Neural Large Neighborhood Search

Ravichandra Addanki
MIT
addanki@mit.edu

Vinod Nair
DeepMind
vinair@google.com

Mohammad Alizadeh
MIT
alizadeh@csail.mit.edu

1 Introduction

Large Neighborhood Search (LNS) [43, 39] is a powerful local search paradigm for solving combinatorial optimization problems. LNS methods have been successful in many challenging problems, including Mixed Integer Programs (MIPs) [17, 40, 13, 20], Traveling Salesman Problem (TSP) [44], Vehicle Routing Problem (VRP) [43, 27], and Constraint Programming [38, 15]. Like other local search methods, LNS starts with a feasible assignment of the variables being optimized. At each iteration, it searches in a neighborhood of the current assignment, and if it finds an assignment with a better objective value it updates the current assignment. It repeats this procedure until the search budget is exhausted. In LNS, the number of neighboring assignments at each iteration is too large for exhaustive search (e.g., exponential in the number of variables).

The choice of the search neighborhood at each iteration is crucial for LNS to be effective. It should 1) contain assignments that allow large objective value improvements in a few LNS iterations, and 2) be computationally tractable to search. Designing a neighborhood selection policy that jointly considers these two requirements is challenging. Successful applications of LNS rely on a domain expert to consider the problem structure and the cost of neighborhood search to time-consumingly handcraft a problem-specific policy [39]. For example, in the case of MIPs [49], an LNS algorithm called RINS [17] at each iteration un-assigns the values of the subset of variables whose linear relaxation solution disagrees with the current assignment. This results in a smaller MIP that can be efficiently solved using an off-the-shelf MIP solver to find a better assignment.

We propose a Reinforcement Learning (RL) [46] approach to learn a neighborhood selection policy for LNS. Given a problem instance and a feasible assignment, a learnable policy defines a search neighborhood around the current assignment. The resulting search problem has the same form as the original one but is significantly smaller, so even an existing (non-learned) solver can be effective on it. The RL environment runs such a solver to find a new assignment. The reward is a function of both the improvement in objective as well as the computational effort for the neighborhood search. We train the policy on problem instances drawn from a specific domain. This allows automatically learning domain-specific neighborhood selection policies that can outperform generic policies.

To demonstrate our approach, we focus specifically on mixed integer programming (MIP), an NP-complete combinatorial optimization problem. See Figure 1. The reward function is defined as the *primal integral* [14] (section 2.1), which combines both solution quality and search effort into a scalar metric. We express the neighborhood selection policy as a graph convolutional neural network (GCNN) [9] that operates on a bipartite graph representation of the MIP.

Related Work Relaxation Induced Neighborhood Search (RINS) [17] is a well-known large neighborhood search meta-heuristic for improving a given feasible MIP solution. It compares the feasible solution to the solution obtained by relaxing the integer variables, and destroys variables whose values differ between the two. The resulting sub-MIP is then solved using a MIP solver. Adaptive LNS [26] uses an ensemble of LNS algorithms for MIPs with a multi-armed bandit to adaptively switch among them during a MIP solve. While we do not focus on an ensemble approach, our work can be used as another ensemble member to improve performance.

Two other works apply learning to large neighborhood search. Hottung *et al.* [28] learn `Repair` using RL for the capacitated vehicle routing problem. This is complementary to our approach and can be

