvost).

[1] Or more generally in a vector space of arbitrary dimension.

https://doi.org/10.1016/j.ejor.2020.07.063
0377-2217/

because the true data distribution is not known analytically, such as when processing images, text, voice or molecules, or with recommender systems, social networks or financial predictions. Most of the times, the learning problem has a statistical formulation that is solved through mathematical optimization. Recently, dramatic progress has been achieved with deep learning, an machine learning sub-field building large parametric approximators by composing simpler functions. Deep learning excels when applied in high dimensional spaces with a large number of data points.

1.1. Motivation

From the combinatorial optimization point of view, machine learning can help improve an algorithm on a distribution of problem instances in two ways. On the one side, the researcher assumes expert knowledge[2] about the optimization algorithm, but wants to replace some heavy computations by a fast approximation. Learning can be used to build such approximations in a generic way, *i.e.*, without the need to derive new explicit algorithms. On the other side, expert knowledge may not be sufficient and some algorithmic decisions may be unsatisfactory. The goal is therefore to explore the space of these decisions, and learn out of this experience the best performing behavior (policy), hopefully improving on the state of the art. Even though machine learning is approximate, we will demonstrate through the examples surveyed in this paper that this does not systematically mean that incorporating learning will compromise overall theoretical guarantees. From the point of view of using machine learning to tackle a combinatorial problem, combinatorial optimization can decompose the problem into smaller, hopefully simpler, learning tasks. The combinatorial optimization structure therefore acts as a relevant prior for the model. It is also an opportunity to leverage the combinatorial optimization literature, notably in terms of theoretical guarantees (*e.g.*, feasibility and optimality).

1.2. Setting

Imagine a delivery company in Montreal that needs to solve traveling salesman problem. Every day, the customers may vary, but usually, many are downtown and few on top of the Mont Royal mountain. Furthermore, Montreal streets are laid on a grid, making the distances close to the ℓ_1 distance. How close? Not as much as Phoenix, but certainly more than Paris. The company does not care about solving all possible traveling salesman problem, but only *theirs*. Explicitly defining what makes a traveling salesman problem a likely one for the company is tedious, does not scale, and it is not clear how it can be leveraged when explicitly writing an optimization algorithm. We would like to automatically specialize traveling salesman problem algorithms for this company.

The true probability distribution of likely traveling salesman problem in the Montreal scenario is defining the instances on which we would like our algorithm to perform well. This is unknown, and cannot even be mathematically characterized in an explicit way. Because we do not know what is in this distribution, we can only learn an algorithm that performs well on a finite set of traveling salesman problem sampled from this distribution (for instance, a set of historical instances collected by the company), thus implicitly incorporating the desired information about the distribution of instances. As a comparison, in traditional machine learning tasks, the true distribution could be that of all possible images of cats, while the training distribution is a finite set of such images. The challenge in learning is that an algorithm that performs well on problem instances used for learning may not work properly on other instances from the true probability distribution. For the company, this would mean the algorithm only does well on past problems, but not on the future ones. To control this, we monitor the performance of the learned algorithm over another independent set of *unseen* problem instances. Keeping the performances similar between the instances used for learning and the unseen ones is known in machine learning as *generalizing*. Current machine learning algorithms can generalize to examples from the same distribution, but tend to have more difficulty generalizing out-of-distribution (although this is a topic of intense research in ML), and so we may expect combinatorial optimization algorithms that leverage machine learning models to fail when evaluated on unseen problem instances that are too far from what has been used for training the machine learning predictor. As previously motivated, it is also worth noting that traditional combinatorial optimization algorithms might not even work consistently across all possible instances of a problem family, but rather tend to be more adapted to particular structures of problems, *e.g.*, Euclidean traveling salesman problem.

Finally, the implicit knowledge extracted by machine learning algorithms is complementary to the hard-won explicit expertise extracted through combinatorial optimization research. Rather, it aims to augment and automate the unwritten expert intuition (or lack of) on various existing algorithms. Given that these problems are highly structured, we believe it is relevant to augment solving algorithms with machine learning – and especially deep learning to address the high dimensionality of such problems.

In the following, we survey the attempts in the literature to achieve such automation and augmentation, and we present a methodological overview of those approaches. In light of the current state of the field, the literature we survey is exploratory, *i.e.*, we aim at highlighting promising research directions in the use of machine learning within combinatorial optimization, instead of reporting on already mature algorithms.

1.3. Outline

We have introduced the context and motivations for building combinatorial optimization algorithms together with machine learning. The remainder of this paper is organized as follows. Section 2 provides minimal prerequisites in combinatorial optimization, machine learning, deep learning, and reinforcement learning necessary to fully grasp the content of the paper. Section 3 surveys the recent literature and derives two distinctive, orthogonal, views: Section 3.1 shows how machine learning policies can either be learned by imitating an expert or discovered through experience, while Section 3.2 discusses the interplay between the machine learning and combinatorial optimization components. Section 5 pushes further the reflection on the use of machine learning for combinatorial optimization and brings to the fore some methodological points. In Section 6, we detail critical practical challenges of the field. Finally, some conclusions are drawn in Section 7.

2. Preliminaries

In this section, we give a basic (sometimes rough) overview of combinatorial optimization and machine learning, with the unique aim of introducing concepts that are strictly required to understand the remainder of the paper.

2.1. Combinatorial optimization

Without loss of generality, a combinatorial optimization problem can be formulated as a constrained min-optimization program. Constraints model natural or imposed restrictions of the problem,

[2] Theoretical and/or empirical.

variables define the decisions to be made, while the objective function, generally a cost to be minimized, defines the measure of the quality of every feasible assignment of values to variables. If the objective and constraints are linear, the problem is called a linear programming problem. If, in addition, some variables are also restricted to only assume integer values, then the problem is a mixed-integer linear programming problem.

The set of points that satisfy the constraints is the feasible region. Every point in that set (often referred to as a feasible solution) yields an upper bound on the objective value of the optimal solution. Exact solving is an important aspect of the field, hence a lot of attention is also given to find lower bounds to the optimal cost. The tighter the lower bounds, with respect to the optimal solution value, the higher the chances that the current algorithmic approaches to tackle mixed-integer linear programming described in the following could be successful, *i.e.*, effective if not efficient.

Linear and mixed-integer linear programming problems are the workhorse of combinatorial optimization because they can model a wide variety of problems and are the best understood, *i.e.*, there are reliable algorithms and software tools to solve them. We give them special considerations in this paper but, of course, they do not represent the entire combinatorial optimization, mixed-integer nonlinear programming being a rapidly expanding and very significant area both in theory and in practical applications. With respect to complexity and solution methods, linear programming is a polynomial problem, well solved, in theory and in practice, through the simplex algorithm or interior points methods. Mixed-integer linear programming, on the other hand, is an NP-hard problem, which does not make it hopeless. Indeed, it is easy to see that the complexity of mixed-integer linear programming is associated with the integrality requirement on (some of) the variables, which makes the mixed-integer linear programming feasible region nonconvex. However, dropping the integrality requirement (i) defines a proper relaxation of mixed-integer linear programming (*i.e.*, an optimization problem whose feasible region contains the mixed-integer linear programming feasible region), which (ii) happens to be an linear programming, *i.e.*, polynomially solvable. This immediately suggests the algorithmic line of attack that is used to solve mixed-integer linear programming through a whole ecosystem of branch-and-bound techniques to perform implicit enumeration. Branch and bound implemements a divide-and-conquer type of algorithm representable by a search tree in which, at every node, an linear programming relaxation of the problem (possibly augmented by branching decisions, see below) is efficiently computed. If the relaxation is infeasible, or if the solution of the relaxation is naturally (mixed-)integer, *i.e.*, mixed-integer linear programming feasible, the node does not need to be expanded. Otherwise, there exists at least one variable, among those supposed to be integer, taking a fractional value in the linear programming solution and that variable can be chosen for branching (enumeration), *i.e.*, by restricting its value in such a way that two child nodes are created. The two child nodes have disjoint feasible regions, none of which contains the solution of the previous linear programming relaxation. We use Fig. 1 to illustrate the branch-and-bound algorithm for a minimization mixed-integer linear programming. At the root node in the figure, the variable x_2 has a fractional value in the linear programming solution (not represented), thus branching is done on the floor (here zero) and ceiling (here one) of this value. When an integer solution is found, we also get an upper bound (denoted as $\bar{z}$) on the optimal solution value of the problem. At every node, we can then compare the solution value of the relaxation (denoted as $\underline{z}$) with the minimum upper bound found so far, called the incumbent solution value. If the latter is smaller than the former for a specific node, no better (mixed-)integer solution can be found in the sub-tree originated by the node itself, and it can be pruned.

All commercial and noncommercial mixed-integer linear programming solvers enhance the above enumeration framework with the extensive use of cutting planes, *i.e.*, valid linear inequalities that are added to the original formulation (especially at the root of the branch-and-bound tree) in the attempt of strengthening its linear programming relaxation. The resulting framework, referred to as the branch-and-cut algorithm, is then further enhanced by additional algorithmic components, preprocessing and primal heuristics being the most crucial ones. The reader is referred to Wolsey (1998) and Conforti, Conrnuéjols, and Zambelli (2014) for extensive textbooks on mixed-integer linear programming and to Lodi (2009) for a detailed description of the algorithmic components of the mixed-integer linear programming solvers.

We end the section by noting that there is a vast literature devoted to (primal) heuristics, *i.e.*, algorithms designed to compute "good in practice" solutions to CO problems without optimality guarantee. Although a general discussion on them is outside the scope here, those heuristic methods play a central role in combinatorial optimization and will be considered in specific contexts in the present paper. The interested reader is referred to Fischetti and Lodi (2011) and Gendreau and Potvin (2010).

2.2. Machine learning

Supervised learning In supervised learning, a set of input (features) / target pairs is provided and the task is to find a function that for every input has a predicted output as close as possible to the provided target. Finding such a function is called learning and is solved through an optimization problem over a family of functions. The loss function, *i.e.*, the measure of discrepancy between the output and the target, can be chosen depending on the task (regression, classification, *etc.*) and on the optimization methods. However, this approach is not enough because the problem has a statistical nature. It is usually easy enough to achieve a good score on the given examples but one wants to achieve a good score on unseen examples (test data). This is known as generalization.

Mathematically speaking, let X and Y, following a joint probability distribution P, be random variables representing the input features and the target. Let ℓ be the per sample loss function to minimize, and let $\{f_\theta \,|\, \theta \in \mathbb{R}^p\}$ be the family of machine learning models (parametric in this case) to optimize over. The supervised learning problem is framed as

$$\min_{\theta \in \mathbb{R}^p} \; \mathbb{E}_{X,Y \sim P}\, \ell(Y, f_\theta(X)). \tag{1}$$

For instance, f_θ could be a linear model with weights θ that we wish to *learn*. The loss function ℓ is task dependent (*e.g.*, classification error) and can sometimes be replaced by a surrogate one (*e.g.*, a differentiable one). The probability distribution is unknown and inaccessible. For example, it can be the probability distribution of all natural images. Therefore, it is approximated by the empirical probability distribution over a finite dataset $D_{train} = \{(x_i, y_i)\}_i$ and the optimization problem solved is

$$\min_{\theta \in \mathbb{R}^p} \sum_{(x,y) \in D_{train}} \frac{1}{|D_{train}|} \ell(y, f_\theta(x)). \tag{2}$$

A model is said to generalize, if low objective values of (2) translate in low objective values of (1). Because (1) remains inaccessible, we estimate the generalization error by evaluating the trained model on a separate test dataset D_{test} with

$$\sum_{(x,y) \in D_{test}} \frac{1}{|D_{test}|} \ell(y, f_\theta(x)). \tag{3}$$

If a model (*i.e.*, a family of functions) can represent many different functions, the model is said to have high capacity and is prone

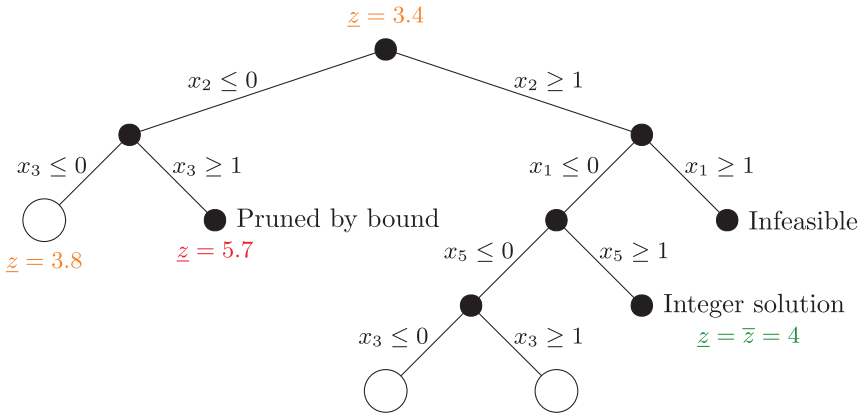


Fig. 1. A branch-and-bound tree for mixed-integer linear programming. The linear programming relaxation is computed at every node (only partially shown in the figure). Nodes still open for exploration are represented as blank.

to overfitting: doing well on the training data but not generalizing to the test data. Regularization is anything that can improve the test score at the expense of the training score and is used to restrict the practical capacity of a model. On the contrary, if the capacity is too low, the model underfits and performs poorly on both sets. The boundary between overfitting and underfitting can be estimated by changing the effective capacity (the richness of the family of functions reachable by training): below the critical capacity one underfits and test error decreases with increases in capacity, while above that critical capacity one overfits and test error increases with increases in capacity.

Selecting the best among various trained models cannot be done on the test set. Selection is a form of optimization, and doing so on the test set would bias the estimator in (2). This is a common form of data dredging, and a mistake to be avoided. To perform model selection, a validation dataset D_{valid} is used to estimate the generalization error of different machine learning models is necessary. Model selection can be done based on these estimates, and the final unbiased generalization error of the selected model can be computed on the test set. The validation set is therefore often used to select effective capacity, *e.g.*, by changing the amount of training, the number of parameters θ, and the amount of regularization imposed to the model.

Unsupervised learning

In unsupervised learning, one does not have targets for the task one wants to solve, but rather tries to capture some characteristics of the joint distribution of the observed random variables. The variety of tasks include density estimation, dimensionality reduction, and clustering. Because unsupervised learning has received so far little attention in conjunction with combinatorial optimization and its immediate use seems difficult, we are not discussing it any further. The reader is referred to Bishop (2006); Goodfellow, Bengio, and Courville (2016); Murphy (2012) for textbooks on machine learning.

Reinforcement learning

In reinforcement learning, an agent interacts with an environment through a markov decision process, as illustrated in Fig. 2. At every time step, the agent is in a given state of the environment and chooses an action according to its (possibly stochastic) policy. As a result, it receives from the environment a reward and enters a new state. The goal in reinforcement learning is to train the agent to maximize the expected sum of future rewards, called the return. For a given policy, the expected return given a current state (resp. state and action pair) is known as the value function (resp. state action value function). Value functions follow the Bellman equation, hence the problem can be formulated as dynamic programming, and solved approximately. The dynamics of the environment

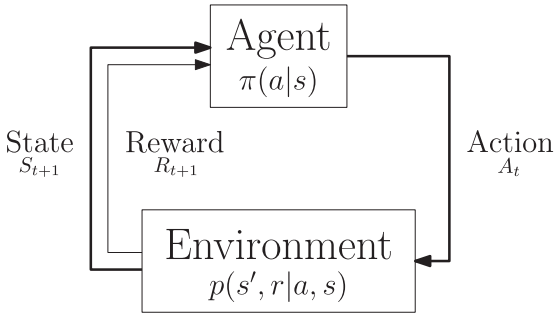


Fig. 2. The Markov decision process associated with reinforcement learning, modified from Sutton and Barto (2018). The agent behavior is defined by its policy π, while the environment evolution is defined by the dynamics p. Note that the reward is not necessary to define the evolution and is provided only as a learning mechanism for the agent. Actions, states, and rewards are random variables in the general framework.

need not be known by the agent and are learned directly or indirectly, yielding an exploration *vs* exploitation dilemma: choosing between exploring new states for refining the knowledge of the environment for possible long-term improvements, or exploiting the best-known scenario learned so far (which tends to be in already visited or predictable states).

The state should fully characterize the environment at every step, in the sense that future states only depend on past states via the current state (the Markov property). When this is not the case, similar methods can be applied but we say that the agent receives an *observation* of the state. The Markov property no longer holds and the markov decision process is said to be partially observable.

Defining a reward function is not always easy. Sometimes one would like to define a very sparse reward, such as 1 when the agent solves the problem, and 0 otherwise. Because of its underlying dynamic programming process, reinforcement learning is naturally able to credit states/actions that lead to future rewards. Nonetheless, the aforementioned setting is challenging as it provides no learning opportunity until the agent (randomly, or through advanced approaches) solves the problem. Furthermore, when the policy is approximated (for instance, by a linear function), the learning is not guaranteed to converge and may fall into local minima. For example, an autonomous car may decide not to drive anywhere for fear of hitting a pedestrian and receiving a dramatic negative reward. These challenges are strongly related to the aforementioned exploration dilemma. The reader is referred to Sutton and Barto (2018) for an extensive textbook on reinforcement learning.

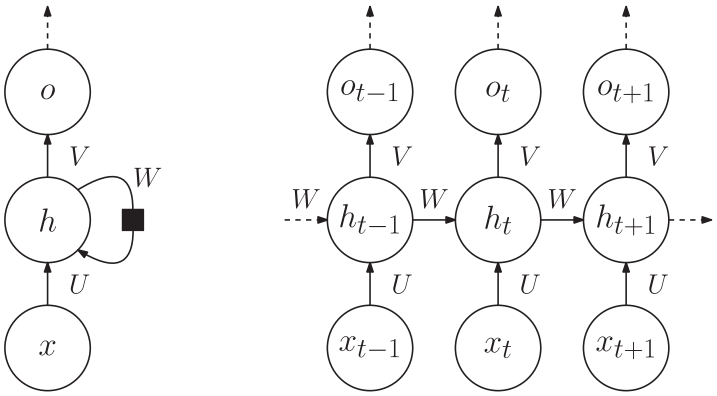


Fig. 3. A vanilla recurrent neural network modified from Goodfellow et al. (2016). On the left, the black square indicates a one step delay. On the right, the same recurrent neural network is shown unfolded. Three sets U, V, and W of parameters are represented and re-used at every time step.

Deep learning

Deep learning is a successful method for building parametric composable functions in high dimensional spaces. In the case of the simplest neural network architecture, the feedforward neural network (also called an multilayer perceptron), the input data is successively passed through a number of layers. For every layer, an affine transformation is applied on the input vector, followed by a non-linear scalar function (named activation function) applied element-wise. The output of a layer, called intermediate activations, is passed on to the next layer. All affine transformations are independent and represented in practice as different matrices of coefficients. They are learned, *i.e.*, optimized over, through stochastic gradient descent, the optimization algorithm used to minimize the selected loss function. The stochasticity comes from the limited number of data points used to compute the loss before applying a gradient update. In practice, gradients are computed using reverse mode automatic differentiation, a practical algorithm based on the chain rule, also known as back-propagation. Deep neural networks can be difficult to optimize, and a large variety of techniques have been developed to make the optimization behave better, often by changing architectural designs of the network. Because neural networks have dramatic capacities, *i.e.*, they can essentially match any dataset, thus being prone to overfitting, they are also heavily regularized. Training them by stochastic gradient descent also regularizes them because of the noise in the gradient, making neural networks generally robust to overfitting issues, even when they are very large and would overfit if trained with more aggressive optimization. In addition, many hyper-parameters exist and different combinations are evaluated (known as hyper-parameters optimization). Deep learning also sets itself apart from more traditional machine learning techniques by taking as inputs all available raw features of the data, *e.g.*, all pixels of an image, while traditional machine learning typically requires to engineer a limited number of domain-specific features.

Deep learning researchers have developed different techniques to tackle this variety of structured data in a manner that can handle variable-size data structures, *e.g.*, variable-length sequences. In this paragraph, and in the next, we present such state-of-the-art techniques. These are complex topics, but lack of comprehension does not hinder the reading of the paper. At a high level, it is enough to comprehend that these are architectures designed to handle different structures of data. Their usage, and in particular the way they are learned, remains very similar to plain feedforward neural networks introduced above. The first architectures presented are the recurrent neural network. These models can operate on sequence data by *sharing parameters* across different

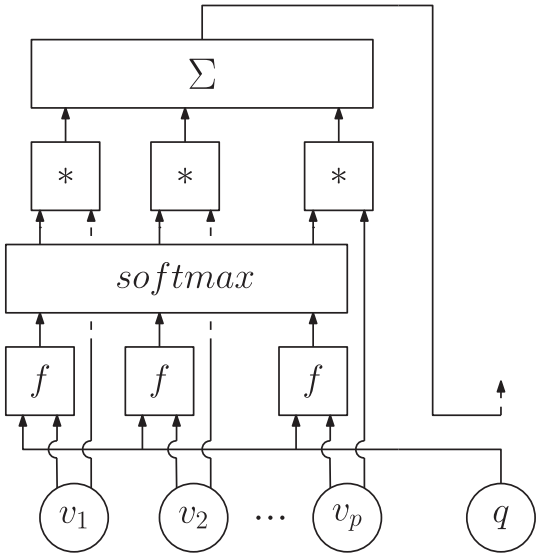


Fig. 4. A vanilla attention mechanism where a query q is computed against a set of values $(v_i)_i$. An affinity function f, such as a dot product, is used on query and value pairs. If it includes some parameters, the mechanism can be learned.

sequence steps. More precisely, a same neural network block is successively applied at every step of the sequence, *i.e.*, with the same architecture and parameter values at each time step. The specificity of such a network is the presence of recurrent layers: layers that take as input both the activation vector of the previous layer and its own activation vector on the preceding sequence step (called a hidden state vector), as illustrated in Fig. 3.

Another important size-invariant technique are *attention mechanisms*. They can be used to process data where each data point corresponds to a set. In that context, parameter sharing is used to address the fact that different sets need not to be of the same size. Attention is used to query information about all elements in the set, and merge it for downstream processing in the neural network, as depicted in Fig. 4. An affinity function takes as input the query (which represents any kind of contextual information which informs where attention should be concentrated) and a representation of an element of the set (both are activation vectors) and outputs a scalar. This is repeated over all elements in the set for the same query. Those scalars are normalized (for instance with a softmax function) and used to define a weighted sum of the representations of elements in the set that can, in turn, be used in

the neural network making the query. This form of content-based soft attention was introduced by Bahdanau, Cho, and Bengio (2015). A general explanation of attention mechanisms is given by Vaswani et al. (2017). Attention can be used to build graph neural network, *i.e.*, neural networks able to process graph structured input data, as done by Veličković et al. (2018). In this architecture, every node attends over the set of its neighbors. The process is repeated multiple times to gather information about nodes further away. graph neural network can also be understood as a form of message passing (Gilmer, Schoenholz, Riley, Vinyals, & Dahl, 2017).

Deep learning and back-propagation can be used in supervised, unsupervised, or reinforcement learning. The reader is referred to Goodfellow et al. (2016) for a machine learning textbook devoted to deep learning.

3. Recent approaches

We survey different uses of machine learning to help solve combinatorial optimization problems and organize them along two orthogonal axes. First, in Section 3.1 we illustrate the two main motivations for using learning: approximation and discovery of new policies. Then, in Section 3.2, we show examples of different ways to combine learned and traditional algorithmic elements.

3.1. Learning methods

This section relates to the two motivations reported in Section 1.1 for using machine learning in combinatorial optimization. In some works, the researcher assumes theoretical and/or empirical knowledge about the decisions to be made for a combinatorial optimization algorithm, but wants to alleviate the computational burden by approximating some of those decisions with machine learning. On the contrary, we are also motivated by the fact that, sometimes, expert knowledge is not satisfactory and the researcher wishes to find better ways of making decisions. Thus, machine learning can come into play to train a model through trial and error reinforcement learning.

We frame both these motivations in the state/action markov decision process framework introduced in Section 2.2, where the environment is the internal state of the algorithm. We care about learning algorithmic decisions utilized by a combinatorial optimization algorithm and we call the function making the decision a *policy*, that, given all available information,[3] returns (possibly stochastically) the action to be taken. The policy is the function that we want to learn using machine learning and we show in the following how the two motivations naturally yield two learning settings. Note that the case where the length of the trajectory of the markov decision process has value 1 is a common edge case (called the bandit setting) where this formulation can seem excessive, but it nonetheless helps comparing methods.

In the case of using machine learning to approximate decisions, the policy is often learned by *imitation learning*, thanks to *demonstrations*, because the expected behavior is shown (demonstrated) to the machine learning model by an expert (also called oracle, even though it is not necessarily optimal), as shown in Fig. 5. In this setting, the learner is not trained to optimize a performance measure, but to *blindly* mimic the expert.

In the case where one cares about discovering new policies, *i.e.*, optimizing an algorithmic decision function from the ground up, the policy may be learned by reinforcement learning through *experience*, as shown in Fig. 6. Even though we present the learning problem under the fundamental markov decision process of reinforcement learning, this does not constrain one to use the major reinforcement learning algorithms (approximate dynamic programming and policy gradients) to maximize the expected sum of rewards. Alternative optimization methods, such as bandit algorithms, genetic algorithms, direct/local search, can also be used to solve the reinforcement learning problem.[4]

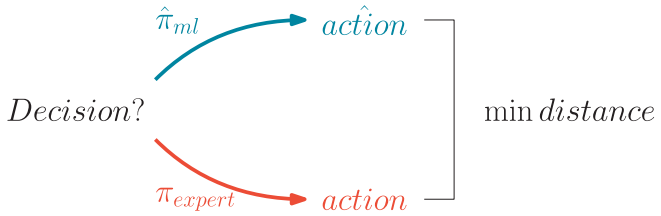


Fig. 5. In the demonstration setting, the policy is trained to reproduce the action of an expert policy by minimizing some discrepancy in the action space.

It is critical to understand that in the imitation setting, the policy is learned through supervised targets provided by an expert for every action (and without a reward), whereas in the experience setting, the policy is learned from a reward (possibly delayed) signal using reinforcement learning (and without an expert). In imitation, the agent is taught *what* to do, whereas in reinforcement learning, the agent is encouraged to quickly *accumulate* rewards. The distinction between these two settings is far more complex than the distinction made here. We explore some of this complexity, including their strengths and weaknesses, in Section 5.1.

In the following, few papers demonstrating the different settings are surveyed.

3.1.1. Demonstration

In Baltean-Lugojan, Misener, Bonami, and Tramontani (2018), the authors use a neural network to approximate the lower bound improvement generated by tightening the current relaxation via cutting planes (cuts, for short). More precisely, Baltean-Lugojan et al. (2018) consider non-convex quadratic programming problems and aim at approximating the associated semidefinite programming relaxation, known to be strong but time-consuming, by a linear program. A straightforward way of doing that is to iteratively add (linear) cutting planes associated with negative eigenvalues, especially considering small-size (square) submatrices of the original quadratic objective function. That approach has the advantage of generating sparse cuts[5] but it is computationally challenging because of the exponential number of those submatrices and because of the difficulty of finding the right metrics to select among the violated cuts. The authors propose to solve semidefinite programming to compute the bound improvement associated with considering specific submatrices, which is also a proxy on the quality of the cuts that could be separated from the same submatrices. In this context, supervised (imitation) learning is applied offline to approximate the objective value of the semidefinite programming problem associated with a submatrix selection and, afterward, the model can be rapidly applied to select the most promising submatrices without the very significant computational burden of solving semidefinite programming. Of course, the rational is that the most promising submatrices correspond to the most promising cutting planes and Baltean-Lugojan et al. (2018) train a model to estimate the objective of an semidefinite programming problem only in or-

[3] A *state* if the information is sufficient to fully characterize the environment at that time in a Markov decision process setting.

[4] In general, identifying which algorithm will perform best is an open research question unlikely to have a simple answer, and is outside of the scope of the methodology presented here.

[5] The reader is referred to Dey and Molinaro (2018) for a detailed discussion on the importance of sparse cutting planes in mixed-integer linear programming.

Fig. 6. When learning through a reward signal, no expert is involved; only maximizing the expected sum of future rewards (the return) matters.

der to decide to add the most promising cutting planes. Hence, cutting plane selection is the ultimate policy learned.

