ORIE 6590 - Project

Reinforcement Learning for Integer Programming techniques

(https://github.com/ajagekarakshay/drl-project-6590)

Alyf Janmohamed and Akshay Ajagekar

1. Problem Description

Our project explores the applications of Reinforcement Learning (RL) techniques to solve combinatorial optimization problems. These problems have applications in a wide variety of disciplines including facility location, emergency response, and transportation. Unfortunately, solving large-scale combinatorial problems often takes a long time. In this project, we will attempt to leverage RL techniques to improve decisions made by algorithms that solve these problems. A significant portion of these problems can be formulated as Mixed Integer Linear Programs (MILPs) which are often solved using the Brand and Bound algorithmic scheme. In branch-and-bound, the feasible region is recursively split into smaller spaces by the process of branching a variable. The algorithm keeps track of the bounds and uses these bounds to prune the search space, eliminating candidate solutions that it can prove will not contain an optimal solution.

Definition : Mixed Integer Linear Program (MILP)

Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$ and a subset $I \subseteq \{1,...,n\}$, the mixed integer program $MIP = (A, b, c, I)$ is:

$$z^* = \min\left\{c^T x \mid Ax \le b, x \in \mathbb{R}^n, x_j \in \mathbb{Z} \ \ \forall j \in I\right\}$$

There are two main decisions that the Branch and Bound algorithm has to make at each step: the leaf node to explore and the fractional variable to split on. In this project, we focus on the latter, known as the branching problem. This problem can be modeled as a sequential decision making process with the objective of minimizing the solution time of an integer programming instance. Two proposed methodologies to the branching problem are strong branching and pseudo-cost branching. In strong branching LP relaxations are solved for each candidate variable for splitting. While this results in smaller search trees, this benefit is dwarfed by the amount of time required at each node. In pseudo-cost branching, the average benefit from branching on a variable is estimated by the benefit from branching on this node previously.

1.1. Related Work

The first work in this area that we are aware of is Khalil et al. [1] which formulated a supervised learning problem with the objective of predicting the relative ranking of variables to split on. The proposed methodology initially used strong branching to generate a training set, before training and using a machine learning algorithm to make branching decisions. In this paper, learning was confined to within each problem instance. Work by Alvarez et al. [2]and Hansknecht et al. [3] trained branching rules offline on a collection of instances before applying them to new instances. They formulated the problems differently, Alvarez et al. formulated a regression problem while Hansknecht et al. formulated a ranking problem (similar to Khalil et al.). In both these papers, the branching policy is learned by imitating strong branching.

Gasse et al. proposes a graph convolutional neural network model for learning variable selection policies [4]. By introducing this structure, they reduce the need for feature engineering. They also demonstrate that their methodology can generalize to larger instances than those used to train the model. The graph convolutional neural network works on graphs of all sizes and topologies and returns a graph of the same structure but with transformed features. It also has many nice properties like its complexity scales with density (combinatorial problems often have sparse structures) and it is permutation invariant. The work by Khalil et al. [5] proposed a graph convolutional neural network for learning heuristics on collections of combinatorial optimization problems defined on graphs. Another work uses a recurrent version of graph convolutions for solving satisfiability problems [6]. Lederman et al. [7] uses a bandit approach for variable selection to solve satisfiability problems using graph neural networks.

1.2. MDP Formulation

Solving a MILP via branch and bound can be modeled as an episodic reinforcement learning problem. In solving, we sequentially select nodes in the tree and variables to branch on. We will always eventually solve the problem since the complete tree has a finite size. However, our goal is to select nodes and variables that minimize the total solution time.

We use the MDP formulation from [4] which is summarized in the figure below (taken directly from below from their paper).

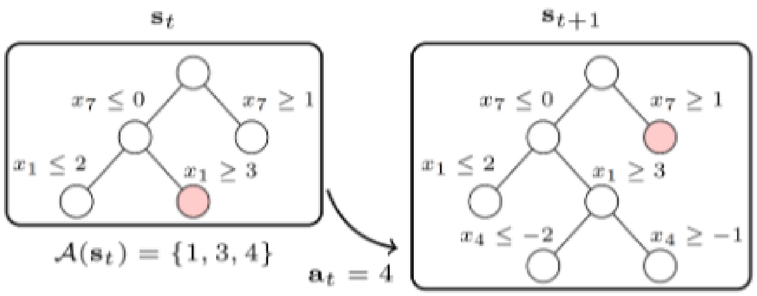

Figure 1. The figure illustrates one step in the MDP. The state on left, s_t consists of the branch and bound tree so far, and the node selected to branch on. We select one of three variables with fractional solutions at this node (x_1, x_3, x_4). We select x_4, the tree is updated and the solver selects the next node for branching.