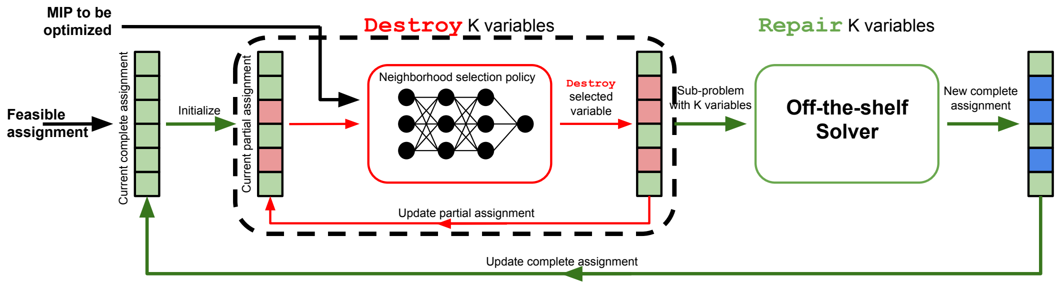

MIP to be optimized
Destroy K variables
Repair K variables
Feasible assignment
Current complete assignment
Initialize
Current partial assignment
Neighborhood selection policy
Destroy selected variable
Sub-problem with K variables
Off-the-shelf Solver
New complete assignment
Update partial assignment
Update complete assignment

Figure 1: Overview of how a learned policy is used in large neighborhood search (LNS) to optimize a mixed integer program (MIP). The inputs are a MIP and a feasible assignment. Adopting terminology from [39], the learned neighborhood selection policy sequentially "destroys" (un-assigns) K variables (indicated by red boxes, with $K = 3$) by selecting one assigned variable (indicated by green boxes) at a time. The resulting sub-MIP with K variables is solved with an off-the-shelf MIP solver to "repair" them (i.e., assign new values, indicated by blue boxes). This is repeated, with the policy selecting a potentially different set of K variables each time, until the search budget is exhausted. A good policy should produce a complete assignment with much better objective value than the initial input assignment at low computational cost.

combined. We focus on learning `Destroy` because the sub-problem in `Repair` can be made small enough for an existing solver to work well. Concurrent with our work, Song *et al.* [45] propose to learn `Destroy` using imitation learning and RL for MIPs. There are two key differences from our work. First, their `Destroy` policy partitions the variables into disjoint subsets and iteratively selects one subset at a time. We do not impose such a restriction. Second, their per-step reward is the change in objective value after each `Repair` step, which ignores its computational expense. We use a reward that explicitly takes into account the computational effort for `Repair`.

Several works apply learning to solving MIPs, such as for branching [25, 31, 5, 19, 51, 50], adding constraints [47], and primal heuristics [32]. These are complementary approaches that aim to improve different components of a MIP solver and can be combined with ours. More generally, our work is an instance of learning for combinatorial optimization - see Bengio *et al.* [10] for a recent survey.

2 Approach

2.1 Background: Integer Programming

An Integer Program is defined as $\min_x\{d^T x \mid Ax \leq b, \quad x_i \in \mathbb{Z} \;\; \forall i\}$, where $x \in \mathbb{Z}^n$ is an n-dimensional integer vector being optimized, $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$ define linear constraints on x, and $d \in \mathbb{R}^n$ defines the linear objective function. A Mixed Integer Program admits both continuous and integer variables. We consider an Integer Program here to simplify notation, but our approach applies equally to MIPs as well. Removing integrality constraints $\{x_i \in \mathbb{Z}\}$ results in a Linear Program (LP).

2.2 MDP Formulation

We consider a contextual MDP [3, 23] $\mathcal{M}_c$ parameterized with respect to a *context* c. Here we define c to be the parameters of an Integer Program, i.e., $c = \{A, b, d\}$. We assume that it is computationally easy to compute a feasible assignment given a MIP. Although theoretically an NP-hard problem, finding a feasible solution is often much easier in practice than finding the optimal solution for many MIPs (*e.g.,* Minimum set cover problem [48]). Figure 1 illustrates how the learned neighborhood selection policy is applied to a MIP. For a given $\mathcal{M}_c$, at each step of an episode the policy selects one of the variables to un-assign (`Destroy`) from its current value. At every K-th step (where K is a hyper-parameter), an integer program containing only the K un-assigned variables is defined. The remaining variables are treated as constants with their values fixed to that in the current assignment. The resulting integer program (called sub-MIP) is then solved using an existing MIP solver to assign values to the K destroyed variables. This is continued until a stopping criterion is satisfied or the maximum episode length is reached.