Another example of demonstration is found in the context of branching policies in branch-and-bound trees of mixed-integer linear programming. The choice of variables to branch on can dramatically change the size of the branch-and-bound tree, hence the solving time. Among many heuristics, a well-performing approach is *strong branching* (Applegate, Bixby, Chvátal, & Cook, 2007). Namely, for every branching decision to be made, strong branching performs a one step look-ahead by tentatively branching on many candidate variables, computes the linear programming relaxations to get the potential lower bound improvements, and eventually branches on the variable providing the best improvement. Even if not all variables are explored, and the linear programming value can be approximated, this is still a computationally expensive strategy. For these reasons, Marcos Alvarez, Louveaux, and Wehenkel (2014, 2017) use a special type of decision tree (a classical model in supervised learning) to approximate strong branching decisions using supervised learning. Khalil, Bodic, Song, Nemhauser, and Dilkina (2016) propose a similar approach, where a linear model is learned on the fly for every instance by using strong branching at the top of the tree, and eventually replacing it by its machine learning approximation. The linear approximator of strong branching introduced in Marcos Alvarez, Wehenkel, and Louveaux (2016) is learned in an active fashion: when the estimator is deemed unreliable, the algorithm falls back to true strong branching and the results are then used for both branching and learning. In all the branching algorithms presented here, inputs to the machine learning model are engineered as a vector of fixed length with static features descriptive of the instance, and dynamic features providing information about the state of the branch-and-bound process. Gasse, Chételat, Ferroni, Charlin, and Lodi (2019) use a neural network to learn an offline approximation to strong branching, but, contrary to the aforementioned papers, the authors use a raw exhaustive representation (*i.e.*, they do not discard nor aggregate any information) of the sub-problem associated with the current branching node as input to the machine learning model. Namely, an mixed-integer linear programming sub-problem is represented as a bipartite graph on variables and constraints, with edges representing non-zero coefficients in the constraint matrix. Each node is augmented with a set of features to fully describe the sub-problem, and a graph neural network is used to build an machine learning approximator able to process this type of structured data. Node selection, *i.e.*, deciding on the next branching node to explore in a branch-and-bound tree, is also a critical decision in mixed-integer linear programming. He, Daume III, and Eisner (2014) learn a policy to select among the open branching nodes the one that contains the optimal solution in its sub-tree. The training algorithm is an online learning method collecting expert behaviors throughout the entire learning phase. The reader is referred to Lodi and Zarpellon (2017) for an extended survey on learning and branching in mixed-integer linear programming.

Branch and bound is a technique not limited to mixed-integer linear programming and can be use for general tree search. Hottung, Tanaka, and Tierney (2017) build a tree search procedure for the container pre-marshalling problem in which they aim to learn, not only a branching policy (similar in principle to what has been discussed in the previous paragraph), but also a value network to estimate the value of partial solutions and used for bounding. The authors leverage a form of convolutional neural network[6] for both networks and train them in a supervised fashion using pre-computed solutions of the problem. The resulting algorithm is heuristic due the approximations made while bounding.

As already mentioned at the beginning of Section 3.1, learning a policy by demonstration is identical to supervised learning, where training pairs of input state and target actions are provided by the expert. In the simplest case, expert decisions are collected beforehand, but more advanced methods can collect them online to increase stability as previously shown in Marcos Alvarez et al. (2016) and He et al. (2014).

3.1.2. Experience

Considering the traveling salesman problem on a graph, it is easy to devise a greedy heuristic that builds a tour by sequentially picking the nodes among those that have not been visited yet, hence defining a permutation. If the criterion for selecting the next node is to take the closest one, then the heuristic is known as the nearest neighbor. This simple heuristic has poor practical performance and many other heuristics perform better empirically, *i.e.*, build cheaper tours. Selecting the nearest node is a fair intuition but turns out to be far from satisfactory. Khalil, Dai, Zhang, Dilkina, and Song (2017a) suggest learning the criterion for this selection. They build a greedy heuristic framework, where the node selection policy is learned using a graph neural network (Dai, Dai, & Song, 2016), a type of neural network able to process input graphs of any finite size by a mechanism of message passing (Gilmer et al., 2017). For every node to select, the authors feed to the network the graph representation of the problem – augmented with features indicating which of the nodes have already been visited – and receive back an action value for every node. Action values are used to train the network through reinforcement learning (Q-learning in particular) and the partial tour length is used as a reward.

This example does not do justice to the rich traveling salesman problem literature that has developed far more advanced algorithms performing orders of magnitude better than machine learning ones. Nevertheless, the point we are trying to highlight here is that given a fixed context, and a decision to be made, machine learning can be used to discover new, potentially better performing policies. Even on state-of-the-art traveling salesman problem algorithms (*i.e.*, when exact solving is taken to its limits), many decisions are made in heuristic ways, *e.g.*, cutting plane selection, thus leaving room for machine learning to assist in making these decisions.

Once again, we stress that learning a policy by experience is well described by the markov decision process framework of reinforcement learning, where an agent maximizes the return (defined in Section 2.2). By matching the reward signal with the optimization objective, the goal of the learning agent becomes to solve the problem, without assuming any expert knowledge. Some methods that were not presented as reinforcement learning can also be

[6] A type of neural network, usually used on image input, that leverages parameter sharing to extract local information.

cast in this markov decision process formulation, even if the optimization methods are not those of the reinforcement learning community. For instance, part of the combinatorial optimization literature is dedicated to automatically build specialized heuristics for different problems. The heuristics are build by orchestrating a set of moves, or subroutines, from a pre-defined domain-specific collections. For instance, to tackle bipartite boolean quadratic programming problems, Karapetyan, Punnen, and Parkes (2017) represent this orchestration as a Markov chain where the states are the subroutines. One Markov chain is parametrized by its transition probabilities. Mascia, López-Ibáñez, Dubois-Lacoste, and Stützle (2014), on the other hand, define valid succession of moves through a grammar, where words are moves and sentences correspond to heuristics. The authors introduce a parametric space to represent sentences of a grammar. In both cases, the setting is very close to the markov decision process of reinforcement learning, but the parameters are learned though direct optimization of the performances of their associated heuristic through so-called *automatic configuration tools* (usually based on genetic or local search, and exploiting parallel computing). Note that the learning setting is rather simple as the parameters do not adapt to the problem instance, but are fixed for various clusters. From the machine learning point of view, this is equivalent to a piece-wise constant regression. If more complex models were to be used, direct optimization may not scale adequately to obtain good performances. The same approach to building heuristics can be brought one level up if, instead of orchestrating sets of moves, it arranges predefined heuristics. The resulting heuristic is then called a *hyper-heuristic*. Özcan, Misir, Ochoa, and Burke (2012) build a hyper-heuristic for examination timetabling by learning to combine existing heuristics. They use a bandit algorithm, a stateless form of reinforcement learning (see Sutton and Barto, 2018, Chapter 2), to learn online a value function for each heuristic.

We close this section by noting that demonstration and experience are not mutually exclusive and most learning tasks can be tackled in both ways. In the case of selecting the branching variables in an mixed-integer linear programming branch-and-bound tree, one could adopt anyone of the two prior strategies. On the one hand, Khalil et al. (2016); Marcos Alvarez et al. (2014, 2017); Marcos Alvarez et al. (2016) estimate that strong branching is an effective branching strategy but computationally too expensive and build a machine learning model to approximate it. On the other hand, one could believe that no branching strategy is good enough and try to learn one from the ground up, for instance through reinforcement learning as suggested (but not implemented) in Khalil et al. (2016). An intermediary approach is proposed by Liberto, Kadioglu, Leo, and Malitsky (2016). The authors recognize that, among traditional variable selection policies, the ones performing well at the top of the branch-and-bound tree are not necessarily the same as the ones performing well deeper down. Hence, the authors learn a model to dynamically switch among predefined policies during branch-and-bound based on the current state of the tree. While this seems like a case of imitation learning, given that traditional branching policies can be thought of as experts, this is actually not the case. In fact, the model is not learning *from* any expert, but really learning to choose between pre-existing policies. This is technically not a branching variable selection, but rather a branching heuristic selection policy. Each sub-tree is represented by a vector of handcrafted features, and a clustering of these vectors is performed. Similarly to what was detailed in the previous paragraph about the work of Karapetyan et al. (2017); Mascia et al. (2014), automatic configuration tools are then used to assign the best branching policy to each cluster. When branching at a given node, the cluster the closest to the current sub-tree is retrieved, and its assigned policy is used.

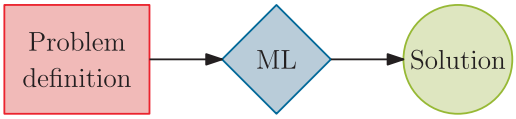


Fig. 7. Machine learning acts alone to provide a solution to the problem.

3.2. Algorithmic structure

In this section, we survey how the learned policies (whether from demonstration or experience) are combined with traditional combinatorial optimization algorithms, *i.e.*, considering machine learning and explicit algorithms as building blocks, we survey how they can be laid out in different templates. The three following sections are not necessarily disjoint nor exhaustive but are a natural way to look at the literature.

3.2.1. End to end learning

A first idea to leverage machine learning to solve discrete optimization problems is to train the machine learning model to output solutions directly from the input instance, as shown in Fig. 7.

This approach has been explored recently, especially on Euclidean traveling salesman problem. To tackle the problem with deep learning, Vinyals, Fortunato, and Jaitly (2015) introduce the pointer network wherein an encoder, namely an recurrent neural network, is used to parse all the traveling salesman problem nodes in the input graph and produces an encoding (a vector of activations) for each of them. Afterward, a decoder, also an recurrent neural network, uses an attention mechanism similar to Bahdanau et al. (2015) (Section 2.2) over the previously encoded nodes in the graph to produce a probability distribution over these nodes (through the softmax layer previously illustrated in Fig. 4). Repeating this decoding step, it is possible for the network to output a permutation over its inputs (the traveling salesman problem nodes). This method makes it possible to use the network over different input graph sizes. The authors train the model through supervised learning with precomputed traveling salesman problem solutions as targets. Bello, Pham, Le, Norouzi, and Bengio (2017) use a similar model and train it with reinforcement learning using the tour length as a reward signal. They address some limitations of supervised (imitation) learning, such as the need to compute optimal (or at least high quality) traveling salesman problem solutions (the targets), that in turn, may be ill-defined when those solutions are not computed exactly, or when multiple solutions exist. Kool and Welling (2018) introduce more prior knowledge in the model using a graph neural network instead of an recurrent neural network to process the input. Emami and Ranka (2018) and Nowak, Villar, Bandeira, and Bruna (2017) explore a different approach by directly approximating a double stochastic matrix in the output of the neural network to characterize the permutation. The work of Khalil et al. (2017a), introduced in Section 3.1.2, can also be understood as an end to end method to tackle the traveling salesman problem, but we prefer to see it under the eye of Section 3.2.3. It is worth noting that tackling the traveling salesman problem through machine learning is not new. Earlier work from the nineties focused on Hopfield neural networks and self organizing neural networks, the interested reader is referred to the survey of Smith (1999).

In another example, Larsen et al. (2018) train a neural network to predict the solution of a stochastic load planning problem for which a deterministic mixed-integer linear programming formulation exists. Their main motivation is that the application needs to make decisions at a tactical level, *i.e.*, under incomplete information, and machine learning is used to address the stochasticity of

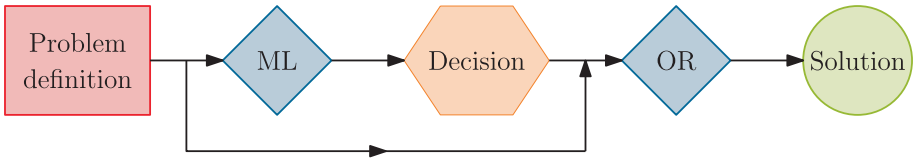


Fig. 8. The machine learning model is used to augment an operation research algorithm with valuable pieces of information.

the problem arising from missing some of the state variables in the observed input. The authors use operational solutions, *i.e.*, solutions to the deterministic version of the problem, and aggregate them to provide (tactical) solution targets to the machine learning model. As explained in their paper, the highest level of description of the solution is its cost, whereas the lowest (operational) is the knowledge of values for all its variables. Then, the authors place themselves in the middle and predict an aggregation of variables (tactical) that corresponds to the stochastic version of their specific problem. Furthermore, the nature of the application requires to output solutions in real time, which is not possible either for the stochastic version of the load planning problem or its deterministic variant when using state-of-the-art mixed-integer linear programming solvers. Then, machine learning turns out to be suitable for obtaining accurate solutions with short computing times because some of the complexity is addressed offline, *i.e.*, in the learning phase, and the run-time (inference) phase is extremely quick. Finally, note that in Larsen et al. (2018) an multilayer perceptron, *i.e.*, a feedforward neural network, is used to process the input instance as a vector, hence integrating very little prior knowledge about the problem structure.

3.2.2. Learning to configure algorithms

In many cases, using only machine learning to tackle the problem may not be the most suitable approach. Instead, machine learning can be applied to provide additional pieces of information to a combinatorial optimization algorithm as illustrated in Fig. 8. For example, machine learning can provide a parametrization of the algorithm (in a very broad sense).

Algorithm configuration, detailed in Bischl et al. (2016); Hoos (2012), is a well studied area that captures the setting presented here. Complex optimization algorithms usually have a set of parameters left constant during optimization (in machine learning they are called hyper-parameters). For instance, this can be the aggressiveness of the pre-solving operations (usually controlled by a single parameter) of an mixed-integer linear programming solver, or the learning rate / step size in gradient descent methods. Carefully selecting their value can dramatically change the performance of the optimization algorithm. Hence, the algorithm configuration community started looking for good default parameters. Then good default parameters for different cluster of similar problem instances. From the machine learning point of view, the former is a constant regression, while the second is a piece-wise constant nearest neighbors regression. The natural continuation was to learn a regression mapping problem instances to algorithm parameters.

In this context, Kruber, Lübbecke, and Parmentier (2017) use machine learning on mixed-integer linear programming instances to estimate beforehand whether or not applying a Dantzig-Wolf decomposition will be effective, *i.e.*, will make the solving time faster. Decomposition methods can be powerful but deciding if and how to apply them depends on many ingredients of the instance and of its formulation and there is no clear cut way of optimally making such a decision. In their work, a data point is represented as a fixed length vector with features representing instance and tentative decomposition statistics. In another example, in the context of mixed-integer quadratic programming, Bonami, Lodi, and Zarpellon (2018) use machine learning to decide if linearizing the problem will solve faster.

When the quadratic programming problem given by the relaxation is convex, *i.e.*, the quadratic objective matrix is semidefinite positive, one could address the problem by a branch-and-bound algorithm that solves quadratic programming relaxations[7] to provide lower bounds. Even in this convex case, it is not clear if quadratic programming branch-and-bound would solve faster than linearizing the problem (by using McCormick (1976) inequalities) and solving an equivalent mixed-integer linear programming. This is why machine learning is a great candidate here to fill the knowledge gap. In both papers (Bonami et al., 2018; Kruber et al., 2017), the authors experiment with different machine learning models, such as support vector machines and random forests, as is good practice when no prior knowledge is embedded in the model.

The heuristic building framework used in Karapetyan et al. (2017) and Mascia et al. (2014), already presented in Section 3.1.2, can be understood under this eye. Indeed, it can be seen as a large parametric heuristic, configured by the transition probabilities in the former case, and by the parameter representing a sentence in the latter.

As previously stated, the parametrization of the combinatorial optimization algorithm provided by machine learning is to be understood in a very broad sense. For instance, in the case of radiation therapy for cancer treatment, Mahmood, Babier, McNiven, Diamant, and Chan (2018) use machine learning to produce candidate therapies that are afterward refined by a combinatorial optimization algorithm into a deliverable plan. Namely, a generative adversarial network is used to color CT scan images into a potential radiation plan, then, inverse optimization (Ahuja & Orlin, 2001) is applied on the result to make the plan feasible (Chan, Craig, Lee, & Sharpe, 2014). In general, generative adversarial network are made of two distinct networks: one (the generator) generates images, and another one (the discriminator) discriminates between the generated images and a dataset of real images. Both are trained alternatively: the discriminator through a usual supervised objective, while the generator is trained to fool the discriminator. In Mahmood et al. (2018), a particular type of generative adversarial network (conditional generative adversarial network) is used to provide coloring instead of random images. The interested reader is referred to Creswell et al. (2018) for an overview on generative adversarial network.

We end this section by noting that an machine learning model used for learning some representation may in turn use as features pieces of information given by another combinatorial optimization algorithm, such as the decomposition statistics used in Kruber et al. (2017), or the linear programming information in Bonami et al. (2018). Moreover, we remark that, in the satisfiability context, the learning of the type of algorithm to execute on a particular cluster of instances has been paired with the learning of the

[7] Note that convex quadratic programming can be solved in polynomial time.

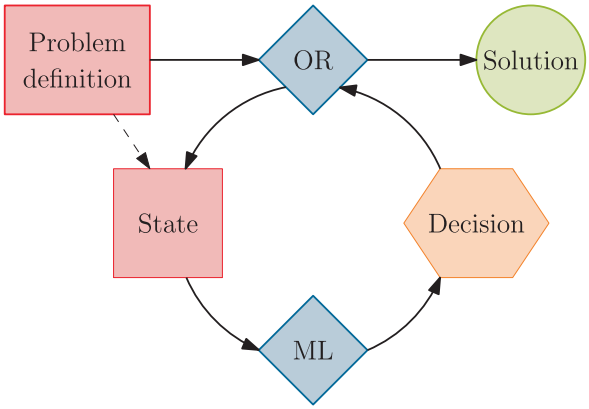


Fig. 9. The combinatorial optimization algorithm repeatedly queries the same machine learning model to make decisions. The machine learning model takes as input the current state of the algorithm, which may include the problem definition.

parameters of the algorithm itself, see, *e.g.*, Ansótegui, Heymann, Pon, Sellmann, and Tierney (2019); Ansótegui, Pon, Sellmann, and Tierney (2017).

3.2.3. Machine learning alongside optimization algorithms

To generalize the context of the previous section to its full potential, one can build combinatorial optimization algorithms that repeatedly call an machine learning model throughout their execution, as illustrated in Fig. 9. A master algorithm controls the high-level structure while frequently calling an machine learning model to assist in lower level decisions. The key difference between this approach and the examples discussed in the previous section is that the *same machine learning model* is used by the combinatorial optimization algorithm to make the same type of decisions a number of times in the order of the number of iterations of the algorithm. As in the previous section, nothing prevents one from applying additional steps before or after such an algorithm.

This is clearly the context of the branch-and-bound tree for mixed-integer linear programming, where we already mentioned how the task of selecting the branching variable is either too heuristic or too slow, and is therefore a good candidate for learning (Lodi & Zarpellon, 2017). In this case, the general algorithm remains a branch-and-bound framework, with the same software architecture and the same guarantees on lower and upper bounds, but the branching decisions made at every node are left to be learned. Likewise, the work of Hottung et al. (2017) learning both a branching policy *and* value network for heuristic tree search undeniably fits in this context. Another important aspect in solving mixed-integer linear programming is the use of primal heuristics, *i.e.*, algorithms that are applied in the branch-and-bound nodes to find feasible solutions, without guarantee of success. On top of their obvious advantages, good solutions also give tighter upper bounds (for minimization problems) on the solution value and make more pruning of the tree possible. Heuristics depend on the branching node (as branching fix some variables to specific values), so they need to be run frequently. However, running them too often can slow down the exploration of the tree, especially if their outcome is negative, *i.e.*, no better upper bound is detected. Khalil, Dilkina, Nemhauser, Ahmed, and Shao (2017b) build an machine learning model to predict whether or not running a given heuristic will yield a better solution than the best one found so far and then greedily run that heuristic whenever the outcome of the model is positive.

The approximation used by Baltean-Lugojan et al. (2018), already discussed in Section 3.2.1, is an example of predicting a high-level description of the solution to an optimization problem, namely the objective value. Nonetheless, the goal is to solve the original quadratic programming. Thus, the learned model is queried repeatedly to select promising cutting planes. The machine learning model is used only to select promising cuts, but once selected, cuts are added to the linear programming relaxation, thus embedding the machine learning outcome into an exact algorithm. This approach highlights promising directions for this type of algorithm. The decision learned is critical because adding the best cutting planes is necessary for solving the problem fast (or fast enough, because in the presence of NP-hard problems, optimization may time out before any meaningful solving). At the same time, the approximate decision (often in the form of a probability) does not compromise the exactness of the algorithm: any cut added is guaranteed to be valid. This setting leaves room for machine learning to thrive, while reducing the need for guarantees from the machine learning algorithms (an active and difficult area of research). In addition, note that, the approach in Larsen et al. (2018) is part of a master algorithm in which the machine learning is iteratively invoked to make booking decisions in real time. The work of Khalil et al. (2017a), presented in Section 3.1.2, also belongs to this setting, even if the resulting algorithm is heuristic. Indeed, an machine learning model is asked to select the most relevant node, while a master algorithm maintains the partial tour, computes its length, *etc*. Because the master algorithm is very simple, it is possible to see the contribution as an end-to-end method, as stated in Section 3.2.1, but it can also be interpreted more generally as done here.

Presented in Section 3.1.2, and mentioned in the previous section, the Markov Chain framework for building heuristics from Karapetyan et al. (2017) can also be framed as repeated decisions. The transition matrix can be queried and sampled from in order to transition from one state to another, *i.e.*, to make the low level decisions of choosing the next move. The three distinctions made in this Section 3.2 are general enough that they can overlap. Here, the fact that the model operates on internal state transitions, yet is learned globally, is what makes it hard to analyze.

Before ending this section, it is worth mentioning that learning recurrent algorithmic decisions is also used in the deep learning community, for instance in the field of meta-learning to decide how to apply gradient updates in stochastic gradient descent (Andrychowicz et al., 2016; Li & Malik, 2017; Wichrowska et al., 2017).

4. Learning objective

In the previous section, we have surveyed the existing literature by orthogonally grouping the main contributions of machine learning for combinatorial optimization into families of approaches, sometimes with overlaps. In this section, we formulate and study the objective that drives the learning process.

4.1. Multi-instance formulation

In the following, we introduce an abstract learning formulation (inspired from Bischl et al. (2016)). How would an machine learning practitioner compare optimization algorithms? Let us define $\mathcal{I}$ to be a set of problem instances, and P a probability distribution over $\mathcal{I}$. These are the problems that we care about, weighted by a probability distribution, reflecting the fact that, in a practical application, not all problems are as likely. In practice, $\mathcal{I}$ or P are inaccessible, but we can observe some samples from P, as motivated in the introduction with the Montreal delivery company. For a set of algorithms $\mathcal{A}$, let $m : \mathcal{I} \times \mathcal{A} \to \mathbb{R}$ be a measure of the performance of an algorithm on a problem instance (lower is better). This could be the objective value of the best solution found, but could also incorporate elements from optimality bounds, absence of results,

running times, and resource usage. To compare $a_1,\ a_2 \in \mathcal{A}$, an machine learning practitioner would compare $\mathbb{E}_{i\sim P}\ m(i, a_1)$ and $\mathbb{E}_{i\sim P}\ m(i, a_2)$, or equivalently

$$\min_{a\in\{a_1,a_2\}} \mathbb{E}_{i\sim P}\ m(i, a). \tag{4}$$

Because measuring these quantities is not tractable, one will typically use empirical estimates instead, by using a finite dataset D_{train} of independent instances sampled from P

$$\min_{a\in\{a_1,a_2\}} \sum_{i\in D_{train}} \frac{1}{|D_{train}|} m(i, a). \tag{5}$$

This is intuitive and done in practice: collect a dataset of problem instances and compare say, average running times. Of course, such expectation can be computed for different datasets (different $\mathcal{I}$'s and P's), and different measures (different m's).

This is already a learning problem. The more general one that we want to solve through leaning is

$$\min_{a\in\mathcal{A}} \mathbb{E}_{i\sim P}\ m(i, a). \tag{6}$$

Instead of comparing between two algorithms, we may compare among an uncountable, maybe non-parametric, space of algorithms. To see how we come up with so many algorithms, we have to look at the algorithms in Section 3, and think of the machine learning model space over which we learn as defining parametrizing the algorithm space $\mathcal{A}$. For instance, consider the case of learning a branching policy π for branch-and-bound. If we define the policy to be a neural network with a set of weights $\theta \in \mathbb{R}^p$, then we obtain a parametric branch-and-bound algorithm $a(\pi_\theta)$ and (6) becomes

$$\min_{\theta\in\mathbb{R}^p} \mathbb{E}_{i\sim P}\ m(i, a(\pi_\theta)). \tag{7}$$

Unfortunately, solving this problem is hard. On the one hand, the performance measure m is most often not differentiable and without closed form expression. We discuss this in Section 4.2. On the other hand, computing the expectation in (6) is intractable. As in (5), one can use an empirical distribution using a finite dataset, but that leads to *generalization* considerations, as explained in Section 4.3.

Before we move on, let us introduce a new element to make (6) more general. That formula suggests that, once given an instance, the outcome of the performance measure is deterministic. That is unrealistic for multiple reasons. The performance measure could itself incorporate some source of randomness due to external factors, for instance with running times which are hardware and system dependent. The algorithm could also incorporate non negligible sources of randomness, if it is designed to be stochastic, or if some operations are non deterministic, or to express the fact that the algorithm should be robust to the choice of some external parameters. Let τ be that source of randomness, $\pi \in \Pi$ the internal policy being learned, and $a(\pi, \tau)$ the resulting algorithm, then we can reformulate (6) as

$$\min_{\pi\in\Pi}\ \mathbb{E}_{i\sim P}[\,\mathbb{E}_\tau[\,m(i,\ a(\pi, \tau))\ |\ i\,]\,]. \tag{8}$$

In particular, when learning repeated decisions, as in Section 3.2.3, this source of randomness can be expressed along the trajectory followed in the markov decision process, using the dynamics of the environment $p(s', r|a, s)$ (see Fig. 2). The addition made in (8) will be useful for the discussion on generalization in Section 4.3.

4.2. Surrogate objectives

In the previous section, we have formulated a proper learning objective. Here, we try to relate that objective to the learning methods of Section 3.1, namely, demonstration and experience. If the usual learning metrics of an machine learning model, *e.g.*, accuracy for classification in supervised (imitation) learning, is improving, does it mean that the performance metric of (6) is also improving?

A straightforward approach for solving (8) is that of reinforcement learning (including direct optimization methods), as surveyed in Section 3.1.2. The objective from (6) can be optimized directly on experience data by matching the total return to the performance measure. Sometimes, a single final reward can naturally be decoupled across the trajectory. For instance, if the performance objective of a branch-and-bound variable selection policy is to minimize the number of opened nodes, then the policy can receive a reward discouraging an increase in the number of nodes, hence giving an incentive to select variables that lead to pruning. However, that may not be always possible, leaving only the option of delaying a single reward to the end of the trajectory. This sparse reward setting is challenging for reinforcement learning algorithms, and one might want to design a surrogate reward signal to encourage intermediate accomplishments. This introduces some discrepancies, and the policy being optimized may learn a behavior not intended by the algorithm designer. There is *a priori* no relationship between two reward signals. One needs to make use of their intuition to design surrogate signals, *e.g.*, minimizing the number of branch-and-bound nodes *should* lead to smaller running times. Reward shaping is an active area of research in reinforcement learning, yet it is often performed by a number of engineering tricks.

In the case of learning a policy from a supervised signal from expert demonstration, the performance measure m does not even appear in the learning problem that is solved. In this context, the goal is to optimize a policy $\pi \in \Pi$ in the action space to mimic an expert policy π_e (as first introduced with Fig. 5)

$$\min_{\pi\in\Pi}\ \mathbb{E}_{i\sim P}[\,\mathbb{E}_s[\,\ell(\pi(s), \pi_e(s))\ |\ i, \pi_e\,]\,], \tag{9}$$

where ℓ is a task dependent loss (classification, regression, *etc.*). We have emphasized that the state S is conditional, not only on the instance, but also on the expert policy π_e used to collect the data. Intuitively, the better the machine learning model learns, *i.e.*, the better the policy imitates the expert, the closer the final performance of the learned policy should be to the performance of the expert. Under some conditions, it is possible to relate the performance of the learned policy to the performance of the expert policy, but covering this aspect is out of the scope of this paper. The opposite is not true, if learning fails, the policy may still turn out to perform well (by encountering an alternative good decision). Indeed, when making a decision with high surrogate objective error, the learning will be fully penalized when, in fact, the decision could have good performances by the original metric. For that reason, it is capital to report the performance metrics. For example, we surveyed in Section 3.2.2 the work of Bonami et al. (2018) where the authors train a classifier to predict if a mixed integer quadratic problem instance should be linearized or not. The targets used for the learner are computed optimally by solving the problem instance in both configurations. Simply reporting the classification accuracy is not enough. Indeed, this metric gives no information on the impact a misclassification has on running times, the metric used to compute the targets. In the binary classification, a properly classified example could also happen to have unsignificant difference between the running times of the two configurations. To alleviate this issue, the authors also introduce a category where running times are not significatively different (and report the real running times). A continuous extension would be to learn a regression of the solving time. However, learning this regression now means that the final algorithm needs to optimize over the set of decisions to find the best one. In reinforcement learning, this is analoguous to learning a value function (see Section 2.2).

Applying the same reasoning to repeated decisions is better understood with the complete reinforcement learning theory.

4.3. On generalization

In Section 4.1, we have claimed that the probability distribution in (6) is inaccessible and needs to be replaced by the empirical probability distribution over a finite dataset D_{train}. The optimization problem solved is

$$\min_{a \in \mathcal{A}} \sum_{i \in D_{train}} \frac{1}{|D_{train}|} m(i, a). \tag{10}$$

As pointed out in Section 2.2, when optimizing over the empirical probability distribution, we risk having a low performance measure on the finite number of problem instances, *regardless of the true probability distribution*. In this case, the *generalization* error is high because of the discrepancy between the training performances and the true expected performances (overfitting). To control this aspect, a validation set D_{valid} is introduced to compare a finite number of candidate algorithms based on estimates of generalization performances, and a test set D_{test} is used for estimating the generalization performances of the selected algorithm.

In the following, we look more intuitively at generalization in machine learning for combinatorial optimization, and its consequences. To make it easier, let us recall different learning scenarios. In the introduction, we have motivated the Montreal delivery company example, where the problems of interest are from an unknown probability distribution of Montreal traveling salesman problem. This is a very restricted set of problems, but enough to deliver value for this business. Much more ambitious, we may want our policy learned on a finite set of instances to perform well (generalize) to any "*real-world*" mixed-integer linear programming instance. This is of interest if you are in the business of selling mixed-integer linear programming solvers and want the branching policy to perform well for as many of your clients as possible. In both cases, generalization applies to the instances that are not known to the algorithm implementer. These are the only instances that we care about; the one used for training are already solved. The topic of probability distribution of instances also appears naturally in stochastic programming/optimization, where uncertainty about the problem is modeled through probability distributions. Scenario generation, an essential way to solve this type of optimization programs, require sampling from this distribution and solving the associated problem multiple times. Nair, Dvijotham, Dunning, and Vinyals (2018) take advantage of this repetitive process to learn an end-to-end model to solve the problem. Their model is composed of a local search and a local improvement policy and is trained through reinforcement learning. Here, generalization means that, during scenario generation, the learned search beats other approaches, hence delivering an overall faster stochastic programming algorithm. In short, *learning without generalization is pointless!*

When the policy generalizes to other problem instances, it is no longer a problem if training requires additional computation for solving problem instances because, learning can be decoupled from solving as it can be done offline. This setting is promising as it could give a policy to use out of the box for similar instances, while keeping the learning problem to be handled beforehand while remaining hopefully reasonable. When the model learned is a simple mapping, as is the case in Section 3.2.2, generalization to new instances, as previously explained, can be easily understood. However, when learning sequential decisions, as in Section 3.2.3, there are intricate levels of generalization. We said that we want the policy to generalize to new instances, but the policy also needs to generalize to internal states of the algorithm for a single instance, even if the model can be learned from complete optimization trajectories, as formulated by (8). Indeed, complex algorithms can have unexpected sources of randomness, even if they are designed to be deterministic. For instance, a numerical approximation may perform differently if the version of some underlying numerical library is changed or because of asynchronous computing, such as when using Graphical Processing Units (Nagarajan, Warnell, & Stone, 2019). Furthermore, even if we can achieve perfect replicability, we do not want the branching policy to break if some other parameters of the solver are set (slightly) differently. At the very least, we want the policy to be robust to the choice of the random seed present in many algorithms, including mixed-integer linear programming solvers. These parameters can therefore be modeled as random variables. Because of these nested levels of generalization, one appealing way to think about the training data from multiple instances is like separate tasks of a multi-task learning setting. The different tasks have underlying aspects in common, and they may also have their own peculiar quirks. One way to learn a single policy that generalizes within a distribution of instances is to take advantage of these commonalities. Generalization in reinforcement learning remains a challenging topic, probably because of the fuzzy distinction between a multi-task setting, and a large environment encompassing all of the tasks.

Choosing how ambitious one should be in defining the characteristics of the distribution is a hard question. For instance, if the Montreal company expands its business to other cities, should they be considered as separate distributions, and learn one branching policy per city, or only a single one? Maybe one per continent? Generalization to a larger variety of instances is challenging and requires more advanced and expensive learning algorithms. Learning an array of machine learning models for different distributions asociated with a same task means of course more models to train, maintain, and deploy. The same goes with traditional combinatorial optimization algorithms, an mixed-integer linear programming solver on its own is not the best performing algorithm to solve traveling salesman problem, but it works across all mixed-integer linear programming problems. It is too early to provide insights about how broad the considered distributions should be, given the limited literature in the field. For scholars generating synthetic distributions, two intuitive axes of investigation are "*structure*" and "*size*". A traveling salesman problem and a scheduling problem seem to have fairly different structure, and one can think of two planar euclidean traveling salesman problem to be way more similar. Still, two of these traveling salesman problem can have dramatically different sizes (number of nodes). For instance, Gasse et al. (2019) assess their methodology independently on three distributions. Each training dataset has a specific problem structure (set covering, combinatorial auction, and capacitated facility location), and a fixed problem size. The problem instance generators used are state-of-art and representative of real-world instances. Nonetheless, when they evaluate their learned algorithm, the authors push the test distributions to larger sizes. The idea behind this is to gauge if the model learned is able to generalize to a larger, more practical, distribution, or only perform well on the restricted distribution of problems of the same size. The answer is largely affirmative.

4.4. Single instance learning

An edge case that we have not much discussed yet is the single instance learning framework. This might be the case for instance for planning the design of a single factory. The factory would only be built once, with very peculiar requirements, and the planners are not interested to relate this to other problems. In this case, one can make as many runs (episodes) and as many calls to a potential expert or simulator as one wants, but ultimately one only cares about solving this one instance. Learning a policy for a

single instance should require a simpler machine learning model, which could thus require less training examples. Nonetheless, in the single instance case, one learns the policy from scratch at every new instance, actually incorporating learning (not learned models but really the learning process itself) into the end algorithm. This means starting the timer at the beginning of learning and competing with other solvers to get the solution the fastest (or get the best results within a time limit). This is an edge scenario that can only be employed in the setting of the Section 3.2.3, where machine learning is embedded *inside* a combinatorial optimization algorithm; otherwise there would be only one training example! There is therefore no notion of generalization to other problem instances, so (6) is not the learning problem being solved. Nonetheless, the model still needs to generalize to *unseen states* of the algorithm. Indeed, if the model was learned from all states of the algorithm that are needed to solve the problem, then the problem is already solved at training time and learning is therefore fruitless. This is the methodology followed by Khalil et al. (2016), introduced in Section 3.1.1, to learn an instance-specific branching policy. The policy is learned from strong-branching at the top of the branch-and-bound tree, but needs to generalize to the state of the algorithm at the bottom of the tree, where it is used. However, as for all combinatorial optimization algorithms, a fair comparison to another algorithm can only be done on an independent dataset of instances, as in (4). This is because through human trials and errors, the data used when building the algorithm leaks into the design of the algorithm, even without explicit learning components.

4.5. Fine tuning and meta-learning

A compromise between instance-specific learning and learning a generic policy is what we typically have in multi-task learning: some parameters are shared across tasks and some are specific to each task. A common way to do that (in the transfer learning scenario) is to start from a generic policy and then adapt it to the particular instance by a form of *fine-tuning* procedure: training proceeds in two stages, first training the generic policy across many instances from the same distribution, and then continuing training on the examples associated with a given instance on which we are hoping to get more specialized and accurate predictions.

Machine learning advances in the areas of meta-learning and transfer learning are particularly interesting to consider here. Meta-learning considers two levels of optimization: the inner loop trains the parameters of a model on the training set in a way that depends on meta-parameters, which are themselves optimized in an outer loop (*i.e.*, obtaining a gradient for each completed inner-loop training or update). When the outer loop's objective function is performance on a validation set, we end up training a system so that it will generalize well. This can be a successful strategy for generalizing from very few examples if we have access to many such training tasks. It is related to transfer learning, where we want that what has been learned in one or many tasks helps improve generalization on another. These approaches can help rapidly adapt to a new problem, which would be useful in the context of solving many mixed-integer linear programming instances, seen as many related tasks.

To stay with the branching example on mixed-integer linear programming, one may not want the policy to perform well out of the box on new instances (from the given distribution). Instead, one may want to learn a policy offline that can be adapted to a new instance in a few training steps, every time it is given one. Similar topics have been explored in the context of automatic configuration tools. Fitzgerald, Malitsky, O'Sullivan, and Tierney (2014) study the automatic configuration in the lifelong learning context (a form of sequential transfer learning). The automatic configuration algorithm is augmented with a set of previous configurations that are prioritized on any new problem instance. A score reflecting past performances is kept along every configuration. It is designed to retain configurations that performed well in the past, while letting new ones a chance to be properly evaluated. The automatic configuration optimization algorithm used by Lindauer and Hutter (2018) requires training an empirical cost model mapping the Cartesian product of parameter configurations and problem instances to expected algorithmic performance. Such a model is usually learned for every cluster of problem instance that requires configuring. Instead, when presented with a new cluster, the authors combine the previously learned cost models and the new one to build an ensemble model. As done by Fitzgerald et al. (2014), the authors also build a set of previous configurations to prioritize, using an empirical cost model to fill the missing data. This setting, which is more general than not performing any adaptation of the policy, has potential for better generalization. Once again, the scale on which this is applied can vary depending on ambition. One can transfer on very similar instances, or learn a policy that transfers to a vast range of instances.

Meta-learning algorithms were first introduced in the 1990s (Bengio, Bengio, Cloutier, & Gecsei, 1991; Schmidhuber, 1992; Thrun & Pratt, 1998) and have since then become particularly popular in machine learning, including, but not limited to, learning a gradient update rule (Andrychowicz et al., 2016; Hochreiter, Younger, & Conwell, 2001), few shot learning (Ravi & Larochelle, 2017), and multi-task reinforcement learning (Finn, Abbeel, & Levine, 2017).

4.6. Other metrics

Other metrics from the process of learning itself are also relevant, such as how fast the learning process is, the sample complexity (number of examples required to properly fit the model), *etc*. As opposed to the metrics suggested earlier in this section, these metrics provide us with information not about final performance, but about offline computation or the number of training examples required to obtain the desired policy. This information is, of course, useful to calibrate the effort in integrating machine learning into combinatorial optimization algorithms.

5. Methodology

In the previous section, we have detailed the theoretical learning framework of using machine learning in combinatorial optimization algorithms. Here, we provide some additional discussion broadening some previously made claims.

5.1. Demonstration and experience

In order to learn a policy, we have highlighted two methodologies: demonstration, where the expected behavior is shown by an expert or oracle (sometimes at a considerable computational cost), and experience, where the policy is learned through trial and error with a reward signal.

In the demonstration setting, the performance of the learned policy is bounded by the expert, which is a limitation when the expert is not optimal. More precisely, without a reward signal, the imitation policy can only hope to marginally outperform the expert (for example because the learner can reduce the variance of the answers across similarly-performing experts). The better the learning, the closer the performance of the learner to the expert's. This means that imitation alone should be used only if it is significantly faster than the expert to compute the policy. Furthermore, the performance of the learned policy may not generalize well to unseen examples and small variations of the task and may be unstable due to accumulation of errors. This is because in (9), the

data was collected according to the expert policy π_e, but when run over multiple repeated decisions, the distribution of states becomes that of the learned policy. Some downsides of supervised (imitation) learning can be overcome with more advanced algorithms, including active methods to query the expert as an oracle to improve behavior in uncertain states. The part of imitation learning presented here is limited compared to the current literature in machine learning.

On the contrary, with a reward, the algorithm learns to optimize for that signal and can potentially outperform any expert, at the cost of a much longer training time. Learning from a reward signal (experience) is also more flexible when multiple decisions are (almost) equally good in comparison with an expert that would favor one (arbitrary) decision. Experience is not without flaws. In the case where policies are approximated (*e.g.*, with a neural network), the learning process may get stuck around poor solutions if exploration is not sufficient or solutions which do not generalize well are found. Furthermore, it may not always be straightforward to define a reward signal. For instance, sparse rewards may be augmented using reward shaping or a curriculum in order to value intermediate accomplishments (see Section 2.2).

Often, it is a good idea to start learning from demonstrations by an expert, then refine the policy using experience and a reward signal. This is what was done in the original AlphaGo paper (Silver et al., 2016), where human knowledge is combined with reinforcement learning. The reader is referred to Hussein, Gaber, Elyan, and Jayne (2017) for a survey on imitation learning covering most of the discussion in this section.

5.2. Partial observability

We mentioned in Section 2.2 that sometimes the states of an markov decision process are not fully observed and the Markov property does not hold, *i.e.*, the probability of the next observation, conditioned on the current observation and action, is not equal to the probability of the next observation, conditioned on all past observations and actions. An immediate example of this can be found in any environment simulating physics: a single frame/image of such an environment is not sufficient to grasp notions such as velocity and is therefore not sufficient to properly estimate the future trajectory of objects. It turns out that, on real applications, partial observability is closer to the norm than to the exception, either because one does not have access to a true state of the environment, or because it is not computationally tractable to represent and needs to be approximated. A straightforward way to tackle the problem is to compress all previous observations using an recurrent neural network. This can be applied in the imitation learning setting, as well as in reinforcement learning, for instance by learning a recurrent policy (Wierstra, Förster, Peters, & Schmidhuber, 2010).

How does this apply in the case where we want to learn a policy function making decisions for a combinatorial optimization algorithm? On the one hand, one has full access to the state of the algorithm because it is represented in exact mathematical concepts, such as constraints, cuts, solutions, branch-and-bound tree, *etc*. On the other hand, these states can be exponentially large. This is an issue in terms of computations and generalization. Indeed, if one does want to solve problems quickly, one needs to have a policy that is also fast to compute, especially if it is called frequently as is the case for, say, branching decisions. Furthermore, considering too high-dimensional states is also a statistical problem for learning, as it may dramatically increase the required number of samples, decrease the learning speed, or fail altogether. Hence, it is necessary to keep these aspects in mind while experimenting with different representations of the data.

5.3. Exactness and approximation

In the different examples we have surveyed, machine learning is used in both exact and heuristic frameworks, for example Baltean-Lugojan et al. (2018) and Larsen et al. (2018), respectively. Getting the output of an machine learning model to respect advanced types of constraints is a hard task. In order to build exact algorithms with machine learning components, it is necessary to apply the machine learning where all possible decisions are valid. Using only machine learning as surveyed in Section 3.2.1 cannot give any optimality guarantee, and only weak feasibility guarantees (see Section 6.1). However, applying machine learning to select or parametrize a combinatorial optimization algorithm as in Section 3.2.2 will keep exactness if all possible choices that machine learning discriminate lead to complete algorithms. Finally, in the case of repeated interactions between machine learning and combinatorial optimization surveyed in Section 3.2.3, all possible decisions must be valid. For instance, in the case of mixed-integer linear programming, this includes branching *among fractional variables* of the linear programming relaxation, selecting the node to explore *among open branching nodes* (He et al., 2014), deciding on the frequency to run heuristics on the branch-and-bound nodes (Khalil et al., 2017b), selecting cutting planes *among valid inequalities* (Baltean-Lugojan et al., 2018), removing previous cutting planes *if they are not original constraints or branching decision*, *etc*. A counter-example can be found in the work of Hottung et al. (2017), presented in Section 3.1.1. In their branch-an-bound framework, bounding is performed by an approximate machine learning model that can overestimate lower bounds, resulting in invalid pruning. The resulting algorithm is therefore not an exact one.

6. Challenges

In this section, we are reviewing some of the algorithmic concepts previously introduced by taking the viewpoint of their associated challenges.

6.1. Feasibility

In Section 3.2.1, we pointed out how machine learning can be used to directly output solutions to optimization problems. Rather than learning the solution, it would be more precise to say that the algorithm is learning a *heuristic*. As already repeatedly noted, the learned algorithm does not give any guarantee in terms of optimality, but it is even more critical that feasibility is not guaranteed either. Indeed, we do not know how far the output of the heuristic is from the optimal solution, or if it even respects the constraints of the problem. This can be the case for every heuristic and the issue can be mitigated by using the heuristic within an exact optimization algorithm (such as branch and bound).

Finding feasible solutions is not an easy problem (theoretically NP-hard for mixed-integer linear programming), but it is even more challenging in machine learning, especially by using neural networks. Indeed, trained with gradient descent, neural architectures must be designed carefully in order not to break differentiability. For instance, both pointer networks (Vinyals et al., 2015) and the Sinkhorn layer (Emami & Ranka, 2018) are complex architectures used to make a network output a permutation, a constraint easy to satisfy when writing a classical combinatorial optimization heuristic.

6.2. Modelling

In machine learning, in general, and in deep learning, in particular, we know some good prior for some given problems. For

instance, we know that a convolutional neural network is an architecture that will learn and generalize more easily than others on image data. The problems studied in combinatorial optimization are different from the ones currently being addressed in machine learning, where most successful applications target natural signals. The architectures used to learn good policies in combinatorial optimization might be very different from what is currently used with deep learning. This might also be true in more subtle or unexpected ways: it is conceivable that, in turn, the optimization components of deep learning algorithms (say, modifications to stochastic gradient descent) could be different when deep learning is applied to the combinatorial optimization context.

Current deep learning already provides many techniques and architectures for tackling problems of interest in combinatorial optimization. As pointed out in Section 2.2, techniques such as parameter sharing made it possible for neural networks to process sequences of variable length with recurrent neural network or, more recently, to process graph structured data through graph neural network. Processing graph data is of uttermost importance in combinatorial optimization because many problems are formulated (represented) on graphs. For a very general example, Selsam et al. (2018) represent a satisfiability problem using a bipartite graph on variables and clauses. This can generalize to mixed-integer linear programming, where the constraint matrix can be represented as the adjacency matrix of a bipartite graph on variables and constraints, as done in Gasse et al. (2019).

6.3. Scaling

Scaling to larger problems can be a challenge. If a model trained on instances up to some size, say traveling salesman problem up to size fifty nodes, is evaluated on larger instances, say traveling salesman problem of size a hundred, five hundred nodes, *etc*, the challenge exists in terms of generalization, as mentioned in Section 4.3. Indeed, all of the papers tackling traveling salesman problem through machine learning and attempting to solve larger instances see degrading performance as size increases much beyond the sizes seen during training (Bello et al., 2017; Khalil et al., 2017a; Kool & Welling, 2018; Vinyals et al., 2015). To tackle this issue, one may try to learn on larger instances, but this may turn out to be a computational and generalization issue. Except for very simple machine learning models and strong assumptions about the data distribution, it is impossible to know the computational complexity and the sample complexity, i.e. the number of observations that learning requires, because one is unaware of the exact problem one is trying to solve, *i.e.*, the true data generating distribution.

6.4. Data generation

Collecting data (for example instances of optimization problems) is a subtle task. Larsen et al. (2018) claim that "*sampling from historical data is appropriate when attempting to mimic a behavior reflected in such data*". In other words, given an external process on which we observe instances of an optimization problem, we can collect data to train some policy needed for optimization, and expect the policy to generalize on future instances of this application. A practical example would be a business that frequently encounters optimization problems related to their activities, such as the Montreal delivery company example used in the introduction.

In other cases, *i.e.*, when we are not targeting a specific application for which we would have historical data, how can we proactively train a policy for problems that we do not yet know of? As partially discussed in Section 4.3, we first need to define to which family of instances we want to generalize over. For instance, we might decide to learn a cutting plane selection policy for Euclidian traveling salesman problem problems. Even so, it remains a complex effort to generate problems that capture the essence of real applications. Moreover, combinatorial optimization problems are high dimensional, highly structured, and troublesome to visualize. The sole exercise of generating graphs is already a complicated one! The topic has nonetheless received some interest. Smith-Miles and Bowly (2015) claim that the confidence we can put in an algorithm "*depends on how carefully we select test instances*", but note however that too often, a new algorithm is claimed "*to be superior by showing that it outperforms previous approaches on a set of well-studied instances*". The authors propose a problem instance generating method that consists of: defining an instance feature space, visualizing it in two dimensions (using dimensionality reduction techniques such as principal component analysis), and using an evolutionary algorithm to drive the instance generation toward a pre-defined sub-space. The authors argue that the method is successful if the easy and hard instances can be easily separated in the reduced instance space. The methodology is then fruitfully applied to graph-based problems, but would require redefining evolution primitives in order to be applied to other type of problems. On the contrary, Malitsky, Merschformann, O'Sullivan, and Tierney (2016) propose a method to generate problem instances from the same probability distribution, in that case, the one of "*industrial*" boolean satisfiability problem instances. The authors use a large neighborhood search, using destruction and reparation primitives, to search for new instances. Some instance features are computed to classify whether the new instances fall under the same cluster as the target one.

Deciding how to represent the data is also not an easy task, but can have a dramatic impact on learning. For instance, how does one properly represent a branch-and-bound node, or even the whole branch-and-bound tree? These representations need to be expressive enough for learning, but at the same time, concise enough to be used frequently without excessive computations.

7. Conclusions

We have surveyed and highlighted how machine learning can be used to build combinatorial optimization algorithms that are partially learned. We have suggested that imitation learning alone can be valuable if the policy learned is significantly faster to compute than the original one provided by an expert, in this case a combinatorial optimization algorithm. On the contrary, models trained with a reward signal have the potential to outperform current policies, given enough training and a supervised initialization. Training a policy that generalizes to unseen problems is a challenge, this is why we believe learning should occur on a distribution small enough that the policy could fully exploit the structure of the problem and give better results. We believe end-to-end machine learning approaches to combinatorial optimization can be improved by using machine learning in combination with current combinatorial optimization algorithms to benefit from the theoretical guarantees and state-of-the-art algorithms already available.

Other than performance incentives, there is also interest in using machine learning as a modelling tool for discrete optimization, as done by Lombardi and Milano (2018), or to extract intuition and knowledge about algorithms as mentioned in Bonami et al. (2018); Khalil et al. (2017a).

Although most of the approaches we discussed in this paper are still at an exploratory level of deployment, at least in terms of their use in general-purpose (commercial) solvers, we strongly believe that this is just the beginning of a new era for combinatorial optimization algorithms.

Acknowlgedgments

The authors are grateful to Emma Frejinger, Simon Lacoste-Julien, Jason Jo, Laurent Charlin, Matteo Fischetti, Rémi Leblond, Michela Milano, Sébastien Lachapelle, Eric Larsen, Pierre Bonami, Martina Fischetti, Elias Khalil, Bistra Dilkina, Sebastian Pokutta, Marco Lübbecke, Andrea Tramontani, Dimitris Bertsimas and the entire CERC team for endless discussions on the subject and for reading and commenting a preliminary version of the paper.

References

Ahuja, R. K., & Orlin, J. B. (2001). Inverse optimization. *Operations Research, 49*(5), 771–783. https://doi.org/10.1287/opre.49.5.771.10607.

Andrychowicz, M., Denil, M., Gómez, S., Hoffman, M. W., Pfau, D., Schaul, T., … de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, & R. Garnett (Eds.), *Advances in neural information processing systems 29* (pp. 3981–3989). Curran Associates, Inc..

Ansótegui, C., Heymann, B., Pon, J., Sellmann, M., & Tierney, K. (2019). Hyper-Reactive Tabu Search for MaxSAT. In R. Battiti, M. Brunato, I. Kotsireas, & P. M. Pardalos (Eds.), *Learning and intelligent optimization*. In *Lecture Notes in Computer Science* (pp. 309–325). Cham: Springer International Publishing.

Ansótegui, C., Pon, J., Sellmann, M., & Tierney, K. (2017). Reactive Dialectic Search Portfolios for MaxSAT. In *Thirty-first AAAI conference on artificial intelligence*. https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14872.

Applegate, D., Bixby, R., Chvátal, V., & Cook, W. (2007). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.

Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *Iclr'2015, arxiv:1409.0473*.

Baltean-Lugojan, R., Misener, R., Bonami, P., & Tramontani, A. (2018). Strong Sparse Cut Selection via Trained Neural Nets for Quadratic Semidefinite Outer-Approximations. *Technical Report*. Imperial College, London.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., & Bengio, S. (2017). Neural Combinatorial Optimization with Reinforcement Learning. In *International conference on learning representations*. https://openreview.net/forum?id=Bk9mxlSFx.

Bengio, Y., Bengio, S., Cloutier, J., & Gecsei, J. (1991). Learning a synaptic learning rule. In *Ijcnn*. II–A969

Bischl, B., Kerschke, P., Kotthoff, L., Lindauer, M., Malitsky, Y., Fréchette, A., … Vanschoren, J. (2016). ASlib: A benchmark library for algorithm selection. *Artificial intelligence, 237*, 41–58. https://doi.org/10.1016/j.artint.2016.04.003. http://www.sciencedirect.com/science/article/pii/S0004370216300388.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. New York: Springer-Verlag.

Bonami, P., Lodi, A., & Zarpellon, G. (2018). Learning a Classification of Mixed-Integer Quadratic Programming Problems. In *Integration of constraint programming, artificial intelligence, and operations research*. In *Lecture Notes in Computer Science* (pp. 595–604). Cham: Springer. https://doi.org/10.1007/978-3-319-93031-2_43.

Chan, T. C. Y., Craig, T., Lee, T., & Sharpe, M. B. (2014). Generalized inverse multiobjective optimization with application to cancer therapy. *Operations research, 62*(3), 680–695. https://doi.org/10.1287/opre.2014.1267.

Conforti, M., Conrnuéjols, G., & Zambelli, G. (2014). *Integer programming*. Springer.

Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., & Bharath, A. A. (2018). Generative adversarial networks: an overview. *IEEE signal processing magazine, 35*(1), 53–65. https://doi.org/10.1109/MSP.2017.2765202.

Dai, H., Dai, B., & Song, L. (2016). Discriminative Embeddings of Latent Variable Models for Structured Data. In M. F. Balcan, & K. Q. Weinberger (Eds.), *Proceedings of The 33rd International Conference on Machine Learning*. In *Proceedings of Machine Learning Research: 48* (pp. 2702–2711). New York, USA: PMLR.

Dey, S., & Molinaro, M. (2018). Theoretical challenges towards cutting-plane selection. *Mathematical programming, 170*, 237–266.

Emami, P., & Ranka, S. (2018). *Learning permutations with sinkhorn policy gradient*. arXiv: 1805.07010[cs, stat].

Finn, C., Abbeel, P., & Levine, S. (2017). Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In D. Precup, & Y. W. Teh (Eds.), *Proceedings of the 34th International Conference on Machine Learning*. In *Proceedings of Machine Learning Research: 70* (pp. 1126–1135). International Convention Centre, Sydney, Australia: PMLR.

Fischetti, M., & Lodi, A. (2011). *Heuristics in mixed integer programming*. In J. J. Cochran, L. A. C. Jr., P. Keskinocak, J. P. Kharoufeh, & J. C. Smith (Eds.) ((vol.3, pp. 2199–2204)). Wiley Online Library. https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470400531.eorms0376.

Fitzgerald, T., Malitsky, Y., O'Sullivan, B., & Tierney, K. (2014). ReACT: Real-Time Algorithm Configuration through Tournaments. In *Seventh annual symposium on combinatorial search*. https://www.aaai.org/ocs/index.php/SOCS/SOCS14/paper/view/8910.

Fortun, M., & Schweber, S. S. (1993). Scientists and the legacy of world war ii: The case of operations research (or). *Social studies of science, 23*(4), 595–642. https://doi.org/10.1177/030631293023004001.

Gasse, M., Chételat, D., Ferroni, N., Charlin, L., & Lodi, A. (2019). Exact combinatorial optimization with graph convolutional neural networks. In H. Wallach, et al. (Eds.), *Advances in Neural Information Processing Systems 32 (NIPS 2019)* (pp. 15554–15566). New York: Curran Associates, Inc.. arXiv preprint arXiv:1906.01629.

Gendreau, M., & Potvin, J.-Y. (2010). *Handbook of metaheuristics (vol. 2)*. Springer. https://doi.org/10.1007/978-3-319-91086-4. https://www.springer.com/gp/book/9783319910857.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural Message Passing for Quantum Chemistry. In D. Precup, & Y. W. Teh (Eds.), *Proceedings of the 34th international conference on machine learning*. In *Proceedings of Machine Learning Research: 70* (pp. 1263–1272). International Convention Centre, Sydney, Australia: PMLR.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

He, H., Daume III, H., & Eisner, J. M. (2014). Learning to search in branch and bound algorithms. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems 27* (pp. 3293–3301). Curran Associates, Inc..

Hochreiter, S., Younger, A. S., & Conwell, P. R. (2001). Learning to learn using gradient descent. In G. Dorffner, H. Bischof, & K. Hornik (Eds.), *Artificial neural networks — ICANN 2001* (pp. 87–94). Berlin, Heidelberg: Springer Berlin Heidelberg.