Reward function: We define the reward based on the *primal integral* [14]. It is normalized to be in $[0, 1]$ and does not depend on the magnitude of the problem parameters. It also accounts for how quickly an optimal solution is reached. Computing the primal integral requires knowing the optimal objective value, but since the reward is needed only during training, it is possible to pre-compute the optimal objective value for each training problem offline. A lower primal integral value corresponds to a better optimizer, so the reward is defined as its negative.

2.3 Policy Network Architecture

Given a MIP instance, we construct an undirected bipartite graph $G = (V, C, E, \mathcal{G})$ using the Constraint-Variable Incidence Graph (CVIG) model [6]. In the CVIG model, vertices correspond to either variables or constraints, and edges correspond to the occurrence of a variable in a constraint with a non-zero coefficient. We use Graph Convolutional Neural Networks (GCNNs) to parameterize our policy and value networks [9, 22, 42, 24, 34]. Due to their permutation invariance and generalization capabilities, GCNNs are a natural choice for learning representations over graphs and have been applied in successfully solving a wide range of problems with graph-structured data [24, 34, 41, 8, 21]. Our GCNN network takes the state representation, $s_t = (V, C, E, \mathcal{G})$ as input and iteratively computes embeddings for the variable and the constraint nodes. Details about the policy architecture can be found in the Appendix.

2.4 Learning Method

For training the policy, we use the *V-trace* algorithm from Espeholt *et al.* [18]. *V-trace* is an off-policy actor-critic algorithm designed for multi-task training at a massive scale. Our training framework consists of actors for policy evaluation and a learner for policy optimization similar to [18]. Actors collect trajectories using randomly sampled MIPs from the training dataset. The collected trajectories are streamed to an experience replay buffer [36], and are sampled uniformly by the learner for training. The update rules used for training are detailed in [18] and reproduced for completeness in the Appendix.

3 Experiments

3.1 Datasets

We evaluate our approach against a wide variety of challenging benchmark MIP datasets from the literature [19, 29, 30, 33, 35, 16, 11, 7]. We use data sets of MIPs of four different kinds labeled *cauction*, *facilities*, *indset*, *setcover*. Details about these data sets can be found in the Appendix.

3.2 Baselines

We use the following heuristics as baselines for comparison against our approach. **Random**: A policy that selects K integer variables to `Destroy` by sampling from a uniform random distribution. **Least-Integral**: Optimize the relaxation of each integer variable one at a time while setting the remaining integer variables to their corresponding values in the current solution. Select the top-K integer variables with relaxed solutions *furthest* away from the current solution. Any ties for the top-K are broken randomly, similar to randomized rounding heuristics for solving MIPs [12]. **Most-Integral**: Identical to the Least-Integral heuristic above, except we select the top-K variables with the relaxed solutions *closest* to the incumbent solution. **RINS**: We modify the Relaxation Induced Neighborhood Search (RINS) heuristic proposed by *Danna et al.* [17] to a local move policy so that it is applicable outside of the branch-and-bound framework for which it was originally designed. In our adaptation of RINS, we compare the current solution with the optimal solution of the LP relaxation and fix those integer variables that take the same value in both. Among the remaining variables, we choose at most K integer variables to `Destroy` using the random policy from above.

3.3 Training Details

To evaluate the generalizability of NLNS, we train a separate policy for each of the datasets and use the trained policy for prediction during LNS on unseen samples from the test set. For each individual training session, we tune the hyper-parameters K and the learning rate to maximize performance on the validation set. See Appendix for a complete list of hyper-parameters and values used for training.

4 Results

In this section, we present the results of our evaluation. We use the primal gap and the primal integral (Section 2.1) as our evaluation metrics.