Hoos, H. H. (2012). Automated algorithm configuration and parameter tuning. In Y. Hamadi, E. Monfroy, & F. Saubion (Eds.), *Autonomous search* (pp. 37–71). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-21434-9_3.

Hottung, A., Tanaka, S., & Tierney, K. (2017). Deep learning assisted heuristic tree search for the container pre-marshalling problem. *arXiv:1709.09972 [cs]*. ArXiv: 1709.09972.

Hussein, A., Gaber, M. M., Elyan, E., & Jayne, C. (2017). Imitation learning: a survey of learning methods. *ACM computing surveys, 50*(2), 21:1–21:35. https://doi.org/10.1145/3054912.

Karapetyan, D., Punnen, A. P., & Parkes, A. J. (2017). Markov chain methods for the bipartite boolean quadratic programming problem. *European journal of operational research, 260*(2), 494–506. https://doi.org/10.1016/j.ejor.2017.01.001. http://www.sciencedirect.com/science/article/pii/S0377221717300061.

Khalil, E., Dai, H., Zhang, Y., Dilkina, B., & Song, L. (2017a). Learning combinatorial optimization algorithms over graphs. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 30* (pp. 6348–6358). Curran Associates, Inc..

Khalil, E. B., Bodic, P. L., Song, L., Nemhauser, G., & Dilkina, B. (2016). Learning to Branch in Mixed Integer Programming. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. In *AAAI'16* (pp. 724–731). Phoenix, Arizona: AAAI Press.

Khalil, E. B., Dilkina, B., Nemhauser, G. L., Ahmed, S., & Shao, Y. (2017b). Learning to Run Heuristics in Tree Search. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17* (pp. 659–666).

Kool, W. W. M., & Welling, M. (2018). Attention solves your TSP, approximately. *arXiv:1803.08475 [cs, stat]*.

Kruber, M., Lübbecke, M. E., & Parmentier, A. (2017). Learning When to Use a Decomposition. In *Integration of AI and OR techniques in constraint programming*. In *Lecture Notes in Computer Science* (pp. 202–210). Springer, Cham. https://doi.org/10.1007/978-3-319-59776-8_16.

Larsen, E., Lachapelle, S., Bengio, Y., Frejinger, E., Lacoste-Julien, S., & Lodi, A. (2018). Predicting solution summaries to integer linear programs under imperfect information with machine learning. *arXiv:1807.11876 [cs, stat]*.

Larson, R. C., & Odoni, A. R. (1981). *Urban operations research*.

Li, K., & Malik, J. (2017). Learning to optimize neural nets. *arXiv:1703.00441 [cs, math, stat]*.

Liberto, G. D., Kadioglu, S., Leo, K., & Malitsky, Y. (2016). DASH: Dynamic approach for switching heuristics. *European journal of operational research, 248*(3), 943–953. https://doi.org/10.1016/j.ejor.2015.08.018. http://www.sciencedirect.com/science/article/pii/S0377221715007559.

Lindauer, M., & Hutter, F. (2018). Warmstarting of Model-Based Algorithm Configuration. In *thirty-Second AAAI conference on artificial intelligence*.

Lodi, A. (2009). MIP Computation. In M. Jünger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, … L. Wolsey (Eds.), *50 years of integer programming 1958–2008* (pp. 619–645). Springer-Verlag.

Lodi, A., & Zarpellon, G. (2017). On learning and branching: A survey. *TOP, 25*(2), 207–236. https://doi.org/10.1007/s11750-017-0451-6.

Lombardi, M., & Milano, M. (2018). Boosting combinatorial problem modeling with machine learning. In *Proceedings of the twenty-seventh international joint conference on artificial intelligence, IJCAI-18* (pp. 5472–5478). https://doi.org/10.24963/ijcai.2018/772. International Joint Conferences on Artificial Intelligence Organization

Mahmood, R., Babier, A., McNiven, A., Diamant, A., & Chan, T. C. Y. (2018). Automated treatment planning in radiation therapy using generative adversarial networks. In *Proceedings of machine learning for health care*. In *Proceedings of Machine Learning Research: 85*.

Malitsky, Y., Merschformann, M., O'Sullivan, B., & Tierney, K. (2016). Structure-Preserving Instance Generation. In P. Festa, M. Sellmann, & J. Vanschoren (Eds.), *Learning and Intelligent Optimization*. In *Lecture Notes in Computer Science* (pp. 123–140). Cham: Springer International Publishing.

Marcos Alvarez, A., Louveaux, Q., & Wehenkel, L. (2014). A supervised machine learning approach to variable branching in branch-and-bound. *Technical Report*. Université de Liège.

Marcos Alvarez, A., Louveaux, Q., & Wehenkel, L. (2017). A machine learning-based approximation of strong branching. *INFORMS journal on computing, 29*(1), 185–195. https://doi.org/10.1287/ijoc.2016.0723.

Marcos Alvarez, A., Wehenkel, L., & Louveaux, Q. (2016). Online Learning for Strong

Branching Approximation in Branch-and-Bound. *Technical Report*. Université de Liège.

Mascia, F., López-Ibáñez, M., Dubois-Lacoste, J., & Stützle, T. (2014). Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & Operations Research, 51*, 190–199. https://doi.org/10.1016/j.cor.2014.05.020. http://www.sciencedirect.com/science/article/pii/S0305054814001555.

McCormick, G. P. (1976). Computability of global solutions to factorable nonconvex programs: part i — convex underestimating problems. *Mathematical programming, 10*(1), 147–175. https://doi.org/10.1007/BF01580665.

Murphy, K. P. (2012). *Machine learning: A Probabilistic perspective*. MIT press.

Nagarajan, P., Warnell, G., & Stone, P. (2019). Deterministic implementations for reproducibility in deep reinforcement learning. In *AAAI 2019 workshop on reproducible AI*.

Nair, V., Dvijotham, D., Dunning, I., & Vinyals, O. (2018). Learning fast optimizers for contextual stochastic integer programs.. In *Conference on uncertainty in artifical intelligence* (pp. 591–600). http://auai.org/uai2018/proceedings/papers/217.pdf.

Nowak, A., Villar, S., Bandeira, A. S., & Bruna, J. (2017). A note on learning algorithms for quadratic assignment with graph neural networks. *arXiv:1706.07450 [cs, stat]*.

Özcan, E., Misir, M., Ochoa, G., & Burke, E. K. (2012). A reinforcement learning: Great-deluge hyper-heuristic for examination timetabling. *Modeling, Analysis, and Applications in Metaheuristic Computing: Advancements and Trends*, 34–55. https://doi.org/10.4018/978-1-4666-0270-0.ch003. https://www.igi-global.com/chapter/reinforcement-learning-great-deluge-hyper/63803.

Ravi, S., & Larochelle, H. (2017). Optimization as a model for few-shot learning. In *International conference on learning representations*.

Schmidhuber, J. (1992). Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural computation, 4*(1), 131–139. https://doi.org/10.1162/neco.1992.4.1.131.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., & Dill, D. L. (2018). Learning a SAT solver from single-bit supervision. *arXiv:1802.03685 [cs]*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., … Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature, 529*(7587), 484–489. https://doi.org/10.1038/nature16961.

Smith, K. A. (1999). Neural networks for combinatorial optimization: a review of more than a decade of research. *INFORMS journal on computing, 11*(1), 15–34. https://doi.org/10.1287/ijoc.11.1.15.

Smith-Miles, K., & Bowly, S. (2015). Generating new test instances by evolving in instance space. *Computers & Operations Research, 63*, 102–113. https://doi.org/10.1016/j.cor.2015.04.022. http://www.sciencedirect.com/science/article/pii/S0305054815001136.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: an introduction* (2nd). MIT press Cambridge. http://incompleteideas.net/book/the-book-2nd.html.

(1998). In S. Thrun, & L. Y. Pratt (Eds.), *Learning to learn*. Kluwer Academic.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2017). Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in neural information processing systems 30* (pp. 5998–6008). Curran Associates, Inc..

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lió, P., & Bengio, Y. (2018). Graph attention networks. In *International conference on learning representations*. https://openreview.net/forum?id=rJXMpikCZ.

Vinyals, O., Fortunato, M., & Jaitly, N. (2015). Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in neural information processing systems 28* (pp. 2692–2700). Curran Associates, Inc.

Wichrowska, O., Maheswaranathan, N., Hoffman, M. W., Colmenarejo, S. G., Denil, M., de Freitas, N., & Sohl-Dickstein, J. (2017). Learned Optimizers that Scale and Generalize. In D. Precup, & Y. W. Teh (Eds.), *Proceedings of the 34th International Conference on Machine Learning*. In *Proceedings of Machine Learning Research: 70* (pp. 3751–3760). International Convention Centre, Sydney, Australia: PMLR.

Wierstra, D., Förster, A., Peters, J., & Schmidhuber, J. (2010). Recurrent policy gradients. *Logic Journal of the IGPL, 18*(5), 620–634. https://doi.org/10.1093/jigpal/jzp049.

Wolsey, L. A. (1998). *Integer programming*. Wiley.

Parameterizing Branch-and-Bound Search Trees to Learn Branching Policies

Giulia Zarpellon,[1*] **Jason Jo,**[2, 3] **Andrea Lodi,**[1] **Yoshua Bengio**[2, 3]
[1]Polytechnique Montréal
[2]Mila
[3]Université de Montréal
giulia.zarpellon@polymtl.ca, jason.jo.research@gmail.com, andrea.lodi@polymtl.ca, yoshua.bengio@mila.quebec

Abstract

Branch and Bound (B&B) is the exact tree search method typically used to solve Mixed-Integer Linear Programming problems (MILPs). Learning branching policies for MILP has become an active research area, with most works proposing to imitate the strong branching rule and specialize it to distinct classes of problems. We aim instead at learning a policy that generalizes across heterogeneous MILPs: our main hypothesis is that parameterizing the state of the B&B search tree can aid this type of generalization. We propose a novel imitation learning framework, and introduce new input features and architectures to represent branching. Experiments on MILP benchmark instances clearly show the advantages of incorporating an explicit parameterization of the state of the search tree to modulate the branching decisions, in terms of both higher accuracy and smaller B&B trees. The resulting policies significantly outperform the current state-of-the-art method for "learning to branch" by effectively allowing generalization to generic unseen instances.

1 Introduction

Many problems arising from transportation, healthcare, energy and logistics can be formulated as Mixed-Integer Linear Programming (MILP) problems, i.e., optimization problems in which some decision variables represent discrete or indivisible choices. A MILP is written as

$$\min_x \{c^T x : Ax \geq b, x \geq 0,\ x_i \in \mathbb{Z}\ \forall i \in \mathcal{I}\}, \qquad (1)$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c, x \in \mathbb{R}^n$ and $\mathcal{I} \subseteq \{1, \ldots, n\}$ is the set of indices of variables that are required to be integral, while the other ones can be real-valued. Note that one can consider a MILP as defined by $(c, A, b, \mathcal{I})$; we do not assume any special combinatorial structure on the parameters c, A, b. While MILPs are in general $\mathcal{NP}$-hard, MILP solvers underwent dramatic improvements over the last decades (Lodi 2009; Achterberg and Wunderling 2013) and now achieve high-performance on a wide range of problems. The fundamental component of any modern MILP solver is Branch and Bound (B&B) (Land and Doig 1960), an *exact* tree search method. Following a divide-and-conquer approach, B&B partitions the search space by branching on variables' values and smartly uses bounds from problem relaxations to prune unpromising regions from the tree. The B&B algorithm actually relies on expertly-crafted *heuristic* rules for its two most fundamental decisions: *branching variable selection* (BVS) and *node selection*. In particular, BVS is a crucial factor for B&B's success (Achterberg and Wunderling 2013), and will be the main focus of the present article.

Understanding why B&B works has been called "one of the mysteries of computational complexity theory" (Lipton and Regan 2012), and there currently is no mathematical theory of branching; to the best of our knowledge, the only attempt in formalizing BVS is the recent work of Le Bodic and Nemhauser. One central reason why B&B is difficult to formalize resides in its inherent exponential nature: millions of BVS decisions could be needed to solve a MILP, and a single bad one could result in a doubled tree size and no improvement in the search. Such a complex and data-rich setting, paired with a lack of formal understanding, makes B&B an appealing ground for machine learning (ML) techniques, which have lately been thriving in discrete optimization (Bengio, Lodi, and Prouvost 2018). In particular, there has been substantial effort towards "learning to branch", i.e., in using ML methods to learn BVS policies (Lodi and Zarpellon 2017). Up to now, most works in this area focused on learning branching policies by supervision or imitation of *strong branching* (SB), a valid but expensive heuristic scheme (see Sections 2 and 5). The latest and state-of-the-art contribution to "learning to branch" (Gasse et al. 2019) frames BVS as a classification problem on SB expert decisions, and employs a graph-convolutional neural network (GCNN) to represent MILPs via their variable-constraint structure. The resulting branching policies improve on the solver by specializing SB to different classes of synthetic problems, and the attained generalization ability is to similar MILP instances (within the same class), possibly larger in formulation size.

The present work seeks a different type of generalization for a branching policy, namely a systematic generalization across *heterogeneous* MILPs, i.e., across problems not belonging to the same combinatorial class, without any restriction on the formulation's structure and size. To achieve this goal, *we parameterize BVS in terms of B&B search trees*. On the one hand, information about the state of the B&B

[*]Corresponding author. Code, data and supplementary materials can be found at: https://github.com/ds4dm/branch-search-trees

tree – abundant yet mostly unexploited by MILP solvers – was already shown to be useful to learn resolution patterns shared across general MILPs (Fischetti, Lodi, and Zarpellon 2019). On the other hand, the state of the search tree ought to have a central role in BVS – which ultimately decides how the tree is expanded and hence how the search itself proceeds. In practice, B&B continually interacts with other algorithmic components of the solver to effectively search the decision tree, and some algorithmic decisions may be triggered depending on which phase the optimization is in (Berthold, Hendel, and Koch 2017). In a highly integrated framework, a branching variable should thus be selected among the candidates based on its role in the search and its various components. Indeed, state-of-the-art heuristic branching schemes employ properties of the tree to make BVS decisions, and the B&B method equipped with such branching rules has proven to be successful across widely heterogeneous instances.

Motivated by these considerations, our main hypothesis is that MILPs share a higher order structure in the space of B&B search trees, and parameterized BVS policies should learn in this representational space. We setup a novel learning framework to investigate this idea. First of all, there is no natural input representation of this underlying space. Our first contribution is to craft input features of the variables that are candidates for branching: we aim at representing their roles in the search and its dynamic evolution. The dimensionality of such descriptions naturally changes with the number of candidates at every BVS step. The deep neural network (DNN) architecture that we propose learns a baseline branching policy (NoTree) from the candidate variables' representations and effectively deals with varying input dimensions. Taking this idea further, we suggest that an explicit representation of the *state of the search tree* should condition the branching criteria, in order for it to flexibly adapt to the tree evolution. We contribute such tree-state parameterization, and incorporate it to the baseline architecture to provide context over the candidate variables at each given branching step. In the resulting policy (TreeGate) the tree state acts as a control mechanism to drive a top-down modulation (specifically, feature gating) of the highly mutable space of candidate variables representations. By training in our hand-crafted input space, the signal-to-noise ratio of the high-level branching structure shared amongst general MILPs is effectively increased, enabling our TreeGate policy to rapidly infer these latent factors and dynamically compose them via top-down modulation. In this sense, *we learn branching from parameterizations of B&B search trees that are shared among general MILPs*. To the best of our knowledge, the present work is the first attempt in the "learning to branch" literature to represent B&B search trees for branching, and to establish such a broad generalization paradigm covering many classes of MILPs.

We perform imitation learning (IL) experiments on a curated dataset of heterogeneous instances from standard MILP benchmarks. We employ as expert rule `relpscost`, the default branching scheme of the optimization solver SCIP (Gleixner et al. 2018), to which our framework is integrated. Machine learning experimental results clearly show the advantage of the policy employing the tree state (TreeGate) over the baseline one (NoTree), the former achieving a 19% improvement in test accuracy. When plugged in the solver, both learned policies compare well with state-of-the-art branching rules. The evaluation of the trained policies in the solver also supports our idea that representing B&B search trees enables learning to branch across generic MILP instances: over test instances, the best TreeGate policy explores on average trees with 27% less nodes than the best NoTree one. In contrast, the GCNN framework of Gasse et al. that we use as benchmark does not appear to be able to attain such broad generalization goal: often the GCNN models fail to solve heterogeneous test instances, exploring search trees that are considerably bigger than those we obtain. The comparison thus remarks the advantage of our fundamentally new architectural paradigm – of representing candidates' role in the search and using a tree-context to modulate BVS – which without training in a class-specific manner nor focusing on constraints structure effectively allows learning across generic MILPs.

2 Background

Simply put, the B&B algorithm iteratively partitions the solution space of a MILP (1) into sub-problems, which are mapped to nodes of a binary decision tree. At each node, integrality requirements for variables in $\mathcal{I}$ are dropped, and a linear programming (LP) (continuous) relaxation of the problem is solved to provide a valid lower bound to the optimal value of (1). When the solution x^* of a node LP relaxation violates the integrality of some variables in $\mathcal{I}$, that node is further partitioned into two children by *branching on a fractional variable*. Formally, $\mathcal{C} = \{i \in \mathcal{I} : x_i^* \notin \mathbb{Z}\}$ defines the index set of *candidate variables* for branching at that node. The BVS problem consists in selecting a variable $j \in \mathcal{C}$ in order to *branch* on it, i.e., create child nodes according to the split

$$x_j \leq \lfloor x_j^* \rfloor \ \vee \ x_j \geq \lceil x_j^* \rceil. \tag{2}$$

Child nodes inherit a lower bound estimate from their parent, while (2) ensures x^* is removed from their solution spaces. After extending the tree, the algorithm moves on to select a new *open node*, i.e., a leaf yet to be explored (node selection): a new relaxation is solved, and new branchings happen. When x^* satisfies integrality requirements, then it is actually feasible for (1), and its value provides a valid upper bound to the optimal one. Maintaining global upper and lower bounds allows one to prune large portions of the search space. During the search, *final leaf nodes* are created in three possible ways: by integrality, when the relaxed solution is feasible for (1); by infeasibility of the subproblem; by bounds, when the comparison of the node's lower bound to the global upper one proves that its sub-tree is not worth exploring. An optimality certificate is reached when the global bounds converge. See (Wolsey 1998; Lodi 2009) for details on B&B and its combination with other components of a MILP solver.

Branching rules Usually, candidates are evaluated with respect to some scoring function, and j is chosen for branch-

ing as the (or a) score-maximizing variable (Achterberg, Koch, and Martin 2005). The most used criterion in BVS measures variables depending on the improvement of the lower bound in their (prospective) child nodes. The *strong branching* (SB) rule (Applegate et al. 1995) explicitly computes bound gains for $\mathcal{C}$. The procedure is expensive, but experimentally realizes trees with the least number of nodes. Instead, *pseudo-cost* (PC) (Benichou et al. 1971) maintains a history of variables' branchings, averaging past improvements to get a proxy for the expected gain. Fast in evaluation, PC can behave badly due to uninitialization, so combinations of SB with PC have been developed. In *reliability branching*, SB is performed until PC scores for a variable are deemed reliable proxies of bound improvements. In *hybrid branching* (Achterberg and Berthold 2009), PC scores are combined with other ones measuring the variables' role on inference and conflict clauses. Many other scoring criteria have been proposed, and some of them are surveyed in Lodi and Zarpellon from a ML perspective.

State-of-the-art branching rules can in fact be interpreted as mechanisms to score variables based on their effectiveness in different search components. While hybrid branching explicitly combines five scores reflecting variables' behaviors in different search tasks, the evaluation performed by SB and PC can also be seen as a measure of how effective a variable is – in the single task of improving the bound from one parent node to its children. Besides, one can assume that the importance of different search functionalities should change dynamically during the tree exploration. [1] In this sense, our approach aims at learning a branching rule that takes into account variables' roles in the search and the tree evolution itself to perform a more flexible BVS, adapted to the search stages.

3 Parameterizing B&B Search Trees

The central idea of our framework is to learn BVS by means of parameterizing the underlying space of B&B search trees. We believe this space can represent the complexity and the dynamism of branching in a way that is shared across heterogeneous problems. However, there are no natural parameterization of BVS or B&B search trees. To this end, our contribution is two-fold: 1) we propose hand-crafted input features to describe candidate variables in terms of their roles in the B&B process, and explicitly encode a "tree state" to provide a richer context to variable selection; 2) we design novel DNN architectures to integrate these inputs and learn BVS policies.

3.1 Hand-crafted Input Features

At each branching step t, we represent the set of variables that are candidates for branching by an input matrix $C_t \in \mathbb{R}^{25\times|\mathcal{C}_t|}$. To capture the multiple roles of a variable throughout the search, we describe each candidate $x_j, j \in \mathcal{C}_t$ in terms of its bounds and solution value in the current subproblem. We also feature statistics of a variable's participation in various search components and in past branchings. In particular, the scores that are used in the SCIP default hybrid-branching formula are part of C_t.

Additionally, we create a separate parameterization $Tree_t \in \mathbb{R}^{61}$ to describe the state of the search tree. We record information of the current node in terms of depth and bound quality. We also consider the growth rate and the composition of the tree, the evolution of global bounds, aggregated variables' scores, statistics on feasible solutions and on bound estimates and depths of open nodes.

All features are designed to capture the dynamics of the B&B process linked to BVS decisions, and are efficiently gathered through a customized version of PySCIPOpt (Maher et al. 2016). Note that $\{C_t, Tree_t\}$ are defined in a way that is not explicitly dependent on the parameters of each instance $(c, A, b, \mathcal{I})$. Even though C_t naturally changes its dimensionality at each BVS step t depending on the highly variable $\mathcal{C}_t$, the fixed lengths of the vectors enable training among branching sets of different sizes (see 3.2). The representations evolve with the search: t-SNE plots (van der Maaten and Hinton 2008) in Figures 1(a) and 1(b) synthesize the evolution of $Tree_t$ throughout the B&B search, for two different MILP instances. The pictures clearly show the high heterogeneity of the branching data across different search stages. A detailed description of the hand-crafted input features is reported in the supplementary material (SM).

3.2 Architectures to Model Branching

We use parameterizations C_t as inputs for a baseline DNN architecture (NoTree). Referring to Figure 2, the 25-feature input of a candidate variable is first embedded into a representation with hidden size h; subsequently, multiple layers reduce the dimensionality from h to an infimum INF by halving it at each step. The vector of length INF is then compressed by global average pooling into a single scalar. The $|\mathcal{C}_t|$ dimension of C_t is conceived (and implemented) as a "batch dimension": this makes it possible to handle branching sets of varying sizes, still allowing the parameters of the nets to be shared across problems. Ultimately, a $softmax$ layer yields a probability distribution over the candidate set $\mathcal{C}_t$, according to which a variable is selected for branching.

We incorporate the tree-state input to the baseline architecture to provide a search-based context over the mutable branching sets. Practically, $Tree_t$ is embedded in a series of subsequent layers with hidden size h. The output of a final sigmoid activation is $g \in [0,1]^H$, where $H = h + h/2 + \cdots + INF$ denotes the total number of units of the NoTree layers. Separate chunks of g are used to modulate by feature gating the representations of NoTree: $[g_1, \ldots, g_h]$ controls features at the first embedding, $[g_{h+1}, \ldots, g_{h+h/2}]$ acts at the second layer, ..., and so on, until exhausting $[g_{H-INF}, \ldots, g_H]$ with the last layer prior the average pooling. In other words, g is used as a control mechanism on variables parameterization, gating their features via a learned tree-based signal. The resulting network (TreeGate) models the high-level idea that a branching scheme should *adapt to the tree evolution*, with variables' selection criteria dynamically changing throughout the tree search.

[1] Indeed, a "dynamic factor" adjusts weights in the default branching scheme of SCIP (relpscost).

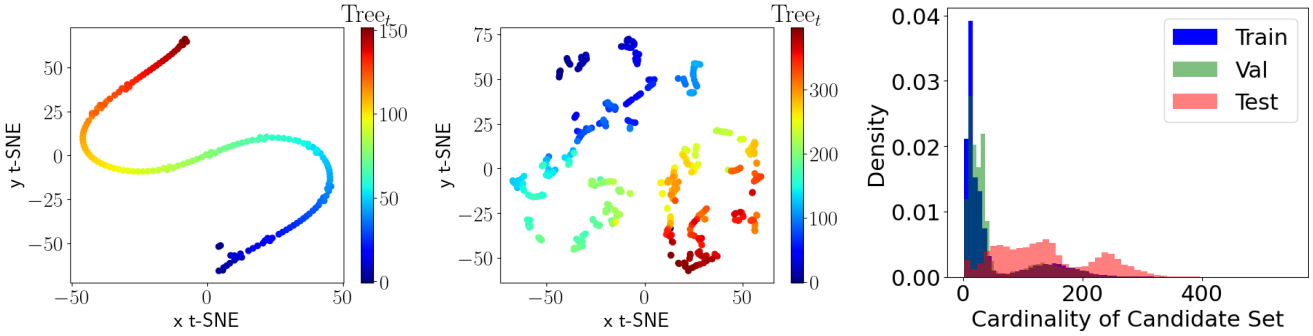


Figure 1: Evolution of $Tree_t$ throughout B&B as synthesized by t-SNE plots (perplexity=5), for instances (a) eil33-2 and (b) seymour1. (c) Histogram of $|\mathcal{C}_t|$ in train, validation and test data.

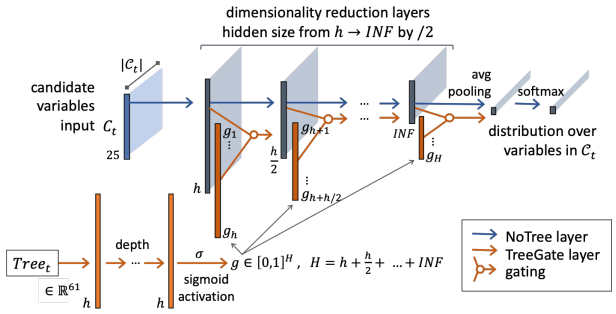


Figure 2: Candidate variables input C_t is processed by NoTree layers (in blue) to select a variable for branching. For the TreeGate model, the $Tree_t$ input is first embedded and then utilized in gating layers (in orange) on the candidates' representations.

3.3 Imitation Learning

We train our BVS policies via imitation learning, specifically behavioral cloning (Pomerleau 1991). Our expert branching scheme is SCIP default, `relpscost`, i.e., a reliability version of hybrid branching in which SB and PC scores are complemented with other ones reflecting the candidates' role in the search; `relpscost` is a more realistic expert (nobody uses SB in practice), and the most suited in our context, given the emphasis we put on the search tree. One of the main challenges of learning to imitate a complex BVS expert policy is that it is only possible to partially observe the true state of the solver. In our learning framework, we approximate the solver state with our parameterized B&B search tree state $\mathrm{x}_t = \{C_t, Tree_t\}$ at branching step t. For each MILP instance, we roll-out SCIP to gather a collection of pairs $\mathcal{D} = (\mathrm{x}_t, \mathrm{y}_t)_{t=1}^N$ where $\mathrm{y}_t \in \mathcal{C}_t$ is the branching decision made by `relpscost` at branching step t. Our policies π_θ are trained to minimize the cross-entropy categorical loss function:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{(\mathrm{x},\mathrm{y}) \in \mathcal{D}} \log \pi_\theta(\mathrm{y} \mid \mathrm{x}). \qquad (3)$$

3.4 Systematic Generalization

Our aim is to explore systematic generalization in learning to branch. We measure systematic generalization by evaluating how well trained policies perform on never-seen heterogeneous MILP instances. To begin, we remark that `relpscost` is a sophisticated ensemble of expertly-crafted heuristic rules that are dependent on high-level branching factors. Due to the systematic generalization failures currently plaguing even state-of-the-art DNNs (Lake and Baroni 2017; Johnson et al. 2017; Goodfellow, Shlens, and Szegedy 2015), we do not believe that current learning algorithms are capable of inferring these high-level branching factors from the raw MILP formulation data alone. Instead, by opting to train BVS policies in our hand-crafted input feature space: (1) the signal-to-noise ratio of the underlying branching factors is effectively increased, and (2) the likelihood of our models overfitting to superficial regularities is vastly decreased as this tree-oriented feature space is abstracted away from instance or class specific peculiarities of the MILP formulation. However, in order to achieve systematic generalization, inference of high-level latent factors must be paired with a corresponding composition mechanism. Here, the top-down neuro-modulatory prior of our TreeGate model represents a powerful mechanism for dynamically composing the tree state and the candidate variable states together. In summary, we hypothesize that BVS policies trained in our framework are able to better infer and compose latent higher order branching factors, which in turn enables flexible generalization across heterogeneous MILP instances.

4 Experiments

Our experiments are designed to carefully measure the inductive bias of BVS learning frameworks. We echo the sentiment of Melis, Kočiský, and Blunsom that merely scaling up a dataset is insufficient to explore the issue of systematic generalization in deep learning models, and we instead choose to curate a controlled dataset with the following properties:

- *Train/test split consisting of heterogeneous MILP instances.* This rules out the possibility of BVS policies

learning superficial class-specific branching rules and instead forces them to infer higher order branching factors shared across MILPs.

- *A restricted set of training instances.* While limiting the sample complexity of the training set poses a significant learning challenge, the learning framework with the best inductive bias will be the one that best learns to infer and compose higher order branching factors from such a difficult training set.

While we train our BVS policies to imitate the SCIP default policy `relpscost`, our objective is not to outperform `relpscost`. Indeed, expert BVS policies like `relpscost` are tuned over *thousands* of solvers' proprietary instances: comprehensively improving on them is a very hard task – impossible to guarantee in our purely-research experimental setting – and should thus not be the only yardstick to determine the validity (and practicality) of a learned policy. Instead, we focus on evaluating how efficiently BVS learning frameworks can learn to infer and compose higher order branching factors by measuring generalization from a controlled training set to heterogeneous MILP test instances.

MILP dataset and solver setting In general, randomly-generated *generic* MILPs are too easy to be of interest; besides, public MILP libraries only contain few hundreds of instances, not all viable for our setting, and a careful dataset curation is thus needed. Comparisons of branching policies become clearer when the explored trees are manageable in size and the problems can be consistently solved to optimality. Thus we select 27 heterogeneous problems from real-world MILP benchmark libraries (Bixby et al. 1998; Koch et al. 2011; Gleixner et al. 2019; Mittelmann 2020), focusing on instances whose tree exploration is on average relatively contained (in the tens/hundreds of thousands nodes, max.) and whose optimal value is known. We partition our selection into 19 train and 8 test problems, which are listed in Table 1(a) (see SM for more details).

We use SCIP 6.0.1. Modifying the solver configuration is common practice in BVS literature (Linderoth and Savelsbergh 1999), especially in a proof-of-concept setting in which our work is positioned. To reduce the effects of the other solver's components on BVS, we work with a configuration specifically designed to fairly compare the performance of branching rules (Gamrath and Schubert 2018). In particular, we disable all primal heuristics and for each problem we provide the known optimal solution value as cutoff. We also enforce a time-limit of 1h. Further details on the solver parameters and hardware settings are reported in the SM.

Data collection and split We collect IL training data from SCIP roll-outs, gathering inputs $\mathrm{x}_t = \{C_t, \mathit{Tree}_t\}$ and corresponding `relpscost` branching decisions (labels) $\mathrm{y}_t \in \mathcal{C}_t$. Given that each branching decision gives rise to a single data-point $(\mathrm{x}_t, \mathrm{y}_t)$, and that the search trees of the selected MILP instances are not extremely big, one needs to augment the data. We proceed in two ways.

TRAIN: air04, air05, dcmulti, eil33-2, istanbul-no-cutoff, l152lav, lseu, misc03, neos20, neos21, neos-476283, neos648910, pp08aCUTS, rmatr100-p10, rmatr100-p5, sp150x300d, stein27, swath1, vpm2
TEST: map18, mine-166-5, neos11, neos18, ns1830653, nu25-pr12, rail507, seymour1

(a)

	Total	(s, k) **pairs**
Train	85,533	$\{0,1,2,3\} \times \{0,1,5,10,15\}$
Valid.	14,413	$\{4\} \times \{0,1,5,10,15\}$
Test	28,307	$\{0,1,2,3,4\} \times \{0\}$

(b)

Table 1: (a) List of MILP instances in train and test sets. (b) For train, validation and test set splits we report the total number of data-points and the seed-k pairs (s, k) from which they are obtained.

Policy	h / d / LR	Test acc (@5)	Val acc (@5)
NT	128 / – / 0.001	64.02 (88.51)	77.69 (95.88)
TG	64 / 5 / 0.01	83.70 (95.83)	84.33 (96.60)
GCNN	– / – / –	15.28 (44.16)	19.28 (38.44)

Table 2: Selected NoTree (NT) and TreeGate (TG) models with corresponding hyper-parameters, and test and validation accuracy. For GCNN we report *average* scores across 5 seeds; validation means use the best scores observed during training.

i. We exploit MILPs *performance variability* (Lodi and Tramontani 2013), and obtain perturbed searches of the same instance by setting solver's random seeds $s \in \{0, ..., 4\}$ to control variables' permutations.

ii. We diversify B&B explorations by running the branching scheme `random` for the first k nodes, before switching to SCIP default rule and starting data collection. The motivation behind this type of augmentation is to gather input states that are unlikely to be observed by an expert rule (He, Daume III, and Eisner 2014). We select $k \in \{0, 1, 5, 10, 15\}$, where $k = 0$ corresponds to a run without random branching. We apply this type of augmentation to *train instances only*.

One can quantify MILP variability by computing the coefficient of variation of the performance measurements (Koch et al. 2011); we report such scores and measure the effect of k initial random branchings in the SM. Overall, both (i) and (ii) appear effective to diversify our dataset. The final composition of train, validation and test sets is summarized in Table 1(b). Train and validation data come from the same instances; the test set contains samples from separate MILPs, using only type (i) augmentations.

An important measure to analyze the dataset is given by the size of the candidate sets (i.e., the varying dimensionality of the C_t inputs) contained in each split. Figure 1(c)

shows histograms for $|\mathcal{C}_t|$ in each subset. While in train and validation the candidate set sizes are mostly concentrated in the $[0, 50]$ range, the test set has a very different distribution of $|\mathcal{C}_t|$, and in particular one with a longer tail (over 300). In this sense, the test instances present never-seen branching data gathered from heterogeneous MILPs, and we test the generalization of our policies to *entirely unknown and larger branching sets*.

IL optimization and GCNN benchmark We train both IL policies using ADAM (Kingma and Ba 2015) with default $\beta_1 = 0.9, \beta_2 = 0.999$, and weight decay 1×10^{-5}. Our hyper-parameter search spans: learning rate $LR \in \{0.01, 0.001, 0.0001\}$, hidden size $h \in \{32, 64, 128, 256\}$, and depth $d \in \{2, 3, 5\}$. The factor by which units of NoTree are reduced is 2, and we fix $INF = 8$. We use PyTorch (Paszke et al. 2019) to train the models for 40 epochs, reducing LR by a factor of 10 at epochs 20 and 30. To benchmark our results, we also train the GCNN framework of Gasse et al. on our MILP dataset. Data collection and experiments are carried out as in Gasse et al., with full SB as expert, but we fix the solver setting as discussed above.

4.1 Results

In our context, standard IL metrics are informative yet incomplete measures of performance for a learned BVS model, and one also cares about assessing the policies' behaviors when plugged in the solver environment. This is why in order to determine the best NoTree and TreeGate policies we take into account both types of evaluations. We first select few policies based on their test accuracy score; next, we specify them as custom branching rules in SCIP and perform full roll-outs on the entire MILP dataset, over five random seeds (i.e., 135 evaluations each). To summarize the policies' performance in the solver, we compute the shifted geometric mean (with a shift of 100) of the total number of nodes, over the 135 B&B executions (ALL), and restricted to TRAIN and TEST instances.

Both types of metrics are extensively reported in the SM, together with the policies' hyper-parameters. Incorporating an explicit parameterization of the state of the search tree to modulate BVS clearly aids generalization: the advantage of TreeGate over NoTree is evident in all metrics, and across multiple trained policies. What we observe is that best test accuracy does not necessarily translate into best solver performance. We select as best policies those yielding the best nodes average over the entire dataset (Table 2). In the case of TreeGate, the best model corresponds to that realizing the best top-1 test accuracy (83.70%), and brings a 19% (resp. 7%) improvement over the NoTree policy, in top-1 (resp. top-5) test accuracy. The GCNN models (trained, tested and evaluated over 5 seeds, as in Gasse et al.) struggle to fit data from heterogeneous instances: their average top-1 (resp. top-5) test accuracy is only 15.28% (resp. 44.16%), and across ALL instances they explore around three times the number of nodes needed by our policies. Note that GCNN is a memory intensive model, and we had to drastically reduce the batch size parameter to avoid memory issues when using our instances. Learning curves and further details on training dynamics and test results can be found in the SM.

In solver evaluations, NoTree and TreeGate are also compared to SCIP default branching scheme `relpscost`, PC branching `pscost` and a `random` one. For `relpscost` we also compute the *fair* number of nodes (Gamrath and Schubert 2018), which accounts for those nodes that are processed as side-effects of SB-like explorations, specifically looking at domain reduction and cutoffs counts. In other words, the fair number distinguishes tree-size reductions due to better branching from those obtained by SB side-effects. For rules that do not involve any SB, the fair number and the usual nodes' count coincide. The selected solver parametric setting (the same used for data collection and GCNN benchmark) allows a meaningful computation of the fair number of nodes, and a honest comparison of branching schemes.

Both NoTree and TreeGate policies are able to solve all instances within the 1h time-limit, like `relpscost`. In contrast, GCNN hits the limit on 7 instances (24 times in total), while `random` does so on 4 instances (17 times in total) and `pscost` on one instance only (neos18), a single time. Table 3 reports the nodes' means for every test instance over five runs (see SM for complete instance-specific results), as well as measures aggregated over train and test sets, and the entire dataset. In aggregation, TreeGate is always better than NoTree, the former exploring on average trees with 14.9% less nodes. This gap becomes more pronounced when measured over test instances only (27%), indicating the advantage of TreeGate over NoTree when exploring unseen data. Results are less clear-cut from an instance-wise perspective, with neither policy emerging as an absolute winner, though the reductions in tree sizes achieved by TreeGate are overall more pronounced. While the multiple time-limits of GCNN hinder a proper comparison in terms of explored nodes, results clearly indicate that the difficulties of GCNN exacerbate over never-seen, heterogeneous test instances.

Our policies also compare well to other branching rules: both NoTree and TreeGate are substantially better than `random` across all instances, and always better than `pscost` in aggregated measures. Only on one training instance both policies are much worse than `pscost` (neos-476283); in the test set, GCNN appears competitive with our models only on neos11. As expected, `relpscost` still realizes the smallest trees, but on 11 (out of 27) instances at least one among NoTree and TreeGate explores less nodes than the `relpscost` fair number. In general, our policies realize tree sizes comparable to the SCIP ones, when SB side effects are taken into account.

5 Related Work

Among the first attempts in "learning to branch", Alvarez, Louveaux, and Wehenkel perform regression to learn proxies of SB scores. Instead, Khalil et al. propose to learn the ranking associated with such scores, and train instance-specific models (that are not end-to-end policies) via SVM^{rank}. Also Hansknecht, Joormann, and Stiller treat BVS as a ranking problem, and specialize their models to the combinatorial class of time-dependent traveling sales-

Instance	NoTree	TreeGate	GCNN	random	pscost	relpscost (fair)
ALL	1241.79	1056.79	*3660.32	*6580.79	*1471.61	286.15 (719.20)
TRAIN	834.40	759.94	*1391.41	*2516.04	884.37	182.27 (558.34)
TEST	3068.96	2239.47	*33713.63	*61828.29	*4674.34	712.77 (1276.76)
map18	457.89	575.92	*3907.64	11655.33	1025.74	270.25 (441.18)
mine-166-5	3438.44	4996.48	*233142.25	*389437.62	4190.41	175.10 (600.22)
neos11	3326.32	3223.46	1642.07	29949.69	4728.49	2618.27 (5468.05)
neos18	15611.63	10373.80	40794.74	228715.62	*133437.40	2439.29 (5774.36)
ns1830653	6422.37	5812.03	*22931.45	288489.30	12307.90	3489.07 (4311.84)
nu25-pr12	357.00	86.80	*45982.34	1658.41	342.47	21.39 (105.61)
rail507	9623.05	3779.05	*75663.48	*80575.84	4259.98	543.39 (859.37)
seymour1	3202.20	1646.82	*319046.04	*167725.65	3521.47	866.32 (1096.67)

Table 3: Total number of nodes explored by learned and SCIP policies for test instances and aggregated over sets, in shifted geometric means over 5 runs on seeds $\{0, \dots, 4\}$. We mark with * the cases in which time-limits were hit. For relpscost, we also compute the *fair* number of nodes.

man problems. More recently, the work of Balcan et al. learns mixtures of existing branching schemes for different classes of synthetic problems, focusing on sample complexity guarantees. In Di Liberto et al., a portfolio approach to BVS is explored. Similarly to us, Gasse et al. frames BVS as classification of SB-expert branching decisions and employs a GCNN model to learn branching. Proposed features in Gasse et al. (as in Alvarez, Louveaux, and Wehenkel; Khalil et al.) focus on static, parameters-dependent properties of MILPs and node LP relaxations, whereas our representations aim at capturing the temporality and dynamism of BVS. Although their resulting policies are specializations of SB that appear to effectively capture structural characteristics of some classes of combinatorial optimization problems, and are able to generalize to larger formulations from the same distribution, we showed how such policies fail to attain a broader generalization paradigm.

Still concerning the B&B framework, He, Daume III, and Eisner employ IL to learn a heuristic class-specific node selection policy; Song et al. propose instead a retrospective approach on IL. A reinforcement learning (RL) approach for node selection can be found in Sabharwal, Samulowitz, and Reddy, where a Multi-Armed Bandit is used to model the tree search.

Feature gating has a long and successful history in machine learning (see Makkuva et al.), ranging from LSTMs (Hochreiter and Schmidhuber 1997) to GRUs (Chung et al. 2014). The idea of using a tree state to drive a feature gating of the branching variables is an example of top-down modulation, which has been shown to perform well in other deep learning applications (Shrivastava et al. 2016; Lin et al. 2016; Son and Mishra 2018). With respect to learning across non-static action spaces, the most similar to our work is Chandak et al., in the continual learning setting. Unlike the traditional Markov Decision Process formulation of RL, the input to our policies is not a generic state but rather includes a parameterized hand-crafted representation of the available actions, thus continual learning is not a relevant concern for our framework. Other works from the RL setting learn representations of *static* action spaces (Dulac-Arnold et al. 2015; Chandak et al. 2019a), while in contrast the action space of BVS changes dynamically with $|\mathcal{C}_t|$.

6 Conclusions and Future Directions

Branching variable selection is a crucial factor in B&B success, and we setup a novel imitation learning framework to address it. We sought to learn branching policies that generalize across heterogeneous MILPs, regardless of the instances' structure and formulation size. In doing so, we undertook a step towards a broader type of generalization. The novelty of our approach is relevant for both the ML and the MILP worlds. On the one hand, we developed parameterizations of the candidate variables and of the search trees, and designed a DNN architecture that handles candidate sets of varying size. On the other hand, the data encoded in our $Tree_t$ parameterization is not currently exploited by state-of-the-art MILP solvers, but we showed that this type of information could indeed help in adapting the branching criteria to different search dynamics. Our results on MILP benchmark instances clearly demonstrated the advantage of incorporating a search-tree context to modulate BVS and aid generalization to heterogeneous problems, in terms of both better test accuracy and smaller explored B&B trees. The comparison with the GCNN setup of Gasse et al. reinforced our conclusions: experiments showcased the inability of the GCNN paradigm alone to generalize to new instances for which no analogs were available during training. One crucial step towards improving over state-of-the-art solvers is precisely that of being able to generalize across heterogeneous problems, and our work is the first paper in the literature attaining this target.

There surely are additional improvements to be gained by continuing to experiment with IL methods for branching, and also by exploring innovative RL settings. Indeed, the idea and the benefits of using an explicit parameterization of B&B search trees – which we demonstrated in the IL setup – could be expanded even more in the RL one, for both state representations and the design of branching rewards.

Acknowledgements

We would like to thank Ambros Gleixner, Gerald Gamrath, Laurent Charlin, Didier Chételat, Maxime Gasse, Antoine Prouvost, Leo Henri and Sébastien Lachapelle for helpful discussions on the branching framework. We also thank Compute Canada for compute resources. This work was supported by CIFAR and IVADO. We also thank the anonymous reviewers for their helpful feedback.

References

Achterberg, T.; and Berthold, T. 2009. *Hybrid Branching*, 309–311. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-01929-6_23.

Achterberg, T.; Koch, T.; and Martin, A. 2005. Branching rules revisited. *Oper Res Lett* 33(1): 42–54. doi:10.1016/j.orl.2004.04.002.

Achterberg, T.; and Wunderling, R. 2013. *Mixed Integer Programming: Analyzing 12 Years of Progress*, 449–481. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-38189-8_18.

Alvarez, M. A.; Louveaux, Q.; and Wehenkel, L. 2017. A Machine Learning-Based Approximation of Strong Branching. *INFORMS Journal on Computing* 29(1): 185–195. doi:10.1287/ijoc.2016.0723.

Applegate, D.; Bixby, R.; Chvatal, V.; and Cook, B. 1995. Finding Cuts in the TSP (A Preliminary Report). Technical report.

Balcan, M.-F.; Dick, T.; Sandholm, T.; and Vitercik, E. 2018. Learning to Branch. In *International Conference on Machine Learning*, 344–353.

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

Benichou, M.; Gauthier, J.; Girodet, P.; and Hentges, G. 1971. Experiments in mixed-integer programming. *Math Program* 1: 76–94.

Berthold, T.; Hendel, G.; and Koch, T. 2017. From feasibility to improvement to proof: three phases of solving mixed-integer programs. *Optimization Methods and Software* 33(3): 499 – 517. doi:10.1080/10556788.2017.1392519.

Bixby, R. E.; Ceria, S.; McZeal, C. M.; and Savelsbergh, M. W. 1998. An updated mixed-integer programming library: MIPLIB 3. Technical Report TR98-03. URL http://miplib2010.zib.de/miplib3/miplib.html. Accessed: 2019.

Chandak, Y.; Theocharous, G.; Kostas, J.; Jordan, S.; and Thomas, P. 2019a. Learning Action Representations for Reinforcement Learning. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, 941–950. PMLR.

Chandak, Y.; Theocharous, G.; Nota, C.; and Thomas, P. S. 2019b. Lifelong Learning with a Changing Action Set. *CoRR* abs/1906.01770.

Chung, J.; Gulcehre, C.; Cho, K.; and Bengio, Y. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv:1412.3555* .

Di Liberto, G.; Kadioglu, S.; Leo, K.; and Malitsky, Y. 2016. DASH: Dynamic Approach for Switching Heuristics. *Eur J Oper Res* 248(3): 943–953. doi:10.1016/j.ejor.2015.08.018.

Dulac-Arnold, G.; Evans, R.; Sunehag, P.; and Coppin, B. 2015. Reinforcement Learning in Large Discrete Action Spaces. *CoRR* abs/1512.07679.

Fischetti, M.; Lodi, A.; and Zarpellon, G. 2019. Learning MILP Resolution Outcomes Before Reaching Time-Limit. In Rousseau, L.-M.; and Stergiou, K., eds., *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 275–291. Cham: Springer International Publishing.

Gamrath, G.; and Schubert, C. 2018. Measuring the Impact of Branching Rules for Mixed-Integer Programming. In Kliewer, N.; Ehmke, J. F.; and Borndörfer, R., eds., *Operations Research Proceedings 2017*, 165–170. Cham: Springer International Publishing.

Gasse, M.; Chetelat, D.; Ferroni, N.; Charlin, L.; and Lodi, A. 2019. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d'Alché Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 32*, 15554–15566. Curran Associates, Inc.

Gleixner, A.; Bastubbe, M.; Eifler, L.; Gally, T.; Gamrath, G.; Gottwald, R. L.; Hendel, G.; Hojny, C.; Koch, T.; Lübbecke, M.; Maher, S. J.; Miltenberger, M.; Müller, B.; Pfetsch, M.; Puchert, C.; Rehfeldt, D.; Schlösser, F.; Schubert, C.; Serrano, F.; Shinano, Y.; Viernickel, J. M.; Walter, M.; Wegscheider, F.; Witt, J. T.; and Witzig, J. 2018. The SCIP Optimization Suite 6.0. Technical Report 18-26, ZIB, Takustr. 7, 14195 Berlin.

Gleixner, A.; Hendel, G.; Gamrath, G.; Achterberg, T.; Bastubbe, M.; Berthold, T.; Christophel, P. M.; Jarck, K.; Koch, T.; Linderoth, J.; Lübbecke, M.; Mittelmann, H. D.; Ozyurt, D.; Ralphs, T. K.; Salvagnin, D.; and Shinano, Y. 2019. *MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library*. URL http://www.optimization-online.org/DB_HTML/2019/07/7285.html. Accessed: 2019.

Goodfellow, I.; Shlens, J.; and Szegedy, C. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*.

Hansknecht, C.; Joormann, I.; and Stiller, S. 2018. Cuts, Primal Heuristics, and Learning to Branch for the Time-Dependent Traveling Salesman Problem. *arXiv:1805.01415* .

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.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8): 1735–1780.

Johnson, J.; Hariharan, B.; van der Maaten, L.; Fei-Fei, L.; Zitnick, C. L.; and Girshick, R. 2017. CLEVR: A Diagnostic Dataset for Compositional Language and Elementary Visual Reasoning. In *CVPR*.

Khalil, E. B.; Bodic, P. L.; Song, L.; Nemhauser, G.; and Dilkina, B. 2016. Learning to Branch in Mixed Integer Programming. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, 724–731. AAAI Press.

Kingma, D. P.; and Ba, J. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*.

Koch, T.; Achterberg, T.; Andersen, E.; Bastert, O.; Berthold, T.; Bixby, R. E.; Danna, E.; Gamrath, G.; Gleixner, A. M.; Heinz, S.; Lodi, A.; Mittelmann, H.; Ralphs, T.; Salvagnin, D.; Steffy, D. E.;

and Wolter, K. 2011. MIPLIB 2010. *Mathematical Programming Computation* 3(2): 103. doi:10.1007/s12532-011-0025-9.

Lake, B. M.; and Baroni, M. 2017. Still not systematic after all these years: On the compositional skills of sequence-to-sequence recurrent networks. *CoRR* abs/1711.00350.

Land, A. H.; and Doig, A. G. 1960. An Automatic Method of Solving Discrete Programming Problems. *Econometrica* 28(3): 497–520.

Le Bodic, P.; and Nemhauser, G. 2017. An abstract model for branching and its application to mixed integer programming. *Math Program* 1–37. doi:10.1007/s10107-016-1101-8.

Lin, T.; Dollár, P.; Girshick, R. B.; He, K.; Hariharan, B.; and Belongie, S. J. 2016. Feature Pyramid Networks for Object Detection. *CoRR* abs/1612.03144.

Linderoth, J. T.; and Savelsbergh, M. W. P. 1999. A Computational Study of Search Strategies for Mixed Integer Programming. *INFORMS Journal on Computing* 11(2): 173–187. doi: 10.1287/ijoc.11.2.173.

Lipton, R. J.; and Regan, R. W. 2012. Branch And Bound—Why Does It Work? https://rjlipton.wordpress.com/2012/12/19/branch-and-bound-why-does-it-work/. Accessed: 2019.

Lodi, A. 2009. Mixed integer programming computation. In Jünger, M.; Liebling, T.; Naddef, D.; Nemhauser, G.; Pulleyblank, W.; Reinelt, G.; Rinaldi, G.; and Wolsey, L., eds., *50 Years of Integer Programming 1958-2008*, 619–645. Springer Berlin Heidelberg.

Lodi, A.; and Tramontani, A. 2013. *Performance Variability in Mixed-Integer Programming*, chapter 1, 1–12. INFORMS. doi: 10.1287/educ.2013.0112.

Lodi, A.; and Zarpellon, G. 2017. On learning and branching: a survey. *TOP* 25(2): 207–236. doi:10.1007/s11750-017-0451-6.

Maher, S.; Miltenberger, M.; Pedroso, J. P.; Rehfeldt, D.; Schwarz, R.; and Serrano, F. 2016. PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite. In Greuel, G.-M.; Koch, T.; Paule, P.; and Sommese, A., eds., *Mathematical Software – ICMS 2016*, 301–307. Cham: Springer International Publishing. ISBN 978-3-319-42432-3.

Makkuva, A. V.; Oh, S.; Kannan, S.; and Viswanath, P. 2019. Learning in Gated Neural Networks. *CoRR* abs/1906.02777.

Melis, G.; Kočiský, T.; and Blunsom, P. 2020. Mogrifier LSTM. In *International Conference on Learning Representations*.

Mittelmann, H. D. 2020. MILPlib. http://plato.asu.edu/ftp/milp/. Accessed 2019.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; and Chintala, S. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d' Alché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 32*, 8024–8035. Curran Associates, Inc.

Pomerleau, D. A. 1991. Efficient Training of Artificial Neural Networks for Autonomous Navigation. *Neural Computation* 3(1): 88–97.

relpscost. 2019. Code for the `relpscost` branching rule in SCIP. https://scip.zib.de/doc-6.0.0/html/branch__relpscost_8c_source.php#l00524. Accessed: 2019.

Sabharwal, A.; Samulowitz, H.; and Reddy, C. 2012. Guiding Combinatorial Optimization with UCT. In Beldiceanu, N.; Jussien, N.; and Pinson, É., eds., *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28 – June 1, 2012. Proceedings*, Lecture Notes in Computer Science, 356–361. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-29828-8_23.

Shrivastava, A.; Sukthankar, R.; Malik, J.; and Gupta, A. 2016. Beyond Skip Connections: Top-Down Modulation for Object Detection. *CoRR* abs/1612.06851.

Son, J.; and Mishra, A. K. 2018. ExGate: Externally Controlled Gating for Feature-based Attention in Artificial Neural Networks. *CoRR* abs/1811.03403.

Song, J.; Lanka, R.; Zhao, A.; Bhatnagar, A.; Yue, Y.; and Ono, M. 2018. Learning to Search via Retrospective Imitation. *arXiv preprint 1804.00846* .

van der Maaten, L.; and Hinton, G. 2008. Visualizing High-Dimensional Data Using t-SNE. *Journal of Machine Learning Research* 9(nov): 2579–2605. Pagination: 27.

Wolsey, L. A. 1998. *Integer programming*. New York, NY, USA: Wiley-Interscience.

Reinforcement Learning for Branch-and-Bound Optimisation using Retrospective Trajectories

Christopher W. F. Parsonson,[1*] **Alexandre Laterre,**[2] **Thomas D. Barrett**[2]

[1]UCL, [2]InstaDeep

Abstract

Combinatorial optimisation problems framed as mixed integer linear programmes (MILPs) are ubiquitous across a range of real-world applications. The canonical branch-and-bound algorithm seeks to exactly solve MILPs by constructing a search tree of increasingly constrained sub-problems. In practice, its solving time performance is dependent on heuristics, such as the choice of the next variable to constrain ('branching'). Recently, machine learning (ML) has emerged as a promising paradigm for branching. However, prior works have struggled to apply reinforcement learning (RL), citing sparse rewards, difficult exploration, and partial observability as significant challenges. Instead, leading ML methodologies resort to approximating high quality handcrafted heuristics with imitation learning (IL), which precludes the discovery of novel policies and requires expensive data labelling. In this work, we propose *retro branching*; a simple yet effective approach to RL for branching. By retrospectively deconstructing the search tree into multiple paths each contained within a sub-tree, we enable the agent to learn from shorter trajectories with more predictable next states. In experiments on four combinatorial tasks, our approach enables learning-to-branch without any expert guidance or pre-training. We outperform the current state-of-the-art RL branching algorithm by 3-5$\times$ and come within 20% of the best IL method's performance on MILPs with 500 constraints and 1000 variables, with ablations verifying that our retrospectively constructed trajectories are essential to achieving these results.

1 Introduction

A plethora of real-world problems fall under the broad category of combinatorial optimisation (CO) (vehicle routing and scheduling (Korte and Vygen 2012); protein folding (Perdomo-Ortiz et al. 2012); fundamental science (Barahona 1982)). Many CO problems can be formulated as mixed integer linear programmes (MILPs) whose task is to assign discrete values to a set of decision variables, subject to a mix of linear and integrality constraints, such that some objective function is maximised or minimised. The most popular method for finding exact solutions to MILPs is branch-and-bound (B&B) (Land and Doig 1960); a collection of heuristics which increasingly tighten the bounds in which an optimal solution can reside (see Section 3). Among the most important of these heuristics is *variable selection* or *branching* (which variable to use to partition the chosen node's search space), which is key to determining B&B solve efficiency (Achterberg and Wunderling 2013).

State-of-the-art (SOTA) learning-to-branch approaches typically use the imitation learning (IL) paradigm to predict the action of a high quality but computationally expensive human-designed branching expert (Gasse et al. 2019). Since branching can be formulated as a Markov decision process (MDP) (He, Daumé, and Eisner 2014), reinforcement learning (RL) seems a natural approach. The long-term motivations of RL include the promise of learning novel policies from scratch without the need for expensive expert data, the potential to exceed expert performance without human design, and the capability to maximise the performance of a policy parameterised by an expressivity-constrained deep neural network (DNN).