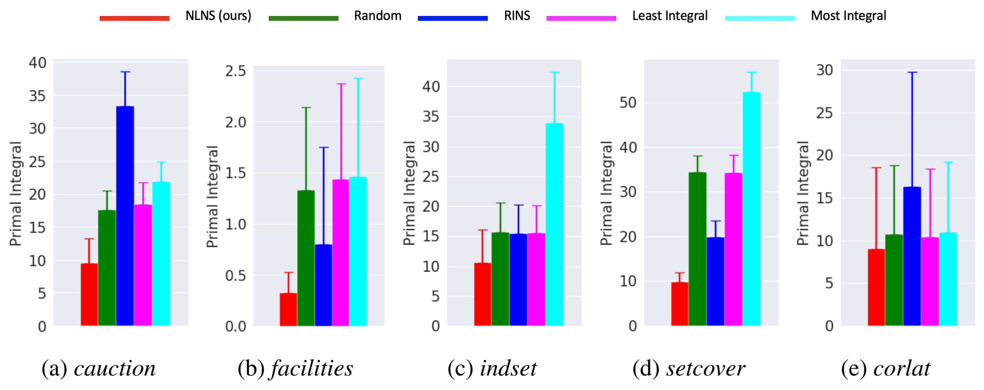


Figure 2: Average Primal Integral achieved by NLNS as a standalone LNS algorithm compared with the baselines. Lower gaps reflect better performance.

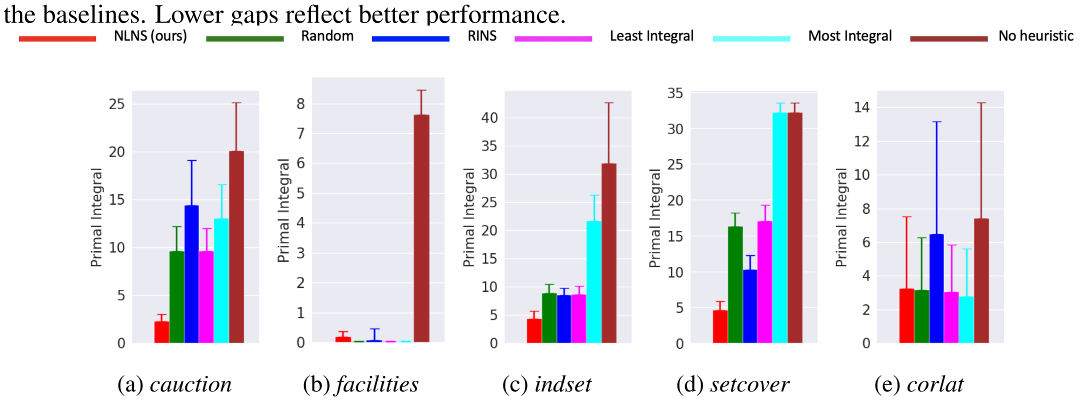


Figure 3: Average Primal Integral of the branch-and-bound with different primal heuristics. Lower integral values reflect better performance.

Standalone Evaluation. Figure 2 shows the average primal integral achieved by NLNS on test problem instances for each dataset. NLNS is at least on-par and in some cases substantially better than the baselines across all datasets. On average, NLNS achieves a relative improvement of 2.1$\times$ over the best baseline across all datasets, with a maximum improvement of 4.5$\times$ on the *facilities* dataset.

Evaluation as a primal heuristic. LNS is typically used in solving MIPs as a primal improvement heuristic [12, 17, 13]. Primal heuristics run alongside the branch-and-bound algorithm to quickly find feasible solutions of high quality. Figure 3 shows the average primal integral of the branch-and-bound algorithm with NLNS on the test datasets. NLNS achieves an average relative improvement of 1.9$\times$ over the best baseline across all datasets, and a maximum improvement of 4.2$\times$ on the *cauction* dataset. Importantly, the best baseline changes for different datasets, while NLNS consistently provides the strong performance because it is able to tailor its neighborhood selection policy to each dataset.

5 Conclusion

We presented NLNS, a reinforcement learning approach to automatically learning problem-specific neighborhood selection polices for LNS. NLNS selects neighborhoods that guide an existing solver to high-quality assignments efficiently. Future work could enable the policy to dynamically select the size of the neighborhood K. We will open-source our code, datasets, and the complete list of hyper-parameters used to reproduce our experiments once the paper is published.

References

[1] Nvidia v100 tensor core gpu. `https://www.nvidia.com/en-us/data-center/v100/`. [Accessed: 04-Jun-2020].