However, branching has thus far proved largely intractable for RL for reasons we summarise into three key challenges. (1) *Long episodes:* Whilst even random branching policies are theoretically guaranteed to eventually find the optimal solution, poor decisions can result in episodes of tens of thousands of steps for the 500 constraint 1000 variable MILPs considered by Gasse et al. 2019. This raises the familiar RL challenges of reward sparsity (Trott et al. 2019), credit assignment (Harutyunyan et al. 2019), and high variance returns (Mao et al. 2019). (2) *Large state-action spaces*: Each branching step might have hundreds or thousands of potential branching candidates with a huge number of unique possible sub-MILP states. Efficient exploration to discover improved trajectories in such large state-action spaces is a well-known difficulty for RL (Agostinelli et al. 2019; Ecoffet et al. 2021). (3) *Partial observability*: When a branching decision is made, the next state given to the brancher is determined by the next sub-MILP visited by the node selection policy. Jumping around the B&B tree without the brancher's control whilst having only partial observability of the full tree makes the future states seen by the agent difficult to predict. Etheve et al. 2020 therefore postulated the benefit of keeping the MDP within a sub-tree to improve observability and introduced the SOTA fitting for minimising the sub-tree size (FMSTS) RL branching algorithm. However, in order to achieve this,

*Work undertaken during internship at InstaDeep.
Corresponding email: cwfparsonson@gmail.com

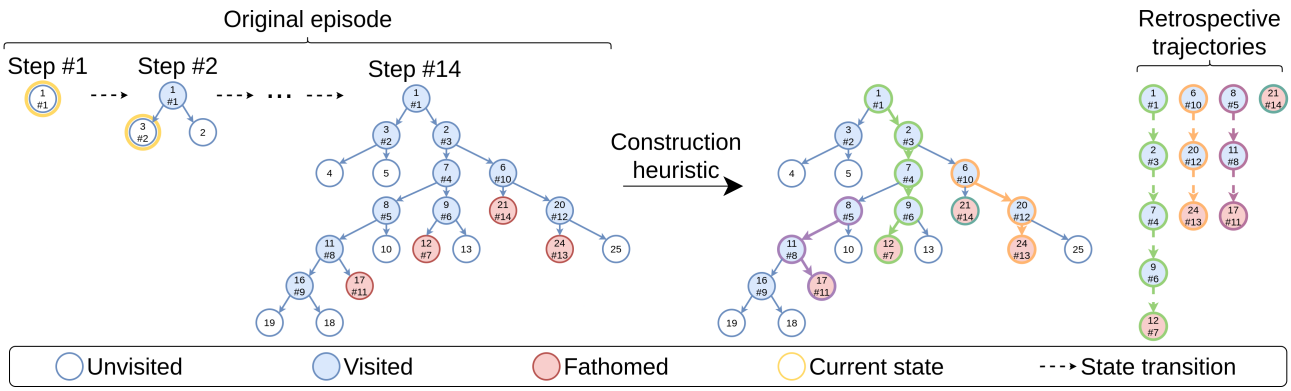


Figure 1: The proposed retro branching approach used during training. Each node is labelled with: Top: The unique ID assigned when it was added to the tree, and (where applicable); bottom: The step number (preceded by a '#') at which it was visited by the brancher in the original Markov decision process (MDP). The MILP is first solved with the brancher and the B&B tree stored as usual (forming the 'original episode'). Then, ignoring any nodes never visited by the agent, the nodes are added to trajectories using some 'construction heuristic' (see Sections 4 and 6) until each eligible node has been added to one, and only one, trajectory. Crucially, the order of the sequential states within a given trajectory may differ from the state visitation order of the original episode, but all states within the trajectory will be within the same sub-tree. These trajectories are then used for training.

FMSTS had to use a depth-first search (DFS) node selection policy which, as we demonstrate in Section 6, is highly sub-optimal and limits scalability.

In this work, we present *retro branching*; a simple yet effective method to overcome the above challenges and learn to branch via reinforcement. We follow the intuition of Etheve et al. (2020) that constraining each sequential MDP state to be within the same sub-tree will lead to improved observability. However, we posit that a branching policy taking the 'best' actions with respect to only the sub-tree in focus can still provide strong overall performance *regardless of the node selection policy used*. This is aligned with the observation that leading heuristics such as SB and PB also do not explicitly account for the node selection policy or predict how the global bound may change as a result of activity in other sub-trees. Assuming the validity of this hypothesis, we can discard the DFS node selection requirement of FMSTS whilst retaining the condition that sequential states seen during training must be within the same sub-tree.

Concretely, our retro branching approach (shown in Figure 1 and elaborated on in Section 4) is to, during training, take the search tree after the B&B instance has been solved and *retrospectively* select each subsequent state (node) to construct multiple trajectories. Each trajectory consists of sequential nodes within a single sub-tree, allowing the brancher to learn from shorter trajectories with lower return variance and more predictable future states. This approach directly addresses challenges (1) and (3) and, whilst the state-action space is still large, the shorter trajectories implicitly define more immediate auxiliary objectives relative to the tree. This reduces the difficulty of exploration since shorter trajectory returns will have a higher probability of being improved upon via stochastic action sampling than when a single long MDP is considered, thereby addressing (2). Furthermore, retro branching relieves the FMSTS requirement that the agent must be trained in a DFS node selection setting, enabling more sophisticated strategies to be used which are better suited for solving larger, more complex MILPs.

We evaluate our approach on MILPs with up to 500 constraints and 1000 variables, achieving a 3-5$\times$ improvement over FMSTS and coming within $\approx 20\%$ of the performance of the SOTA IL agent of Gasse et al. (2019). Furthermore, we demonstrate that, for small instances, retro branching can uncover policies superior to IL; a key motivation of using RL. Our results open the door to the discovery of new branching policies which can scale without the need for labelled data and which could, in principle, exceed the performance of SOTA handcrafted branching heuristics.

2 Related Work

Since the invention of B&B for exact CO by Land and Doig (1960), researchers have sought to design and improve the node selection (tree search), variable selection (branching), primal assignment, and pruning heuristics used by B&B, with comprehensive reviews provided by Achterberg 2007 and Tomazella and Nagano 2020. We focus on branching.

Classical branching heuristics. Pseudocost branching (PB) (Benichou et al. 1971) and strong branching (SB) (Applegate et al. 1995, 2007) are two canonical branching algorithms. PB selects variables based on their historic branching success according to metrics such as bound improvement. Although the per-step decisions of PB are computationally fast, it must initialise the variable pseudocosts in some way which, if done poorly, can be particularly damaging to overall performance since early B&B decisions tend to be the most influential. SB, on the other hand, conducts a one-step lookahead for all branching candidates by computing their potential local dual bound gains before selecting the most favourable variable, and thus is able to make high quality decisions during the critical early stages of the search tree's evolution. Despite its simplicity, SB is still today the best known policy for minimising the overall number of B&B

nodes needed to solve the problem instance (a popular B&B quality indicator). However, its computational cost renders SB infeasible in practice.

Learning-to-branch. Recent advances in deep learning have led machine learning (ML) researchers to contribute to exact CO (surveys provided by Lodi and Zarpellon 2017, Bengio, Lodi, and Prouvost 2021, and Cappart et al. 2021). Khalil et al. 2016 pioneered the community's interest by using IL to train a support vector machine (SVM) to imitate the variable rankings of SB after the first 500 B&B node visits and thereafter use the SVM. Alvarez, Louveaux, and Wehenkel 2017 similarly imitated SB, but learned to predict the SB scores directly using Extremely Randomized Trees (Geurts, Ernst, and Wehenkel 2006). These approaches performed promisingly, but their per-instance training and use of SB at test time limited their scalability.

These issues were overcome by Gasse et al. 2019, who took as input a bipartite graph representation capturing the current B&B node state and predicted the corresponding action chosen by SB using a graph convolutional network (GCN). This alleviated the reliance on extensive feature engineering, avoided the use of SB at inference time, and demonstrated generalisation to larger instances than seen in training. Works since have sought to extend this method by introducing new observation features to generalise across heterogeneous CO instances (Zarpellon et al. 2021) and designing SB-on-a-GPU expert labelling methods for scalability (Nair et al. 2021).

Etheve et al. 2020 proposed FMSTS which, to the best of our knowledge, is the only published work to apply RL to branching and is therefore the SOTA RL branching algorithm. By using a DFS node selection strategy, they used the deep Q-network (DQN) approach (Mnih et al. 2013) to approximate the Q-function of the B&B sub-tree size rooted at the current node; a local Q-function which, in their setting, was equivalent to the number of global tree nodes. Although FMSTS alleviated issues with credit assignment and partial observability, it relied on using the DFS node selection policy (which can be far from optimal), was fundamentally limited by exponential sub-tree sizes produced by larger instances, and its associated models and data sets were not open-accessed.

3 Background

Mixed integer linear programming. An MILP is an optimisation task where values must be assigned to a set of n *decision variables* subject to a set of m linear *constraints* such that some linear *objective function* is minimised. MILPs can be written in the standard form

$$\arg\min_{\mathbf{x}} \left\{ \mathbf{c}^\top \mathbf{x} | \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \right\}, \quad (1)$$

where $\mathbf{c} \in \mathbb{R}^n$ is a vector of the objective function's coefficients for each decision variable in $\mathbf{x}$ such that $\mathbf{c}^\top \mathbf{x}$ is the objective value, $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a matrix of the m constraints' coefficients (rows) applied to n variables (columns), $\mathbf{b} \in \mathbb{R}^m$ is the vector of variable constraint right-hand side bound values which must be adhered to, and $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ are the respective lower and upper variable value bounds. MILPs are hard to solve owing to their *integrality constraint(s)* whereby $p \leq n$ decision variables must be an integer. If these integrality constraints are relaxed, the MILP becomes a linear programme (LP), which can be solved efficiently using algorithms such as simplex (Nelder and Mead 1965). The most popular approach for solving MILPs exactly is B&B.

Branch-and-bound. B&B is an algorithm composed of multiple heuristics for solving MILPs. It uses a search tree where nodes are MILPs and edges are partition conditions (added constraints) between them. Using a *divide and conquer* strategy, the MILP is iteratively partitioned into sub-MILPs with smaller solution spaces until an optimal solution (or, if terminated early, a solution with a worst-case optimality gap guarantee) is found. The task of B&B is to evolve the search tree until the provably optimal node is found.

Concretely, as summarised in Figure 2, at each step in the algorithm, B&B: (1) Selects an open (unfathomed leaf) node in the tree whose sub-tree seems promising to evolve; (2) selects ('branches on') a variable to tighten the bounds on the sub-MILP's solution space by adding constraints either side of the variable's LP solution value, generating two child nodes (sub-MILPs) beneath the focus node; (3) for each child, i) solve the relaxed LP (the *dual problem*) to get the *dual bound* (a bound on the best possible objective value in the node's sub-tree) and, where appropriate, ii) solve the *primal problem* and find a feasible (but not necessarily optimal) solution satisfying the node's constraints, thus giving the *primal bound* (the worst-case feasible objective value in the sub-tree); and (4) fathom any children (i.e. consider the sub-tree rooted at the child 'fully known' and therefore excluded from any further exploration) whose relaxed LP solution is integer-feasible, is worse than the incumbent (the globally best feasible node found so far), or which cannot meet the non-integrality constraints of the MILP. This process is repeated until the *primal-dual gap* (global primal-dual bound difference) is 0, at which point a provably optimal solution to the original MILP will have been found.

Note that the heuristics (i.e. primal, branching, and node selection) at each stage jointly determine the performance of B&B. More advanced procedures such as cutting planes (Mitchell 2009) and column generation (Barnhart et al. 1998) are available for enhancement, but are beyond the scope of this work. Note also that solvers such as SCIP 2022 only store 'visitable' nodes in memory, therefore in practice fathoming occurs at a feasible node where a branching decision led to the node's two children being outside the established optimality bounds, being infeasible, or having an integer-feasible dual solution, thereby closing the said node's sub-tree.

Q-learning. Q-learning is typically applied to sequential decision making problems formulated as an MDP defined by tuple $\{\mathcal{S}, \mathcal{U}, \mathcal{T}, \mathcal{R}, \gamma\}$. $\mathcal{S}$ is a finite set of states, $\mathcal{U}$ a set of actions, $\mathcal{T} : \mathcal{S} \times \mathcal{U} \times \mathcal{S} \rightarrow [0, 1]$ a transition function from state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$ given action $u \in U$, $\mathcal{R} : \mathcal{S} \rightarrow \mathbb{R}$ a function returning a scalar reward from s, and $\gamma \in [0, 1]$ a factor by which to discount expected future returns to their present value. It is an off-policy temporal difference method which aims to learn the action-value function mapping state-action pairs to the expected discounted sum of their immediate and future re-

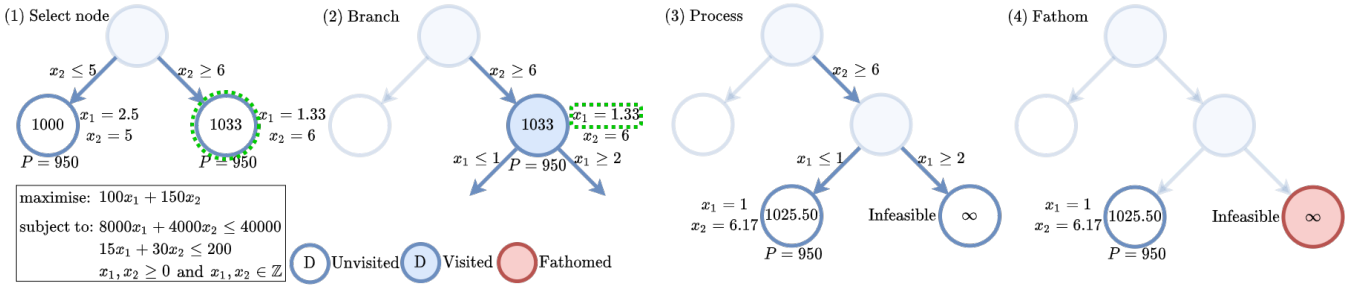


Figure 2: Typical 4-stage procedure iteratively repeated by B&B to solve an MILP. Each node represents an MILP derived from the original MILP being solved, and each edge represents the constraint added to derive a new child node (sub-MILP) from a given parent. Each node is labelled with the decision variable values of the solved LP relaxation on the right hand side, the corresponding dual bound in the centre, and the established primal bound beneath. Each edge is labelled with the introduced constraint to generate the child node. Green dotted outlines are used to indicate which node and variable were selected in stages (1) and (2) to lead to stages (3) and (4). The global primal (P) and dual (D) bounds are increasingly constrained by repeating stages 1-4 until P and D are equal, at which point a provably optimal solution will have been found. Note that for clarity we only show the detailed information needed at each stage, but that this does not indicate any change to the state of the tree.

wards when following a policy $\pi : \mathcal{S} \to \mathcal{U}$, $Q^\pi(s,u) = \mathbb{E}_\pi\left[\sum_{t'=t+1}^{\infty} \gamma_{t'-1} r(s_{t'}) | s_t{=}s, u_t{=}u\right]$. By definition, an optimal policy π_* will select an action which maximises the true Q-value $Q_*(s,u)$, $\pi_*(s) = \arg\max_{u'} Q_*(s,u')$. For scalability, DQN (Mnih et al. 2013; van Hasselt, Guez, and Silver 2015) approximates this true Q-function using a DNN parameterised by θ such that $Q_\theta(s,u) \approx Q_*(s,u)$.

4 Retro Branching

We now describe our retro branching approach for learning-to-branch with RL.

States. At each time step t the B&B solver state is comprised of the search tree with past branching decisions, per-node LP solutions, the global incumbent, the currently focused leaf node, and any other solver statistics which might be tracked. To convert this information into a suitable input for the branching agent, we represent the MILP of the focus node chosen by the node selector as a bipartite graph. Concretely, the n variables and m constraints are connected by edges denoting which variables each constraint applies to. This formulation closely follows the approach of Gasse et al. 2019, with a full list of input features at each node detailed in Appendix E.

Actions. Given the MILP state s_t of the current focus node, the branching agent uses a policy $\pi(u_t|s_t)$ to select a variable u_t from among the p branching candidates.

Original full episode transitions. In the original full B&B episode, the next node visited is chosen by the node selection policy from amongst any of the open nodes in the tree. This is done independently of the brancher, which observes state information related only to the current focus node and the status of the global bounds. As such, the transitions of the 'full episode' are partially observable to the brancher, and it will therefore have the challenging task of needing to aggregate over unobservable states in external sub-trees to predict the long-term values of states and actions.

Retrospectively constructed trajectory transitions (retro branching). To address the partial observability of the full episode, we retrospectively construct multiple trajectories where all sequential states in a given trajectory are within the same sub-tree, and where the trajectory's terminal state is chosen from amongst the as yet unchosen fathomed sub-tree leaves. A visualisation of our approach is shown in Figure 1. Concretely, during training, we first solve the instance as usual with the RL brancher and any node selection heuristic to form the 'original episode'. When the instance is solved, rather than simply adding the originally observed MDP's transitions to the DQN replay buffer, we retrospectively construct multiple trajectory paths through the search tree. This construction process is done by starting at the highest level node not yet added to a trajectory, selecting an as yet unselected fathomed leaf in the sub-tree rooted at said node using some 'construction heuristic' (see Section 6), and using this root-leaf pair as a source-destination with which to construct a path (a 'retrospective trajectory'). This process is iteratively repeated until each eligible node in the original search tree has been added to one, and only one, retrospective trajectory. The transitions of each trajectory are then added to the experience replay buffer for learning. Note that retrospective trajectories are only used during training, therefore retro branching agents have no additional inference-time overhead.

Crucially, retro branching determines the sequence of states in each trajectory (i.e. the transition function of the MDP) such that the next state(s) observed in a given trajectory will *always* be within the same sub-tree (see Figure 1) regardless of the node selection policy used in the original B&B episode. Our reasoning behind this idea is that the state(s) beneath the current focus node within its sub-tree will have characteristics (bounds, introduced constraints, etc.) which are strongly related with those of the current node, making them more observable than were the next states to be chosen from elsewhere in the search tree, as can occur in the 'original B&B' episode. Moreover, by correlating the agent's maximum trajectory length with the depth of the tree rather than the total number of nodes, reconstructed trajectories

have orders of magnitude fewer steps and lower return variance than the original full episode, making learning tractable on large MILPs. Furthermore, because the sequential nodes visited are chosen retrospectively in each trajectory, unlike with FMSTS, any node selection policy can be used during training. As we show in Section 6, this is a significant help when solving large and complex MILPs.

Rewards. As demonstrated in Section 6, the use of reconstructed trajectories enables a simple distance-to-goal reward function to be used; a $r = -1$ punishment is issued to the agent at each step except when the agent's action fathomed the sub-tree, where the agent receives $r = 0$. This provides an incentive for the the branching agent to reach the terminal state as quickly as possible. When aggregated over all trajectories in a given sub-tree, this auxiliary objective corresponds to fathoming the whole sub-tree (and, by extension, solving the MILP) in as few steps as possible. This is because the only nodes which are stored by SCIP 2022 and which the brancher will be presented with will be feasible nodes which *potentially* contain the optimal solution beneath them. As such, any action chosen by the brancher which provably shows either the optimal solution to not be beneath the current node or which finds an integer feasible dual solution (i.e. an action which fathoms the sub-tree beneath the node) will be beneficial, because it will prevent SCIP from being able to further needlessly explore the node's sub-tree.

A note on partial observability. In the above retrospective formulation of the branching MDP, the primal, branching, and node selection heuristics active in other sub-trees will still influence the future states and fathoming conditions of a given retrospective trajectory. We posit that there are two extremes; DFS node selection where future states are fully observable to the brancher, and non-DFS node selection where they are heavily obscured. As shown in Section 6, our retrospective node selection setting strikes a balance between these two extremes, attaining sufficient observability to facilitate learning while enabling the benefits of short, low variance trajectories with sophisticated node selection strategies which make handling larger MILPs tractable.

5 Experimental Setup

All code for reproducing the experiments and links to the generated data sets are provided at https://github.com/cwfparsonson/retro_branching.

Network architecture and learning algorithm. We used the GCN architecture of Gasse et al. 2019 to parameterise the DQN value function with some minor modifications which we found to be helpful (see Appendix B.1). We trained our network with n-step DQN (Sutton 1988; Mnih et al. 2013) using prioritised experience replay (Schaul et al. 2016), soft target network updates (Lillicrap et al. 2019), and an epsilon-stochastic exploration policy (see Appendix A.1 for a detailed description of our RL approach and the corresponding algorithms and hyperparameters used).

B&B environment. We used the open-source Ecole (Prouvost et al. 2020) and PySCIPOpt (Maher et al. 2016) libraries with SCIP 7.0.1 (SCIP 2022) as the backend solver to do instance generation and testing. Where possible, we used the training and testing protocols of Gasse et al. (2019).

MILP Problem classes. In total, we considered four NP-hard problem benchmarks: set covering (Balas, Ho, and Center 2018), combinatorial auction (Leyton-Brown, Pearson, and Shoham 2000), capacitated facility location (Litvinchev and Ozuna Espinosa 2012), and maximum independent set (Bergman et al. 2016).

Baselines. We compared retro branching against the SOTA FMSTS RL algorithm of Etheve et al. (2020) (see Appendix F for implementation details) and the SOTA IL approach of Gasse et al. (2019) trained and validated with $100\,000$ and $20\,000$ strong branching samples respectively. For completeness, we also compared against the SB heuristic imitated by the IL agent, the canonical PB heuristic, and a random brancher (equivalent in performance to most infeasible branching (Achterberg, Koch, and Martin 2004)). Note that we have ommited direct comparison to the SOTA tuned commercial solvers, which we do not claim to be competitive with at this stage. To evaluate the quality of the agents' branching decisions, we used 100 validation instances (see Appendix C for an analysis of this data set size) which were unseen during training, reporting the total number of tree nodes and LP iterations as key metrics to be minimised.

6 Results & Discussion

6.1 Performance of Retro Branching

Comparison to the SOTA RL branching heuristics. We considered set covering instances with 500 rows and 1000 columns. To demonstrate the benefit of the proposed retro branching method, we trained a baseline 'Original' agent on the original full episode, receiving the same reward as our retro branching agent (-1 at each non-terminal step and 0 for a terminal action which ended the episode). We also trained the SOTA RL FMSTS branching agent in a DFS setting and, at test time, validated the agent in both a DFS ('FMSTS-DFS') and non-DFS ('FMSTS') environment to fairly compare the policies. Note that the FMSTS agent serves as an ablation to analyse the influence of training on retrospective trajectories, since it uses our auxiliary objective but without retrospective trajectories, and that the Original agent further ablates the auxiliary objective since its 'terminal step' is defined as ending the B&B episode (where it receives $r_t = 0$ rather than $r_t = -1$). As shown in Figure 3a, the Original agent was unable to learn on these large instances, with retro branching achieving $14\times$ fewer nodes at test time. FMSTS also performed poorly, with highly unstable learning and a final performance $5\times$ and $3\times$ poorer than retro branching in the DFS and non-DFS settings respectively (see Figure 3c). We posit that the cause of the poor FMSTS performance is due to its use of the sub-optimal DFS node selection policy, which is ill-suited for handling large MILPs and results in $\approx 10\%$ of episodes seen during training being on the order of 10-100k steps long (see Figure 3b), which makes learning significantly harder for RL.

Comparison to non-RL branching heuristics. Having demonstrated that the proposed retro branching method makes learning-to-branch at scale tractable for RL, we now compare retro branching with the baseline branchers to understand the efficacy of RL in the context of the current literature.

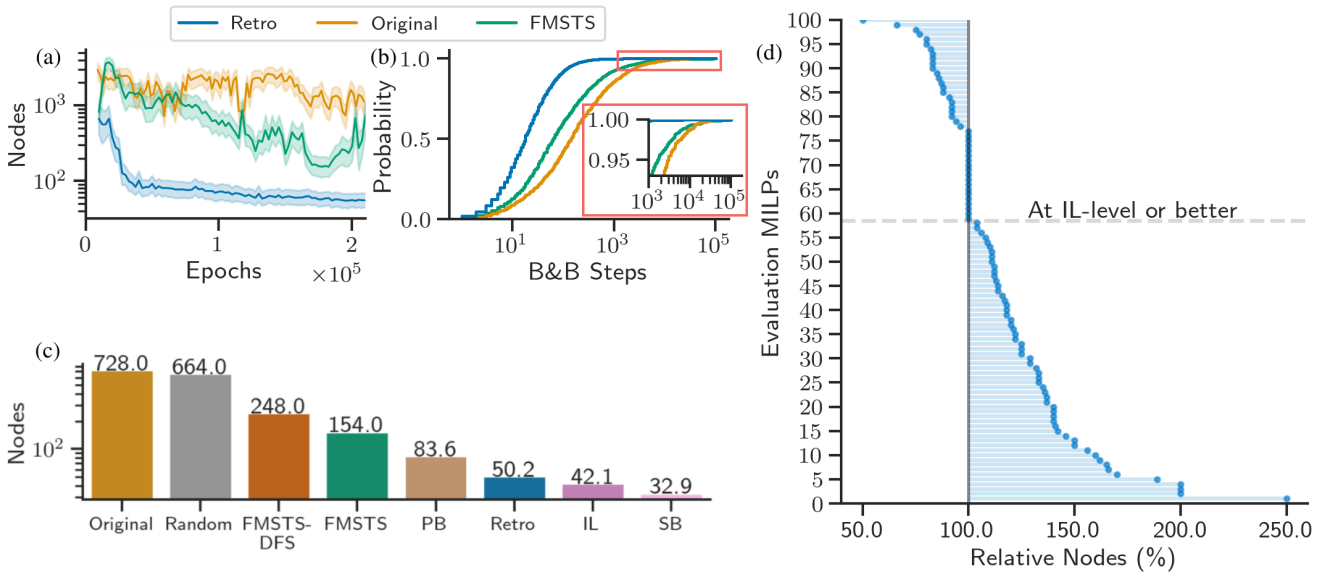


Figure 3: Performances of the branching agents on the 500×1000 set covering instances. (a) Validation curves for the RL agents evaluated in the same non-DFS setting. (b) CDF of the number of B&B steps taken by the RL agents for each instance seen during training. (c) The best validation performances of each branching agent. (d) The instance-level validation performance of the retro branching agent relative to the IL agent, with RL matching or beating IL on 42% of test instances.

Figure 3c shows how retro branching compares to other policies on large 500×1000 set covering instances. While the agent outperforms PB, it only matches or beats IL on 42% of the test instances (see Figure 3d) and, on average, has a $\approx 20\%$ larger B&B tree size. Therefore although our RL agent was still improving and was limited by compute (see Appendix A.2), and in spite of our method outperforming the current SOTA FMSTS RL brancher, RL has not yet been able to match or surpass the SOTA IL agent at scale. This will be an interesting area of future work, as discussed in Section 7.

6.2 Analysis of Retro Branching

Verifying that RL can outperform IL. In addition to not needing labelled data, a key motivation for using RL over IL for learning-to-branch is the potential to discover superior policies. While Figure 3 showed that, at test-time, retro branching matched or outperformed IL on 42% of instances, IL still had a lower average tree size. As shown in Table 1, we found that, on small set covering instances with 165 constraints and 230 variables, RL could outperform IL by $\approx 20\%$. While improvement on problems of this scale is not the primary challenge facing ML-B&B solvers, we are encouraged by this demonstration that it is possible for an RL agent to learn a policy better able to maximise the performance of an expressivity-constrained network than imitating an expert such as SB without the need for pre-training or expensive data labelling procedures (see Appendix H).

For completeness, Table 1 also compares the retro branching agent to the IL, PB, and SB branching policies evaluated on 100 unseen instances of four NP-hard CO benchmarks. We considered instances with 10 items and 50 bids for combinatorial auction, 5 customers and facilities for capacitated facility location, and 25 nodes for maximum independent set. RL achieved a lower number of tree nodes than PB and IL on all problems except combinatorial auction. This highlights the potential for RL to learn improved branching policies to solve a variety of MILPs.

Demonstrating the independence of retro branching to future state selection. As described in Section 4, in order to retrospectively construct a path through the search tree, a fathomed leaf node must be selected. We refer to the method for selecting the leaf node as the *construction heuristic*. The future states seen by the agent are therefore determined by the construction heurisitc (used in training) and the node selection heuristic (used in training and inference).

During our experiments, we found that the specific construction heuristic used had little impact on the performance of our agent. Figure 4a shows the validation curves for four agents trained on 500×1000 set covering instances each using one of the following construction heuristics: Maximum LP gain ('MLPG': Select the leaf with the largest LP gain); random ('R': Randomly select a leaf); visitation order ('VO': Select the leaf which was visited first in the original episode); and deepest ('D': Select the leaf which results in the longest trajectory). As shown, all construction heuristics resulted in roughly the same performance (with MLPG performing only slightly better). This suggests that the agent learns to reduce the trajectory length regardless of the path chosen by the construction heuristic. Since the specific path chosen is independent of node selection, we posit that the relative strength of an RL agent trained with retro branching will also be independent of the node selection policy used.

To test this, we took our best retro branching agent trained with the default SCIP node selection heuristic and tested it

Table 1: Test-time comparison of the best agents on the evaluation instances of the four NP-hard small CO problems considered.

	Set Covering		Combinatorial Auction		Capacitated Facility Location		Maximum Independent Set	
Method	# LPs	# Nodes	# LPs	# Nodes	# LPs	# Nodes	# LPs	# Nodes
SB	184	6.76	13.2	4.64	28.2	10.2	19.2	3.80
PB	258	12.8	22.0	7.80	**28.0**	10.2	25.4	5.77
IL	244	10.5	**16.0**	**5.29**	**28.0**	10.2	20.1	4.08
Retro	**206**	**8.68**	18.1	5.73	28.4	**10.1**	**19.1**	**4.01**

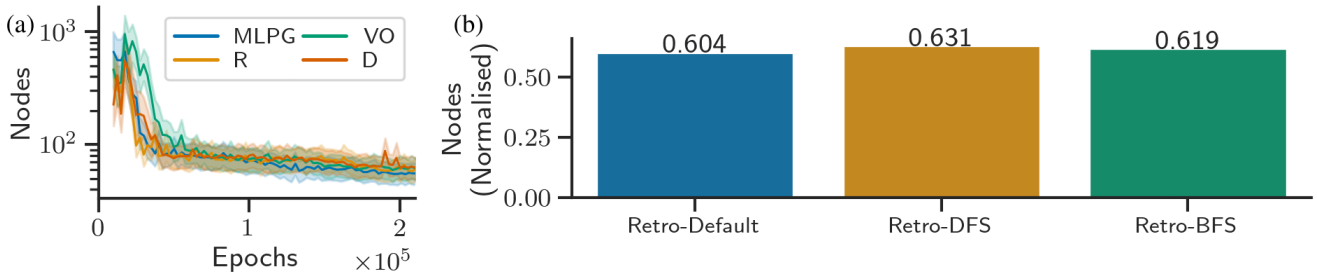


Figure 4: 500×1000 set covering performances. (a) Validation curves for four retro branching agents each trained with a different trajectory construction heuristic: Maximum LP gain (MLPG); random (R); visitation order (VO); and deepest (D). (b) The performances of the best retro branching agent deployed in three different node selection environments (default SCIP, DFS, and breadth-first search (BFS)) normalised relative to the performances of PB (measured by number of tree nodes).

on the 500×1000 validation instances in the default, DFS, and breadth-first search (BFS) SCIP node selection settings. To make the performances of the brancher comparable across these settings, we normalised the mean tree sizes with those of PB (a branching heuristic independent of the node selector) to get the performance relative to PB in each environment. As shown in Figure 4b, our agent achieved consistent relative performance regardless of the node selection policy used, indicating its indifference to the node selector.

7 Conclusions

We have introduced retro branching; a retrospective approach to constructing B&B trajectories in order to aid learning-to-branch with RL. We posited that retrospective trajectories address the challenges of long episodes, large state-action spaces, and partially observable future states which otherwise make branching an acutely difficult task for RL. We empirically demonstrated that retro branching outperforms the current SOTA RL method by 3-5$\times$ and comes within 20% of the performance of IL whilst matching or beating it on 42% of test instances. Moreover, we showed that RL can surpass the performance of IL on small instances, exemplifying a key advantage of RL in being able to discover novel performance-maximising policies for expressivity-constrained networks without the need for pre-training or expert examples. However, retro branching was not able to exceed the IL agent at scale. In this section we outline areas of further work.

Partial observability. A limitation of our proposed approach is the remaining partial observability of the MDP, with activity external to the current sub-tree and branching decision influencing future bounds, states, and rewards. In this and other studies, variable and node selection have been considered in isolation. An interesting approach would be to combine node and variable selection, giving the agent full control over how the B&B tree is evolved.

Reward function. The proposed trajectory reconstruction approach can facilitate a simple RL reward function which would otherwise fail were the original 'full' tree episode used. However, assigning a -1 reward at each step in a given trajectory ignores the fact that certain actions, particularly early on in the B&B process, can have significant influence over the length of multiple trajectories. This could be accounted for in the reward signal, perhaps by using a retrospective backpropagation method (similar to value backpropagation in Monte Carlo tree search (Silver et al. 2016, 2017)).

Exploration. The large state-action space and the complexity of making thousands of sequential decisions which together influence final performance in complex ways makes exploration in B&B an acute challenge for RL. One reason for RL struggling to close the 20% performance gap with IL at scale could be that, at some point, stochastic action sampling to explore new policies is highly unlikely to find trajectories with improved performance. As such, more sophisticated exploration strategies could be promising, such as novel experience intrinsic reward signals (Burda et al. 2018; Zhang et al. 2021), reverse backtracking through the episode to improve trajectory quality (Salimans and Chen 2018; Agostinelli et al. 2019; Ecoffet et al. 2021), and avoiding local optima using auxiliary distance-to-goal rewards (Trott et al. 2019) or evolutionary strategies (Conti et al. 2018).

8 Acknowledgments

We would like to thank Maxime Gasse, Antoine Prouvost, and the rest of the Ecole development team for answering our questions on SCIP and Ecole, and also the anonymous reviewers for their constructive comments on earlier versions of this paper.

References

Achterberg, T. 2007. *Constraint Integer Programming*. Doctoral thesis, Technische Universität Berlin, Fakultät II - Mathematik und Naturwissenschaften, Berlin.

Achterberg, T.; Koch, T.; and Martin, A. 2004. Branching rules revisited. Technical Report 04-13, ZIB, Takustr. 7, 14195 Berlin.

Achterberg, T.; and Wunderling, R. 2013. *Mixed Integer Programming: Analyzing 12 Years of Progress*.

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8): 356–363.

Alvarez, A. M.; Louveaux, Q.; and Wehenkel, L. 2017. A Machine Learning-Based Approximation of Strong Branching. *INFORMS J. Comput.*, 29: 185–195.

Applegate, D. L.; Bixby, R. E.; Chvatal, V.; and Cook, W. J. 2007. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.

Applegate, D. L.; Bixpy, R. E.; Chvatal, V.; and Cook, W. J. 1995. Finding cuts in the TSP (A preliminary report). Technical report, DIMACS.

Balas, E.; Ho, A.; and Center, C. M. U. R. 2018. Set covering algorithms using cutting planes, heuristics, and subgradient optimization : a computational study.

Barahona, F. 1982. On the computational complexity of Ising spin glass models. *Journal of Physics A: Mathematical and General*, 15(10): 3241.

Barnhart, C.; Johnson, E. L.; Nemhauser, G. L.; Savelsbergh, M. W. P.; and Vance, P. H. 1998. Branch-And-Price: Column Generation for Solving Huge Integer Programs. *Oper. Res.*, 46(3): 316–329.

Bengio, Y.; Lodi, A.; and Prouvost, A. 2021. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2): 405–421.

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

Bergman, D.; Cire, A. A.; Hoeve, W.-J. v.; and Hooker, J. 2016. *Decision Diagrams for Optimization*. Springer Publishing Company, Incorporated, 1st edition. ISBN 3319428470.

Burda, Y.; Edwards, H.; Storkey, A.; and Klimov, O. 2018. Exploration by Random Network Distillation.

Cappart, Q.; Chételat, D.; Khalil, E.; Lodi, A.; Morris, C.; and Veličković, P. 2021. Combinatorial optimization and reasoning with graph neural networks.

Conti, E.; Madhavan, V.; Such, F. P.; Lehman, J.; Stanley, K. O.; and Clune, J. 2018. Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, 5032–5043. Red Hook, NY, USA: Curran Associates Inc.

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

Ecoffet, A.; Huizinga, J.; Lehman, J.; Stanley, K. O.; and Clune, J. 2021. First return, then explore. *Nature*, 590(7847): 580–586.

Etheve, M.; Alès, Z.; Bissuel, C.; Juan, O.; and Kedad-Sidhoum, S. 2020. Reinforcement Learning for Variable Selection in a Branch and Bound Algorithm. *Lecture Notes in Computer Science*, 176–185.

Gasse, M.; Chételat, D.; Ferroni, N.; Charlin, L.; and Lodi, A. 2019. *Exact Combinatorial Optimization with Graph Convolutional Neural Networks*. Red Hook, NY, USA: Curran Associates Inc.

Geurts, P.; Ernst, D.; and Wehenkel, L. 2006. Extremely randomized trees. *Machine Learning*, 63(1): 3–42.

Harutyunyan, A.; Dabney, W.; Mesnard, T.; Heess, N.; Azar, M. G.; Piot, B.; van Hasselt, H.; Singh, S.; Wayne, G.; Precup, D.; and Munos, R. 2019. *Hindsight Credit Assignment*. Red Hook, NY, USA: Curran Associates Inc.

He, H.; Daumé, H.; and Eisner, J. 2014. Learning to Search in Branch-and-Bound Algorithms. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, 3293–3301. Cambridge, MA, USA: MIT Press.

Hessel, M.; Modayil, J.; van Hasselt, H.; Schaul, T.; Ostrovski, G.; Dabney, W.; Horgan, D.; Piot, B.; Azar, M.; and Silver, D. 2017. Rainbow: Combining Improvements in Deep Reinforcement Learning.

Khalil, E. B.; Bodic, P. L.; Song, L.; Nemhauser, G. L.; and Dilkina, B. N. 2016. Learning to Branch in Mixed Integer Programming. In *AAAI*.

Korte, B. H.; and Vygen, J. 2012. *Combinatorial Optimization: Theory and Algorithms*. New York, NY: Springer-Verlag. ISBN 9783642244889 3642244882 3642244874 9783642244872.

Land, A. H.; and Doig, A. G. 1960. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3): pp. 497–520.

Leyton-Brown, K.; Pearson, M.; and Shoham, Y. 2000. Towards a Universal Test Suite for Combinatorial Auction Algorithms. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, EC '00, 66–76. New York, NY, USA: Association for Computing Machinery. ISBN 1581132727.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2019. Continuous control with deep reinforcement learning.

Lin, L.-J. 1992. Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Mach. Learn.*, 8(3–4): 293–321.

Litvinchev, I.; and Ozuna Espinosa, E. L. 2012. Solving the Two-Stage Capacitated Facility Location Problem by the Lagrangian Heuristic. In Hu, H.; Shi, X.; Stahlbock, R.; and Voß, S., eds., *Computational Logistics*, 92–103. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-33587-7.

Lodi, A.; and Zarpellon, G. 2017. On learning and branching: a survey. *TOP*, 25(2): 207–236.

Maher, S.; Miltenberger, M.; Pedroso, J. P.; Rehfeldt, D.; Schwarz, R.; and Serrano, F. 2016. PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite. In *Mathematical Software – ICMS 2016*, 301–307. Springer International Publishing.

Mao, H.; Venkatakrishnan, S. B.; Schwarzkopf, M.; and Alizadeh, M. 2019. Variance Reduction for Reinforcement Learning in Input-Driven Environments. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Mitchell, J. E. 2009. *Integer programming: branch and cut algorithmsInteger Programming: Branch and Cut Algorithms*, 1643–1650. Boston, MA: Springer US. ISBN 978-0-387-74759-0.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with Deep Reinforcement Learning.

Nair, V.; Bartunov, S.; Gimeno, F.; von Glehn, I.; Lichocki, P.; Lobov, I.; O'Donoghue, B.; Sonnerat, N.; Tjandraatmadja, C.; Wang, P.; Addanki, R.; Hapuarachchi, T.; Keck, T.; Keeling, J.; Kohli, P.; Ktena, I.; Li, Y.; Vinyals, O.; and Zwols, Y. 2021. Solving Mixed Integer Programs Using Neural Networks.

Nelder, J. A.; and Mead, R. 1965. A simplex method for function minimization. *Computer Journal*, 7: 308–313.

Perdomo-Ortiz, A.; Dickson, N.; Drew-Brook, M.; Rose, G.; and Aspuru-Guzik, A. 2012. Finding low-energy conformations of lattice protein models by quantum annealing. *Scientific Reports*, 2: 571.

Prouvost, A.; Dumouchelle, J.; Scavuzzo, L.; Gasse, M.; Chételat, D.; and Lodi, A. 2020. Ecole: A Gym-like Library for Machine Learning in Combinatorial Optimization Solvers. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*.

Salimans, T.; and Chen, R. 2018. Learning Montezuma's Revenge from a Single Demonstration.

Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2016. Prioritized Experience Replay.

SCIP. 2022. The SCIP Optimization Suite 7.0. Technical report, Optimization Online.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587): 484–489.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; van den Driessche, G.; Graepel, T.; and Hassabis, D. 2017. Mastering the game of Go without human knowledge. *Nature*, 550(7676): 354–359.

Sutton, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1): 9–44.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

Tomazella, C. P.; and Nagano, M. S. 2020. A comprehensive review of Branch-and-Bound algorithms: Guidelines and directions for further research on the flowshop scheduling problem. *Expert Systems with Applications*, 158: 113556.

Trott, A.; Zheng, S.; Xiong, C.; and Socher, R. 2019. Keeping Your Distance: Solving Sparse Reward Tasks Using Self-Balancing Shaped Rewards. In *NeurIPS*.

van Hasselt, H.; Guez, A.; and Silver, D. 2015. Deep Reinforcement Learning with Double Q-learning.

Zarpellon, G.; Jo, J.; Lodi, A.; and Bengio, Y. 2021. Parameterizing Branch-and-Bound Search Trees to Learn Branching Policies. arXiv:2002.05120.

Zhang, T.; Xu, H.; Wang, X.; Wu, Y.; Keutzer, K.; Gonzalez, J. E.; and Tian, Y. 2021. NovelD: A Simple yet Effective Exploration Criterion. In Beygelzimer, A.; Dauphin, Y.; Liang, P.; and Vaughan, J. W., eds., *Advances in Neural Information Processing Systems*.

A RL Training

A.1 Training Parameters

The RL training hyperparameters are summarised in Table 2. We used n-step DQN (Sutton 1988; Mnih et al. 2013) with prioritised experience replay (Schaul et al. 2016), with overviews of each of these approaches provided below. For exploration, we followed an ϵ-stochastic policy ($\epsilon \in [0, 1]$) whereby the probabilities for action selection were ϵ for a random action and $1 - \epsilon$ for an action sampled from the softmax probability distribution over the Q-values of the branching candidates. We also found it helpful for learning stability to clip the gradients of our network before applying parameter updates.

Conventional DQN. At each time step t during training, $Q_\theta(s, u)$ is used with an exploration strategy to select an action and add the observed transition $T = (s_t, u_t, r_{t+1}, \gamma_{t+1}, s_{t+1})$ to a replay memory buffer (Lin 1992). The network's parameters θ are then optimised with stochastic gradient descent to minimise the mean squared error loss between the *online* network's predictions and a bootstrapped estimate of the Q-value,

$$J_{DQN}(Q) = \left[r_{t+1} + \gamma_{t+1} \max_{u'} Q_{\bar{\theta}}(s_{t+1}, u') - Q_\theta(s_t, u_t)\right]^2, \tag{2}$$

where t is a time step randomly sampled from the buffer and $Q_{\bar{\theta}}$ a *target* network with parameters $\bar{\theta}$ which are periodically copied from the acting online network. The target network is not directly optimised, but is used to provide the bootstrapped Q-value estimates for the loss function.

Training Parameter	Value
Batch size	64 (128)
Actor steps per learner update	5 (10)
Learning rate	5e−5
Discount factor	0.99
Optimiser	Adam
Buffer size $\lvert\mathcal{M}\rvert_{\text{init}}$	20e3
Buffer size $\lvert\mathcal{M}\rvert_{\text{capacity}}$	100e3
Prioritised experience replay β_{init}	0.4
Prioritised experience replay β_{final}	1.0
$\beta_{\text{init}} \rightarrow \beta_{\text{final}}$ learner steps	5e3
Prioritised experience replay α	0.6
Minimum experience priority	1e−3
Soft target network update τ_{soft}	1e−4
Gradient clip value	10
n-step DQN n	3
Exploration probability ϵ	2.5e−2

Table 2: Training parameters used for training the RL agent. All parameters were kept the same across CO instances except for the large 500×1000 set covering instances, which we used a larger batch size and actor steps per learner update (specified in brackets).

Prioritised experience replay. Vanilla DQN replay buffers are sampled uniformly to obtain transitions for network updates. A preferable approach is to more frequently sample transitions from which there is much to learn. Prioritised experience replay (Schaul et al. 2016) deploys this intuition by sampling transitions with probability p_t proportional to the last encountered absolute temporal difference error,

$$p_t \propto |r_{t+1} + \gamma_{t+1} \max_{u'} Q_{\bar{\theta}}(s_{t+1}, u') - Q_\theta(s_t, u_t)|^\omega, \quad (3)$$

where ω is a tuneable hyperparameter for shaping the probability distribution. New transitions are added to the replay buffer with maximum priority to ensure all experiences will be sampled at least once to have their errors evaluated.

n-Step Q-learning. Traditional Q-learning uses the target network's greedy action at the next step to bootstrap a Q-value estimate for the temporal difference target. Alternatively, to improve learning speeds and help with convergence (Sutton and Barto 2018; Hessel et al. 2017), forward-view *multi-step* targets can be used (Sutton 1988), where the n-step discounted return from state s is

$$r_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} r_{t+k+1}, \quad (4)$$

resulting in an n-step DQN loss of

$$J_{DQN_n}(Q) = \left[r_t^{(n)} + \gamma_t^{(n)} \max_{u'} Q_{\bar{\theta}}(s_{t+n}, u') - Q_\theta(s_t, u_t)\right]^2. \quad (5)$$

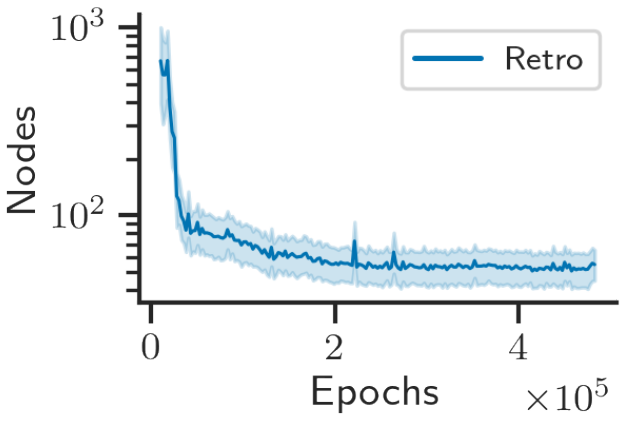


Figure 5: Validation curve for the retro branching agent on the 500×1000 set covering test instances. Although most performance gains were made in the first $\approx$ 200k epochs, the agent did not stop improving, with the last recorded checkpoint improvement at 485k epochs.

A.2 Training Time and Convergence

To train our RL agent, we had a compute budget limited to one A100 GPU which was shared by other researchers from different groups. This resulted in highly variable training times. On average, one epoch on the large 500×1000 set covering instances took roughly 0.42 seconds (which includes the time to act in the B&B environment to collect and save the experience transitions, sample from the buffer, make online vs. target network predictions, update the network, etc.). Therefore training for 200k epochs (roughly the amount needed to converge on a strong policy within $\approx 20\%$ of the imitation agent) took 5-6 days.

As shown in Figure 5, when we left our retro branching agent to train for $\approx$ 13 days ($\approx$ 500k epochs), although most performance gains had been made in the first $\approx$ 200k epochs, the agent never stopped improving (the last improved checkpoint was at 485k epochs). A potentially promising next step might therefore be to increase the compute budget of our experiments by distributing retro branching across multiple GPUs and CPUs and see whether or not the agent does eventually match or exceed the 500×1000 set covering performance of the IL agent after enough epochs.

B Neural Network

B.1 Architecture

We used the same GCN architecture as Gasse et al. 2019 to parameterise our DQN value function with some minor modifications which we found to be helpful. Firstly, we replaced the ReLU activations with Leaky ReLUs which we inverted in the final readout layer in order to predict the negative Q-values of our MDP. Secondly, we initialised our linear layer weights and biases with a normal distribution ($\mu = 0, \sigma = 0.01$) and all-zeros respectively, and our layer normalisation weights and biases with all-ones and all-zeros

respectively. Thirdly, we removed a network forward pass in the bipartite graph convolution message passing operation which we found to be unhelpfully computationally expensive. For clarity, Figure 6 shows the high-level overview of the neural network architecture. For a full analysis of the benefit of using GCNs for learning to branch, refer to Gasse et al. 2019.

B.2 Inference & Solving Times

The key performance criterion to optimise for any branching method is the reduction of the overall B&B solving time. However, accurate and precise solving time and primal-dual integral over time comparisons are difficult because they are hardware-dependent. This is particularly problematic in research settings where CPU/GPU resources are often shared between multiple researchers and therefore hardware performance (and consequently solving time) significantly varies. Consequently, as in other works (Khalil et al. 2016; Gasse et al. 2019; Etheve et al. 2020), we presented and optimised for the number of B&B tree nodes as this is hardware-independent and, in the context of prior work, can be used to infer the solving time.

Specifically, we use the same GCN-based architecture of Gasse et al. 2019 for all ML branchers, thus all ML approaches have the same per-step inference cost. Therefore the relative difference in the number of tree nodes is exactly the relative wall-clock times on equal hardware. When the per-step inference process is different (as for our non-ML baselines, such as SB), the number of tree nodes is not an adequate proxy for solving time. However, Gasse et al. 2019 have already demonstrated that the GCN-based branching policies of IL outperform the solving time of other branchers such as SB. As this ML speed-up has already been established, in this manuscript we focus on improving the per-step ML decision quality using RL rather than further optimising network architecture, or otherwise, for speed, which we leave to further work.

However, empirical solving times are of interest to the broader optimisation community. Therefore, Table 3 provides a summary of the solving times of the branching agents on the large 500×1000 set covering instances under the assumption that they were ran on the same hardware as Gasse et al. 2019.

Method	Solving time (s)
SB	33.5
IL	2.1
Retro	2.5
FMSTS-DFS	12.2
FMSTS	7.6
Original	35.8

Table 3: Inferred mean solving times of the branching agents on the large 500×1000 set covering instances under the assumption that they were ran on the same hardware as Gasse et al. 2019.

C Data Set Size Analysis

As described in Section 5, we used 100 MILP instances unseen during training to evaluate the performance of each branching agent. This is in line with prior works such as Khalil et al. 2016 who used 84 instances and Gasse et al. 2019 who used 20. To ensure that 100 instances are a large enough data set to reliably compare branching agents, we also ran the agents on 1000 large 500×1000 set covering instances. The relative performance of each branching agent was approximately the same as when evaluated on 100 instances, with Retro scoring 65.3 nodes, FMSTS 250 ($3.8\times$ worse than Retro), IL 55.4 (17.8% better than Retro), and SB 43.3. In the interest of saving evaluation time and hardware demands and to make the development of and comparison to our work by future research projects more accessible, as well as for clarity in the per-instance Retro-IL comparison of Figure 3d, we report the results for 100 evaluation instances in the main paper in the knowledge that the relative performances are unchanged as we scale the data set to a larger size.

D SCIP Parameters

For all non-DFS branching agents we used the same SCIP 2022 B&B parameters as Gasse et al. 2019, as summarised in Table 4.

SCIP Parameter	Value
`separating/maxrounds`	0
`separating/maxroundsroot`	0
`limits/time`	3600

Table 4: Summary of the SCIP 2022 hyperparameters used for all non-DFS branching agents (any parameters not specified were the default SCIP 2022 values).

E Observation Features

We found it useful to add 20 features to the variable nodes in the bipartite graph in addition to the 19 features used by Gasse et al. 2019. These additional features are given in Table 5; their purpose was to help the agent to learn to aggregate over the uncertainty in the future primal-dual bound evolution caused by the partially observable activity occurring in subtrees external to its retrospectively constructed trajectory.

F FMSTS Implementation

Etheve et al. (2020) did not open-source any code, used the paid commercial CPLEX (2009) solver, and experimented with proprietary data sets. Furthermore, they omitted comparisons to any other ML baseline such as Gasse et al. (2019), further limiting their comparability. However, we have done a 'best effort' implementation of the relatively simple FMSTS algorithm, whose core idea is to set the Q-function of a DQN agent as minimising the sub-tree size rooted at the current node and to use a DFS node selection heuristic. To replicate the DFS setting of Etheve et al. (2020) in SCIP (2022), we used the parameters shown in Table 6. We will release the

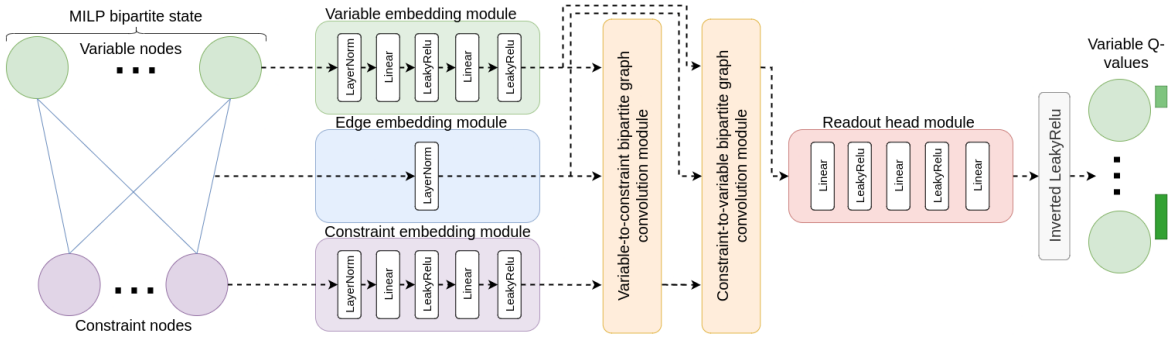


Figure 6: Neural network architecture used to parameterise the Q-value function for our ML agents, taking in a bipartite graph representation of the MILP and outputting the predicted Q-values for each variable in the MILP.

Variable Feature	Description
db_frac_change	Fractional dual bound change
pb_frac_change	Fractional primal bound change
max_db_frac_change	Maximum possible fractional dual change
max_pb_frac_change	Maximum possible fractional primal change
gap_frac	Fraction primal-dual gap
num_leaves_frac	# leaves divided by # nodes
num_feasible_leaves_frac	# feasible leaves divided by # nodes
num_infeasible_leaves_frac	# infeasible leaves divided by # nodes
num_lp_iterations_frac	# nodes divded by # LP iterations
num_siblings_frac	Focus node's # siblings divided by # nodes
is_curr_node_best	If focus node is incumbent
is_curr_node_parent_best	If focus node's parent is incumbent
curr_node_depth	Focus node depth
curr_node_db_rel_init_db	Initial dual divided by focus' dual
curr_node_db_rel_global_db	Global dual divided by focus' dual
is_best_sibling_none	If focus node has a sibling
is_best_sibling_best_node	If focus node's sibling is incumbent
best_sibling_db_rel_init_db	Initial dual divided by sibling's dual
best_sibling_db_rel_global_db	Global dual divided by sibling's dual
best_sibling_db_rel_curr_node_db	Sibling's dual divided by focus' dual

Table 5: Descriptions of the 20 variable features we included in our observation in addition to the 19 features used by Gasse et al. 2019.

full re-implementation to the community along with our own code.

G Pseudocode

Retrospective Trajectory Construction Algorithm 1 shows the proposed 'retrospective trajectory construction' method, whereby fathomed leaf nodes not yet added to a trajectory are selected as the brancher's terminal states and paths to them are iteratively established using some construction method.

Algorithm 1: Retrospectively construct trajectories.

Input: B&B tree $\mathcal{T}$ from solving MILP
Output: Retrospectively constructed trajectories
Initialise: nodes_added, subtree_episodes = [$\mathcal{T}_{\text{root}-1}$], []
// Construct trajectories until all valid node(s) in $\mathcal{T}$ added
while True **do**
  // Root trajectories at highest level unselected node(s)
  subtrees = []
  for node in nodes_added **do**
    for child_node in $\mathcal{T}_{\text{node}}$.children **do**
      if child_node not in nodes_added **then**
        // Use depth-first-search to get sub-tree
        subtrees.append(dfs($\mathcal{T}$, root=child_node))
      end if
    end for
  end for
  // Construct trajectory episode(s) from sub-tree(s)
  if len(subtrees) $>$ 0 **then**
    for subtree in subtrees **do**
      subtree_episode = construct_path(subtree) (2)
      subtree_episode[-1].done = True
      subtree_episodes.append(subtree_episode)
      for node in subtree_episode **do**
        nodes_added.append(node)
      end for
    end for
  else
    // All valid nodes in $\mathcal{T}$ added to a trajectory
    break
  end if
end while

Maximum Leaf LP Gain Algorithm 2 shows the proposed 'maximum leaf LP gain' trajectory construction method, whereby the fathomed leaf node with the greatest change in the dual bound ('LP gain') is used as the terminal state of the trajectory.