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Y. Abbasi-Yadkori and G. Neu. Online learning in mdps with side information. *arXiv preprint arXiv:1406.6812*, 2014.

[4] T. Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[5] A. Alvarez, Q. Louveaux, and L. Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29:185–195, 01 2017.

[6] C. Ansótegui, J. Giráldez-Cru, and J. Levy. The community structure of sat formulas. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 410–423. Springer, 2012.

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

[8] P. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in neural information processing systems*, pages 4502–4510, 2016.

[9] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[10] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon, 2018.

[11] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[12] T. Berthold. Primal heuristics for mixed integer programs. 2006.

[13] T. Berthold. Rens-relaxation enforced neighborhood search. 2007.

[14] T. Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41:611–614, 11 2013.

[15] T. Berthold, S. Heinz, M. Pfetsch, and S. Vigerske. Large neighborhood search beyond mip. 2012.

[16] G. Cornuéjols, R. Sridharan, and J.-M. Thizy. A comparison of heuristics and relaxations for the capacitated plant location problem. *European journal of operational research*, 50(3):280–297, 1991.

[17] E. Danna, E. Rothberg, and C. L. Pape. Exploring relaxation induced neighborhoods to improve mip solutions. *Mathematical Programming*, 102(1):71–90, Jan 2005.

[18] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

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

[20] S. Ghosh. Dins, a mip improvement heuristic. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 310–323. Springer, 2007.

[21] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.

[22] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734. IEEE, 2005.

[23] A. Hallak, D. Di Castro, and S. Mannor. Contextual markov decision processes. *arXiv preprint arXiv:1502.02259*, 2015.

[24] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[25] H. He, H. Daume III, and J. 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.

[26] G. Hendel. Adaptive large neighborhood search for mixed integer programming. 2018.

[27] H. Hojabri, M. Gendreau, J.-Y. Potvin, and L.-M. Rousseau. Large neighborhood search with constraint programming for a vehicle routing problem with synchronization constraints. *Computers & Operations Research*, 92:87–97, 2018.

[28] A. Hottung and K. Tierney. Neural large neighborhood search for the capacitated vehicle routing problem. *arXiv preprint arXiv:1911.09539*, 2019.

[29] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 186–202. Springer, 2010.

[30] F. Hutter, M. López-Ibáñez, C. Fawcett, M. Lindauer, H. H. Hoos, K. Leyton-Brown, and T. Stützle. Aclib: A benchmark library for algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 36–40. Springer, 2014.

[31] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In D. Schuurmans and M. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 724–731, United States of America, 2016. Association for the Advancement of Artificial Intelligence (AAAI). AAAI Conference on Artificial Intelligence 2016, AAAI 16 ; Conference date: 12-02-2016 Through 17-02-2016.

[32] E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao. Learning to run heuristics in tree search. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 659–666, 2017.

[33] 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.

[34] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

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

[36] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[37] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[38] L. Perron, P. Shaw, and V. Furnon. Propagation guided large neighborhood search. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, CP'04, page 468–481, Berlin, Heidelberg, 2004. Springer-Verlag.

[39] D. Pisinger and S. Ropke. Large neighborhood search. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, pages 399–419, Boston, MA, 2010.

[40] E. Rothberg. An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing*, 19(4):534–541, 2007.

[41] A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, J. Merel, M. Riedmiller, R. Hadsell, and P. Battaglia. Graph networks as learnable physics engines for inference and control. *arXiv preprint arXiv:1806.01242*, 2018.

[42] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[43] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pages 417–431. Springer, 1998.

[44] S. L. Smith and F. Imeson. Glns: An effective large neighborhood search heuristic for the generalized traveling salesman problem. *Computers & Operations Research*, 87:1–19, 2017.

[45] J. Song, R. Lanka, Y. Yue, and B. Dilkina. A general large neighborhood search framework for solving integer programs. *arXiv preprint arXiv:2004.00422*, 2020.

[46] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[47] Y. Tang, S. Agrawal, and Y. Faenza. Reinforcement learning for integer programming: Learning to cut, 2019.

[48] Wikipedia contributors. Set cover problem — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Set_cover_problem&oldid=944382146`, 2020. [Online; accessed 1-June-2020].

[49] L. Wolsey. *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 1998.

[50] Y. Yang, N. Boland, B. Dilkina, and M. Savelsbergh. Learning generalized strong branching for set covering, set packing, and 0-1 knapsack problems. 2020.

[51] Y. Yang, N. Boland, M. Savelsbergh, and H. Stewart. Multi-variable branching: A case study with 0-1 knapsack problems. 10 2019.

Appendix

Primal Gap

Let us denote an Integer Program as

$$\min_x \{d^T x \mid Ax \leq b,\ \ x_i \in \mathbb{Z}\ \forall i\}$$

, where $x \in \mathbb{Z}^n$ is an n-dimensional integer vector being optimized, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ define linear constraints on x, and $d \in \mathbb{R}^n$ defines the linear objective function. Following the notation from [14], given a solution x and an optimal solution x_{opt} we define the primal gap $\gamma(x)$ as

$$\gamma(x) = \begin{cases} 0, & \text{if } |d^T(x_{\text{opt}})| = |d^T(x_{\text{opt}})| = 0, \\ 1, & \text{if } d^T(x_{\text{opt}}) \cdot d^T(x) < 0, \\ \dfrac{|d^T x_{\text{opt}} - d^T x|}{\max\{|d^T x_{\text{opt}}|\,,\,|d^T x|\}}, & \text{else.} \end{cases}$$

Primal Integral

Let γ_i denote the primal gap of the best solution after expansion of the i^{th} branch-and-bound node. If no solution has been found, we use a default value of $\gamma_i = 1$. Let N denote the total number of branch-and-bound nodes expanded so far. Then primal integral function $P(N)$ of a run is defined as follows:

$$P(T) = \sum_{i=1}^{N} \gamma_i \tag{1}$$

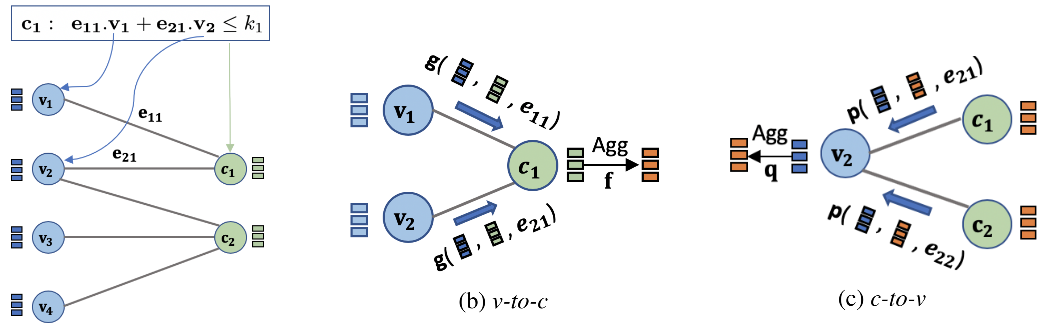
Primal integral also requires measuring the resource usage needed to achieve a given objective function value. The original definition uses running time [14], but that is hardware-dependent. Instead we use the total number of branch-and-bound nodes required across the calls to the `Repair`, which is hardware-agnostic.

Datasets

We evaluate our approach against a wide variety of challenging benchmark MIP datasets from the literature [19, 29, 30, 33, 35, 16, 11, 7]. We use data sets of MIPs of four different kinds labeled *cauction*, *facilities*, *indset*, *setcover*. We provide references to the original sources where details to generate them can be found. We also provide a brief description below for completeness. **Combinatorial Auction (*cauction*):** Following Leyton-Brown *et al.* [35], we generate combinatorial auction instances with 100 items and 500 bids. **Capacitated Facility Location (*facilities*):** Following Cornuéjols *et al.* [16], we generate capacitated facility location instances with 100 facilities and 100 customers. **Maximum Independent Set (*indset*):** Following Bergman *et al.* [11] we generate maximum independent set instances on Erdos-Rényi random graphs with 500 nodes, with affinity set to 4 (see [19]). **Set Cover (*setcover*):** Following Balas and Ho [7], we generate set cover instances with 500 rows and 1000 columns. In order to filter out trivial instances and bolster hardness, we apply several rounds of pre-processing before finally extracting useful features from the raw MIPs. Details about the two steps can be found below. All datasets except for *corlat* have been taken from [19].