H Cost of Strong Branching Labels

As well as performance being limited to that of the expert imitated, IL methods have the additional drawback of requiring an expensive data labelling phase. Figure 7 shows how the explore-then-strong-branch labelling scheme of Gasse et al. 2019 scales with set covering instance size (rows $\times$ columns) and how this becomes a hindrance for larger instances. Although an elaborate infrastructure can be developed to try to label large instances at scale (Nair et al. 2021), ideally the need for this should be avoided; a key motivator for using RL to branch.

Algorithm 2: Maximum leaf LP gain trajectory construction.

Input: Sub-tree $\mathcal{S}$
Output: Trajectory $\mathcal{S}_E$
Initialise: gains = { }
for leaf in $\mathcal{S}$.leaves **do**
  if leaf closed by brancher **then**
    gains.leaf = $|\mathcal{S}_{\text{root}}.\text{dual\_bound} - \mathcal{S}_{\text{leaf}}.\text{dual\_bound}|$
  end if
end for
terminal_node = max(gains)
$\mathcal{S}_E$ = shortest_path(source=$\mathcal{S}_{\text{root}}$, target=terminal_node)

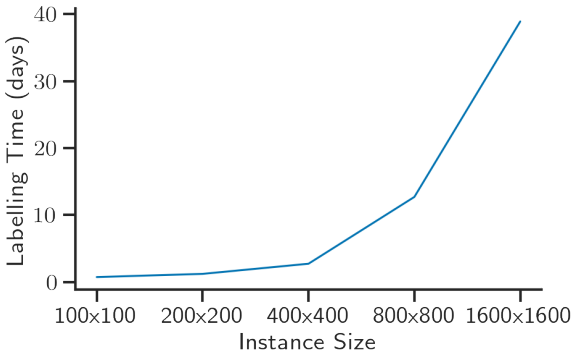


Figure 7: How the explore-then-strong-branch data labelling phase of the strong branching imitation agent scales with set covering instance size (rows $\times$ columns) using an Intel Xeon ES-2660 CPU and assuming 120 000 samples are needed for each set.

SCIP Parameter	Value
separating/maxrounds	0
separating/maxroundsroot	0
limits/time	3600
nodeselection/dfs/stdpriority	1 073 741 823
nodeselection/dfs/memsavepriority	536 870 911
nodeselection/restartdfs/stdpriority	−536 870 912
nodeselection/restartdfs/memsavepriority	−536 870 912
nodeselection/restartdfs/selectbestfreq	0
nodeselection/bfs/stdpriority	−536 870 912
nodeselection/bfs/memsavepriority	−536 870 912
nodeselection/breadthfirst/stdpriority	−536 870 912
nodeselection/breadthfirst/memsavepriority	−536 870 912
nodeselection/estimate/stdpriority	−536 870 912
nodeselection/estimate/memsavepriority	−536 870 912
nodeselection/hybridestim/stdpriority	−536 870 912
nodeselection/hybridestim/memsavepriority	−536 870 912
nodeselection/uct/stdpriority	−536 870 912
nodeselection/uct/memsavepriority	−536 870 912

Table 6: Summary of the SCIP 2022 hyperparameters used the DFS FMSTS branching agent of Etheve et al. 2020 (any parameters not specified were the default SCIP 2022 values).

Reinforcement Learning for Variable Selection in a Branch and Bound Algorithm

Marc Etheve[1,3][0000−0003−4436−3391], Zacharie Alès[2,3][0000−0003−4602−2638], Côme Bissuel[1][0000−0002−5430−3168], Olivier Juan[1][0000−0003−3445−4847], and Safia Kedad-Sidhoum[3][0000−0002−2184−2261]

[1] EDF R&D, France
{marc.etheve, come.bissuel, olivier.juan}@edf.fr
[2] ENSTA Paris, Institut Polytechnique de Paris, France
zacharie.ales@ensta-paris.fr
[3] CNAM Paris, CEDRIC, France
safia.kedad_sidhoum@cnam.fr

Abstract. Mixed integer linear programs are commonly solved by Branch and Bound algorithms. A key factor of the efficiency of the most successful commercial solvers is their fine-tuned heuristics. In this paper, we leverage patterns in real-world instances to learn from scratch a new branching strategy optimised for a given problem and compare it with a commercial solver. We propose FMSTS, a novel Reinforcement Learning approach specifically designed for this task. The strength of our method lies in the consistency between a local value function and a global metric of interest. In addition, we provide insights for adapting known RL techniques to the Branch and Bound setting, and present a new neural network architecture inspired from the literature. To our knowledge, it is the first time Reinforcement Learning has been used to fully optimise the branching strategy. Computational experiments show that our method is appropriate and able to generalise well to new instances.

Keywords: Reinforcement Learning· Mixed Integer Linear Programming · Neural Network · Branch and Bound · Branching Strategy

1 Introduction

Mixed Integer Linear Programming (MILP) is an active field of research due to its tremendous usefulness in real-world applications. The most common method designed to solve MILP problems is the Branch and Bound (B&B) algorithm (see [1] for an exhaustive introduction). B&B is a general purpose procedure dedicated to solve any MILP instance, based on a divide and conquer strategy and driven by generic heuristics and bounding procedures.
Recently, a lot of attention has been paid to the interactions between MILP and machine learning. As pointed out in [2], learning methods may compensate for the lack of mathematical understanding of the B&B method and its variants [3,4]. The plethora of different approaches in this young field of research

gives evidence of the variety of ways in which learning can be leveraged. For instance, a natural idea is to bypass the whole B&B procedure to directly learn solutions of MILP instances [5]. If one wants to preserve the optimality guarantee provided by the B&B algorithm, a solution could be to rather learn the output of a computationally expensive heuristic used in a B&B scheme [6,7,8]. Alternatively, [9] suggests learning to select the best cut among a set of available cuts at each node of the B&B tree. Whether it is by Imitation Learning or by Reinforcement Learning (RL), these solutions are often limited by their scope: they seek to take decisions according to a local criterion.

In the present work, we propose FMSTS (*Fitting for Minimising the SubTree Size*), a novel approach based on Reinforcement Learning aiming at optimising a global criterion at the scale of the whole B&B tree. We learn a branching strategy from scratch, independent of any heuristic.
The paper is structured as follows. First, we define the general setting of our study. Using RL to minimise a global criterion, we then demonstrate that, under certain assumptions, a specific kind of value functions enforces the optimality of such criterion. Next, we propose to adapt known generic learning methods and neural network architectures to the Branch and Bound setting. We illustrate our proposed method on industrial problems and discuss it before concluding.

2 General Setting

In real-world applications, companies often optimise systems on a regular basis given fluctuating data. This case has been studied in the literature for different purposes, such as learning an approximate solution [5] or imitating heuristics [8]. However, to our knowledge, no concrete contribution has been made regarding the use of Reinforcement Learning for variable selection (branching) in this setting. The present work fills this gap.

Throughout this paper, we are interested in the following setting. For a given problem $\mathcal{P}$, the instances are perceived as randomly distributed according to an unknown distribution $\mathcal{D}$. This distribution, emanating from real-world systems, governs the fluctuating data (A, b, c) across instances, written as

$$p \in \mathcal{P} : \begin{cases} \min\limits_{x \in \mathbb{R}^n} c^\top x \\ \quad Ax \leq b \\ s.t. \quad x_{\mathcal{J}} \in \{0,1\}^{|\mathcal{J}|},\ x_{-\mathcal{J}} \in \mathbb{R}^{n-|\mathcal{J}|} \end{cases} \tag{1}$$

with $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$. In practice, as the instances come from a single problem, they share the same structure. Especially, the set of null coefficients, the number of constraints m, of variables n and the set of binary variables $\mathcal{J}$ are the same for every instance of a given problem.

In this setting, we seek to learn and optimise a branching strategy to solve to optimality any instance of a given problem. As pointed out in [10], the problem of optimising the decisions along a B&B tree is naturally formulated as a control problem on a sequential decision-making process. More specifically, it is equivalent to solving a finite-horizon deterministic Markov Decision Process (MDP) and may thus be tackled by Reinforcement Learning (see [11] for an introduction).

3 Fitting for Minimising the SubTree Size (FMSTS)

In a finite-horizon setting, Reinforcement Learning aims at optimising an agent to produce sequences of actions that achieve a global objective. The agent is guided by local costs associated with the actions it takes. Starting from an initial state in a possibly stochastic environment, it must learn to take the best sequence of actions and thus transitioning from state to state to minimise the overall costs. Exact Q-learning solves such problem by updating a table mapping (Q-function) from state/action pairs to discounted future costs (Q-values).
In the following, we define the RL problem of interest and, as exact Q-learning is not practicable, the approximate framework considered. Next, we propose a specific informative Q-function allowing us to use this framework in practice.

3.1 Approximate Q-learning with Observable Q-values

Let us denote by $\mathcal{S}$ the set of every reachable state for a given problem $\mathcal{P}$, a state being defined as all the information available when taking a branching decision in a B&B tree. Under perfect information, a state associated to a B&B node is the whole B&B tree as it has been expanded at the time the branching decision is taken. We write $\mathcal{A}$ the set of actions, *i.e.* the set of available branching decisions on a specific problem (the set of binary variables $\mathcal{A} = \mathcal{J}$ in our case) and π a policy mapping any state to a branching decision:

$$\pi \ : \ \begin{cases} \mathcal{S} \to \mathcal{A} \\ s \mapsto \pi(s) = a \end{cases}.$$

The transitions between states are governed by the B&B solver, and the MDP is regarded as deterministic: given an instance and a state, performing an action will always lead to the same next state. In practice, such assumption is met as soon as the solver's decisions (apart from branching) are non stochastic.
We call agent any generator Π of branching sequences following policy π and denote $\Pi(p)$ the sequence of decisions that maps an instance p to a complete B&B tree. Let $\mu\left(\Pi(p)\right)$ be any metric of interest on the tree generated by Π for an instance p, and assume this metric is to be minimised. For instance, we can think of μ as the size of the generated tree, the number of simplex iterations, etc. In this setting, we are looking for the $\mu-$optimal agent Π^* such that

$$\Pi^* \in \arg\min_{\Pi} \mathbb{E}_{p\sim\mathcal{D}}\left[\mu\left(\Pi\left(p\right)\right)\right]. \tag{2}$$

Note here that the expectation is only on the MILP instances, as the MDP is deterministic.

Let us assume that one can define a Q-function Q^π which is consistent with μ, in the sense that μ is minimised if $\pi(s) = \arg\min_{a \in \mathcal{A}} Q^\pi(s, a)$. Even in this case, exact Q-learning cannot be used to minimise μ. First, maintaining an exact table for the Q-function is not tractable due to the size of $\mathcal{S}$, including for small real-world problems. Second, the transition from a state to the next is too complex to be modeled since it partly results from a linear optimisation.
To bypass these problems, we approximate the Q-function (see [11] for an introduction to Approximate Q-learning) by a neural network $\hat{Q}$ parametrised by θ and optimised by a dedicated gradient method as in the DQN (Deep Q-Network) approach [12]. We define the policy π_θ resulting from the Q-network as $\pi_\theta(s) = \arg\min_{a \in \mathcal{A}} \hat{Q}(s, a; \theta)$.
When facing a deterministic MDP, the exact Q-value $Q^{\pi_\theta}(s, a)$ of an action a given a state s and a policy π_θ is not stochastic and thus may be observable. In that case, the classic Temporal Difference loss used in [12] for training the Q-Network comes down to the simple expression

$$L(\theta) = \mathbb{E}_{s,a \sim \rho(.)} \left[\left(Q^{\pi_\theta}(s, a) - \hat{Q}(s, a; \theta) \right)^2 \right] \tag{3}$$

where ρ is the behaviour distribution of our agent, as stated in [12]. Note that, in Equation (3), the observed Q-values are naturally influenced by parameter θ through the policy. Such loss is actually intuitive: if Q^{π_θ} is consistent with μ, if each action has non-zero probability to be taken in any encountered state and if $L(\theta) = 0$, then each B&B tree built by agent Π_θ (following π_θ) is optimal with respect to μ with probability 1.

3.2 Using the Subtree Size as Value Function

As highlighted in [8], an important difficulty when applying Reinforcement Learning to B&B algorithms is the credit assignment problem [13]: in order to determine the actions that lead to a specific outcome, one may define non-sparse informative local costs (negative rewards) consistent with the global objective. This is not mandatory in a RL setting but may facilitate the learning task.

We choose the number of nodes in the generated tree as the global metric μ. This metric is often used to compare B&B methods (see for instance [6,7,8]), as it is a proxy for computational efficiency and independent of hardware considerations.
One of the main contributions of this paper is to propose a local Q-function Q^π which is consistent with the chosen global metric μ. We take advantage of the deterministic aspect of the environment and define $Q^\pi(s, a)$ as the size of the subtree rooted in the B&B node corresponding to s generated by action a and policy π. As stated in Proposition 1, this particular Q-function is not consistent

in general with our choice of global criterion μ. Nonetheless, Proposition 2 asserts its optimality when using Depth First Search as node selection strategy.

Proposition 1. *In general, minimising the size of the subtree under any node in a B&B tree is not optimal with respect to the tree size.*

The proof is omitted for the sake of conciseness, but one can prove that minimising the subtree size can be sub-optimal when using Breadth First Search as the node selection strategy.

Proposition 2. *When using Depth First Search (DFS) as the node selection strategy, minimising the whole B&B tree size is achieved when any subtree is of minimal size.*

Proof. Let us call $\mathcal{O}$ the set of open nodes at a given iteration of the B&B process for a specific instance. The set of closed nodes (either by pruning or branching) is denoted $\mathcal{C}$.
We write $V^{\pi}(s\,|\zeta,\eta)$ the size of the subtree under s, entirely determined by the policy π followed in this subtree, a set of primal bounds ζ found in other subtrees and a node selection strategy η. When using DFS, the subtrees under each open node are expanded and fully solved sequentially, thus we can assume with no loss of generality that $\mathcal{O}$ is equal to $\{1,...,k\}$ and is sorted according to the planned visiting order. In that case, the size V of the whole B&B tree can be expressed as

$$V = |\mathcal{C}| + \sum_{i=1}^{k} V^{\pi_i}\left(s_i \;\middle|\; \{z_0\} \cup \left(\bigcup_{j<i} \zeta_j\right),\ \eta = DFS\right)$$

with ζ_i the set of all bounds to be found in the subtree rooted in s_i and z_0 the best bound obtained in $\mathcal{C}$.
It remains to prove that π_1 is optimal only if it leads to the minimal subtree under s_1. As two separate subtrees can only affect each other through their best primal bound under DFS, we have

$$V = |\mathcal{C}| + V^{\pi_1}(s_1\ |\{z_0\},\ \eta = DFS) + \sum_{i=2}^{k} V^{\pi_i}(s_i\ |\{z_{i-1}\},\ \eta = DFS)$$

with $z_{i-1} = \min\left\{\{z_0\} \cup \left(\bigcup_{j<i} \zeta_j\right)\right\}$.
Since the B&B procedure (with a gap set to zero) guarantees that we find the best primal bound of any expanded subtree, $(z_i)_{i=1}^{k}$ are completely independent of the branching policies, which gives, for any $\pi_j,\ j \in \{2,...,k\}$:

$$\underset{\pi_1 \in \Pi_1}{\arg\min}\, V = \underset{\pi_1 \in \Pi_1}{\arg\min}\, V^{\pi_1}(s_1\ |\{z_0\},\ \eta = DFS)$$

with Π_1 the set of all valid branching policies under s_1. Therefore, choosing any other policy than $\pi_1 \in \arg\min_{\pi_1 \in \Pi_1} V^{\pi}(s_1\ |\{z_0\},\ \eta = DFS)$ is sub-optimal

with respect to the tree size.

□

In the remaining, we use DFS as the node selection strategy according to Proposition 2. We now focus on optimising the branching strategy (variable selection) to minimise at each node the size of the underlying subtree. If we write $D_0^{\pi(s)}(s)$ and $D_1^{\pi(s)}(s)$ the child nodes of s following policy π, such value function satisfies the Bellman Equation (4). The relationship between the value and the Q-function is trivially defined by $Q^\pi(s,a) = 1 + V^\pi(D_0^a(s)) + V^\pi(D_1^a(s))$.

$$V^\pi(s) = 1 + V^\pi\left(D_0^{\pi(s)}(s)\right) + V^\pi\left(D_1^{\pi(s)}(s)\right) \tag{4}$$

This value function has two advantages. First, it is observable as assumed earlier: we only need to count the number of inheriting nodes once the B&B tree is fully expanded. Second, it is a local objective which guarantees the optimality of a global criterion, hence allowing us to perform RL without designing a sub-optimal reward using any domain knowledge.

3.3 Algorithm

Using Approximate Q-learning and the subtree size as value function leads us to propose the FMSTS algorithm (Algorithm 1). Using Experience Replay and ε-greedy exploration as in [12], the algorithm essentially boils down to consecutively solving a MILP instance following the current policy or random choices with probability ε, fitting the observed values sampled from an experience replay buffer and iterating with the updated policy.

Algorithm 1 FMSTS

for t = 0,...,N-1 **do**
  Draw randomly an instance p.
  Solve p following π_{θ_t} with ε-greedy exploration.
  Collect experiences along the generated tree $\left(s_i, a_i, Q^{\pi_{\theta_t}}(s,a), \hat{Q}(s,a;\theta_t)\right)$ and store them into an experience replay buffer $\mathcal{B}$.
  Update to θ_{t+1} using loss (3) on experiences drawn from $\mathcal{B}$.
end for

4 Adapting Learning to the Branch and Bound Setting

To ensure the success of the FMSTS method (Algorithm 1) with respect to the objective (2), we need to adapt some components to the Branch and Bound setting. First, we adapt the loss guiding the neural network's training. Next, we use Prioritized Experience Replay while normalising probabilities. Last, we propose a new neural network architecture inspired from the literature.

4.1 Minimising an Expectation on the Instance Distribution

The loss defined by Equation (3) does not seem to correspond to our objective (2). Indeed, it naturally gives more importance to the biggest trees, which can be heavily instance dependent. To neutralise this effect, we weight the loss by the inverse of the size of the corresponding B&B tree generated by the agent:

$$L(\theta) = \mathbb{E}_{s,a\sim\rho(.)}\left[\frac{1}{V^{\pi_\theta}\left(r(s)\right)}\left(Q^{\pi_\theta}(s,a) - \hat{Q}(s,a;\theta)\right)^2\right] \tag{5}$$

with $r(s)$ the root node of the tree containing s, such that $V^{\pi_\theta}\left(r(s)\right)$ corresponds to the size of this tree. Then, any instance has equal weight in loss (5).

4.2 Performing Prioritized Experience Replay

Prioritized Experience Replay [14] biases the uniform replay sampling of Experience Replay [12] towards experiences with high Temporal Difference errors, *i.e.* when the predicted Q-values are far from their target. In FMSTS, an experience is a 4-tuple $\left(s_j, a_j, Q^{\pi_{\theta_j}}(s_j,a_j), \hat{Q}(s_j,a_j;\theta_j)\right)$ and the target $Q^{\pi_{\theta_j}}(s_j,a_j)$ is observed, which reduces the error to the simple expression $\left|Q^{\pi_{\theta_j}}(s_j,a_j) - \hat{Q}(s_j,a_j;\theta_j)\right|$. In the context of sampling experiences in a B&B tree, one should take into account that the scale of the target $Q^{\pi_{\theta_j}}$ may vary exponentially both along the tree and across instances. As the scale of the error may likely vary with that of the target, we normalise this error by the target to get the sampling probability in the experience replay buffer

$$p_j \propto \frac{\left|Q^{\pi_{\theta_j}}(s_j,a_j) - \hat{Q}(s_j,a_j;\theta_j)\right|}{Q^{\pi_{\theta_j}}(s_j,a_j)}. \tag{6}$$

4.3 Designing a Regressor for the Q-function

As in [6], we use both static and dynamic features to represent a state. Although many features may be relevant for the states' encoding, we opted to keep them limited in the present work. For static information, we perform a dimension reduction by PCA [15]: each instance is represented as the concatenation of its data (A, b, c) and PCA is applied on the resulting vectors. Our representation also includes the following dynamic features: the node's depth, the distance of the current primal solution to the bounds and the branching state. Concretely, the branching state B is one-hot encoded in a $3\,|\mathcal{J}|$ vector. Let us call $\mathcal{B}_0$ and $\mathcal{B}_1$ the set of variables that have been respectively set to 0 and 1 in the ascendant nodes of the current state. With no loss of generality, let us assume that $\mathcal{J} = \{1, ..., J\}$. Then we have $B_j = \mathbb{1}_{x_j\in\mathcal{B}_0}$, $B_{j+J} = \mathbb{1}_{x_j\in\mathcal{B}_1}$ and $B_{j+2J} = 1 - B_j - B_{j+J}$ for any $j \in \mathcal{J}$.
The chosen Q-function is essentially multiplicative, in the sense that the ratio between the targets in two consecutive states may be of magnitude 2 due to the

binary tree structure. In addition, the scale of these targets may strongly vary between instances. A basic feedforward neural network, based on summations, may struggle to handle such phenomena. To compensate for these effects and adapt to the B&B setting, we take inspiration from the Dueling architecture of [16] and propose the Multiplicative Dueling Architecture (MDA). As shown in Figure 1, MDA implements the product between the 1-D output of a block of fully-connected layers fed with static features and the $|\mathcal{J}|$-D output of a block fed with both static and dynamic features. A linear activation on the 1-D output allows our agent to capture the variability of the chosen Q-function.

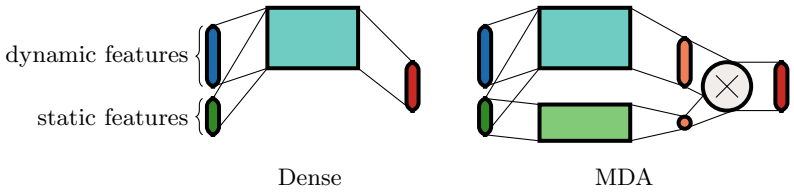


Fig. 1. Dense and Multiplicative Dueling architectures for the Q-network. The rectangles represent consecutive dense layers, the lightblue block being fed with all the features whereas the lightgreen one is only fed with static features (darkgreen). The output of the MDA is the product between a single unit and a $|\mathcal{J}|$-unit dense layer.

5 Experiments and discussion

We test our algorithms on two sets of instances provided by Electricité de France (EDF), a french electric utility company. They are drawn from two different problems, one is related to energy management in a microgrid ($\mathcal{P}_1$) whereas the other one comes from a hydroelectric valley ($\mathcal{P}_2$). The problems have respectively 186 and 282 constraints, 120 and 207 variables, and 54 and 96 binary variables.
We compare our algorithms to the default branching strategy of CPLEX (denoted CPLEX in the following) and full Strong Branching (denoted SB). We use CPLEX 12.7.1 [17] under DFS while turning off all presolving and cutting.
To avoid any dependency of our results to the train or the test set, we present cross-validated results. Algorithm 1 is run 100 times independently on randomly partitioned train and test sets. Each time, 200 instances are used for training while 500 unseen instances are used for testing. Figure 2 shows the averaged number of nodes in the complete B&B trees on the test sets during the learning process: test instances are solved using the strategy learned on train instances at the current iteration of Algorithm 1.

As exhibited in Figure 2, our method is able to learn an efficient strategy from scratch. As expected, the MDA agent is more flexible than its additive counterpart (Dueling) inspired from the Dueling architecture of [16] and the fully-connected agent (Dense). It outperforms systematically the Strong Branching policy, and finds comparable or better strategies than CPLEX, depending

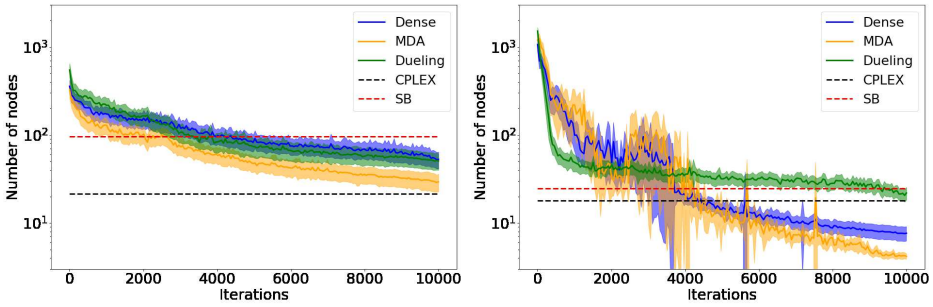


Fig. 2. Cross-validated performance on test instances (averaged number of nodes in log scale) for $\mathcal{P}_1$ (left) and $\mathcal{P}_2$ (right) through iterations of Algorithm 1. Gaussian confidence intervals are shown around the means.

on the problem. Results on training data are not displayed for the sake of conciseness, but it is worth mentioning that our agents do not overfit and are able to generalise well. In addition, the computation time is negligible compared with full Strong Branching as an action comes only at the price of a forward pass in our neural network.

Despite these good performances, some limits have to be pointed out at this stage. First, our framework requires DFS as the node selection strategy, which can be far from optimal for certain problems. Note that using another strategy may be complicated to handle due to more complex dependencies, but may also turn out to be effective as targetting small subtrees makes sense in general. Second, we only showed promising results on easy problems. With more difficult problems, the training becomes computationally prohibitive as a randomly initialised agent produces exponential trees. To tackle these limitations, we encompass different solutions such as fine-tuning the features and network architecture or using supervision to decrease the size of the generated trees during the first episodes. To reduce the cost of exploration, one could apply the same methodology with a set of branching heuristics as action set, similarly to what is proposed in [18].

6 Conclusion

In this paper, we presented a novel Reinforcement Learning framework designed to learn from scratch the branching strategy in a B&B algorithm. In addition to the specific metrics used in our FMSTS method, we introduced a new neural network architecture designed to tackle the multiplicative nature of the value function. Besides, we adapted some known RL techniques to the B&B setting. We ran experiments on real-world problems to validate our method and showed better or comparable performances with existing strategies.

It is worthwhile to highlight that our method is generic enough to be applied to other metrics than the tree size, e.g. the number of simplex iterations or even the

computation time. If one is not interested in proving optimality, many other value functions may be encompassed. Furthermore, it may be interesting to enlarge the scope of the method, especially to include Branch and Cut algorithms as they usually are more efficient.

References

1. L. A. Wolsey, *Integer programming*. Wiley, 1998.
2. Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *arXiv preprint arXiv:1811.06128*, 2018.
3. C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. Savelsbergh, and P. H. Vance, "Branch-and-price: Column generation for solving huge integer programs," *Operations research*, vol. 46, no. 3, pp. 316–329, 1998.
4. J. E. Mitchell, "Branch-and-cut algorithms for combinatorial optimization problems," *Handbook of applied optimization*, vol. 1, pp. 65–77, 2002.
5. E. Rachelson, A. B. Abbes, and S. Diemer, "Combining mixed integer programming and supervised learning for fast re-planning," in *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, vol. 2, pp. 63–70, IEEE, 2010.
6. E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
7. M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik, "Learning to branch," *arXiv preprint arXiv:1803.10150*, 2018.
8. M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi, "Exact combinatorial optimization with graph convolutional neural networks," *arXiv preprint arXiv:1906.01629*, 2019.
9. Y. Tang, S. Agrawal, and Y. Faenza, "Reinforcement learning for integer programming: Learning to cut," *arXiv preprint arXiv:1906.04859*, 2019.
10. H. He, H. Daume III, and J. M. Eisner, "Learning to search in branch and bound algorithms," in *Advances in Neural Information Processing Systems 27* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 3293–3301, Curran Associates, Inc., 2014.
11. R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
12. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.
13. M. Minsky, "Steps toward artificial intelligence," *Proceedings of the IRE*, vol. 49, no. 1, pp. 8–30, 1961.
14. T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.
15. K. Pearson, "LIII. On lines and planes of closest fit to systems of points in space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901.
16. Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.
17. C. U. Manual, *IBM ILOG CPLEX Optimization Studio*, 1987.
18. G. Di Liberto, S. Kadioglu, K. Leo, and Y. Malitsky, "Dash: Dynamic approach for switching heuristics," *European Journal of Operational Research*, vol. 248, no. 3, pp. 943–953, 2016.