Dataset Pre-processing & Feature Extraction

We generate a large number of sample problem instances and solve them to optimality, recording the primal integral achieved during this process. We select only those instances with primal integral at or above the 99.9^{th} percentile to include in our datasets. Finally we randomly split this filtered set into training, validation and test sets with 1000, 100 and 100 samples respectively. **COR LAT (*corlat*):** About 2000 MIP instances are generated based on real data used in the construction of a wildlife corridor for grizzly bears in the Northern Rockies region [29]. This dataset is part of the ACLIB benchmark library [30], which is a standard for evaluation of different methods for solving hard computational problems. Each MIP instance in the dataset is passed through the following stages of pre-processing and extraction before it can be used for training or evaluation.


$\mathbf{c_1}: \ \mathbf{e_{11}}.\mathbf{v_1} + \mathbf{e_{21}}.\mathbf{v_2} \leq k_1$
v1
v2
v3
v4
c1
c2
e11
e21
g(, , e11)
g(, , e21)
Agg
f
p(, , e21)
p(, , e22)
q

(a) Bipartite Representation

(b) *v-to-c*

(c) *c-to-v*

Figure 4: (a) MIP encoding using a bipartite graph. (b) Message passing step from variables to constraints followed by mean aggregation and update of constraint embeddings. (c) Message passing step from constraints to variables followed by mean aggregation and update of variable embeddings.

Pre-processing: The input MIP instance is transformed into an equivalent but a more compact formulation by passing it through the presolve routine in SCIP. Presolving is a relatively quick step run at the beginning of the solving process that usually results in instances with fewer variables and constraints that are easier to solve. *Feature Extraction:* We extract several static features corresponding to constraints, edges and variables similar to [19, 33]. Additionally, other dynamic features are incorporated into the variable features. A complete list of these features is provided below. *Solving MIP:* Finally, the MIP and its continuous relaxation are solved to optimality. The optimal objective and solution are stored alongside the dataset to be used to calculate the reward during training and report the final performance during evaluation. Note that the optimal solution is *not* used as an input feature or for any other purpose during training or evaluation. The first feasible solution found by SCIP during the solving process is also stored to be used as the starting solution for the LNS during training.

We will release all our datasets and the code required to reproduce our results once the paper is published.

Features

We provide a brief list of features used for training below. For a comprehensive list, please see the README file in our codebase.

Variable Features:

Variable feature vector $\mathbf{v}_i$ comprises of several static and dynamic features. Static features assigned to the variables include the solution to the LP relaxation, coefficients in the objective function, flags indicating the variable types (discrete/continuous). Some of the key dynamic features include the current incumbent solution, if and when the agent has called `Destroy` on the variables in the current local move *etc.*

Constraint and Edge Features:

All the constraints are first converted to $\leq$ form. Feature vector $\mathbf{c}_i$ of the constraint-nodes is simply the value on the right hand side of the constraint. For each edge, $e = (\mathbf{e}_k, v_k, c_k)$, the feature vector $\mathbf{e}_k$ is the normalized coefficient of the variable corresponding to v_k in the constraint corresponding to c_k.

Global Features:

Among others we use the following global features $\mathcal{G}$ alongside the bipartite graph features: objective value of the current solution, number of `Destroy` moves left in the current local move, number of local moves completed so far in this episode *etc.*

Policy Network Architecture:

Our GCNN network takes the state representation, $s_t = (V, C, E, \mathcal{G})$ as input and iteratively computes embeddings for the variable and the constraint nodes. This proceeds in T rounds as defined below.

Let the variable node embeddings at round t be $\mathbf{v}_k^{(t)}$ and the constraint node embeddings be $\mathbf{c}_k^{(t)}$. We initialize these embeddings with features from §3.4 *i.e.,* $\mathbf{v}_k^{(0)} = \mathbf{v}_k$, $\mathbf{c}_k^{(0)} = \mathbf{c}_k$. Each round involves two sequential stages of message passing, from variables to constraints and then from constraints to variables which we refer to as *v-to-c* and *c-to-v* respectively. See Figure 4 for a visual illustration. In the *v-to-c* stage, for each edge $\mathbf{e}_{(j,i)}$ incident to the constraint-node $\mathbf{c}_i$ from the variable-node $\mathbf{v}_j$, a "message" $\mathbf{m}_{(j,i)}$ is computed as $\mathbf{m}_{(j,i)} = \mathbf{g}^{(t)}\left(\mathbf{c}_i^{(t)}, \mathbf{v}_j^{(t)}, \mathbf{e}_{(j,i)}\right)$ where $\mathbf{g}^{(t)}$ is a multi-layer perceptron (MLP). These messages are pooled using mean aggregation and passed as input along with $\mathbf{c}_i$ to $\mathbf{f}^{(t)}$, another MLP network. The final update for the *v-to-c* stage takes the following form:

$$\mathbf{c}_i^{(t+1)} \leftarrow \mathbf{f}^{(t)}\left(\mathbf{c}_i^{(t)}, \quad \frac{1}{|\xi_{\mathbf{c}_i}|}. \sum_{j}^{j\in\xi_{\mathbf{c}_i}} \mathbf{g}^{(t)}\left(\mathbf{c}_i^{(t)}, \mathbf{v}_j^{(t)}, \mathbf{e}_{(j,i)}\right)\right), \tag{2}$$

where $\xi_{\mathbf{c}_i}$ is the set of all variable-nodes with an edge to the constraint-node $\mathbf{c}_i$ and $|.|$ denotes the cardinality operator. Once the constraint-node embeddings $\mathbf{c}_i^{(t+1)}$ are computed using the update rule (2), they are used to compute the updated variable-node embeddings $\mathbf{v}_i^{(t+1)}$ in a similar way using the following rule where $\mathbf{p}^{(t)}$, $\mathbf{q}^{(t)}$ are MLPs:

$$\mathbf{v}_i^{(t+1)} \leftarrow \mathbf{p}^{(t)}\left(\mathbf{v}_i^{(t)}, \quad \frac{1}{|\xi_{\mathbf{v}_i}|}. \sum_{j}^{j\in\xi_{\mathbf{v}_i}} \mathbf{q}^{(t)}\left(\mathbf{c}_j^{(t+1)}, \mathbf{v}_i^{(t)}, \mathbf{e}_{(i,j)}\right)\right). \tag{3}$$

Finally, after T rounds of updates, we get $\mathbf{v}_i^{(T)}$, $\mathbf{c}_i^{(T)}$. We discard the constraint-node embeddings. A final MLP is applied independently to the variable-node embedding and the global features independently at each node. A masked softmax function produces the final policy action distribution $\pi(a|s_t)$. All of our MLPs use the ReLU activation function [37]. Full list of all the architectural hyper-parameters can be found in the Appendix.

Hyper Parameters

A full list of the hyper-parameters will be made available along with the code which will be released upon publication. A brief list of few key hyper-parameters in our approach is provided below:

Hyper Parameter	**Value**
Learning Rate	10^{-4}
Entropy Regularization Constant	$2 * 10^{-2}$
GNN Dept	4
Number of Local Moves	25
Mini-batch size	8
Trajectory Length	32
Node embedding dimension	32
Edge embedding dimension	32
Global embedding dimension	32
Policy Torso Hidden Layer Sizes	$64, 64$
Value Torso Hidden Layer Sizes	$64, 64$
Replay memory size	64

Implementation Details

We use TensorFlow [2] framework with GraphNets library [9] for implementing and training our approach. Policy optimization in learner and batched inference in actors are accelerated using NVIDIA V100 GPUs [1]. We use SCIP (version: 6.0.1) [4] as our default MIP solver.