State

As depicted in Figure 1, the state variable represents the current branch and bound tree and the node we are splitting on. More specifically, we have

- the branch and bound tree with all previous branching decisions
- the LP solution at each node
- the best feasible solution so far
- the current leaf node

The most important part of the state is the problem represented by the node that we are splitting on. As an input to our model, we construct a bipartite graph between the sets of constraints *C* and variables *V* An edge exists between a constraint and variable node only if the corresponding constraint has a non-zero coefficient for the specified variable. This procedure is described in [4].

Action

The action a_t represents the variable selected for branching at time step *t*, given the current state s_t.. The action space is dynamic in nature, since some variables cannot be branched based on the feasible region dictated by the current node of the branch and bound tree.

Reward

The objective is to minimize total solution time. The solution time directly corresponds with the number of B&B nodes explored. Therefore, we use the negative number of nodes explored as the reward function. Other reward functions like a heuristic similar to information gain can be used like conflict score or pseudo-cost.

Transition

The action selected specifies the fractional variable we will branch on (at the current leaf described in the state). The transition dynamics are summarized below:

- Solve the 2 LPs determined by the current node and the branching decision
- Extend the current tree by adding nodes that correspond to the 2 LPs we just solved
- Prune the tree according to standard branch and bound rules (if necessary) and update the best solution (if necessary)
- Use a heuristic from the solver the pick the next node to split on

Following these updates, we will have the next state.

2. Methodology

2.1. Model architecture

There are several possible ways to parameterize a Q-value function or a policy using a neural network. In this work, since the action space is limited to a select range of variables at each state that is encoded as a bipartite graph, we choose to parameterize the Q-value function with a graph neural network. Graph neural networks (GNNs) can be encompassed under the class of message-passing neural network (MPNN) [8]. The most commonly used MPNN is the graph convolution network that is an extension of the convolution neural network for graph-structure data. The choice of GNN for Q-value function parameterization is based on their unique properties of input size-invariance and permutation-invariant. This means that GNNs can use graphs of any size making them ideal for both sparse and dense inputs, and they produce the same output for a particular input graph irrespective of the order of considered nodes. The basic operations of a typical MPNN network are shown in Figure 2.

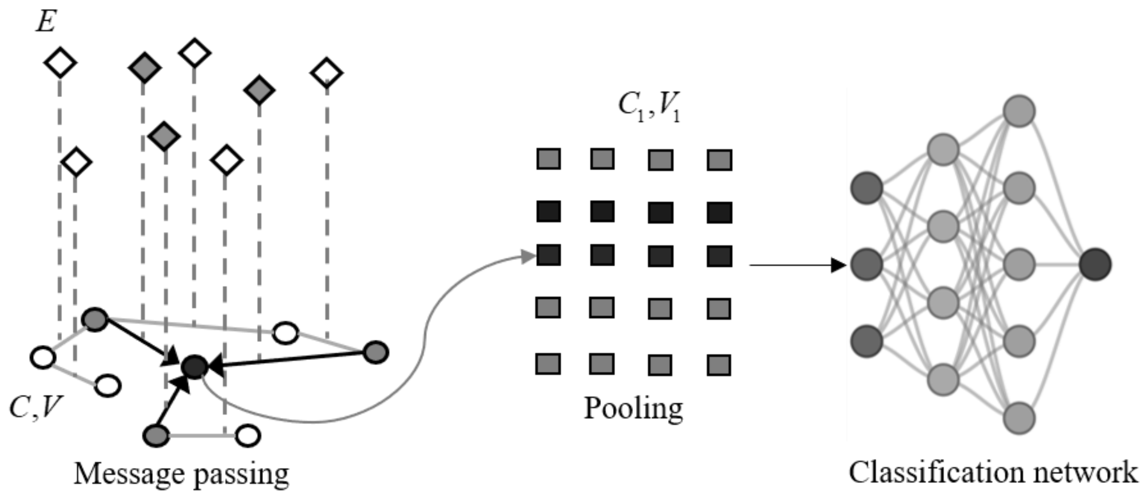

Figure 2. Operations in message passing neural network (MPNN)

Our MPNN model uses the bipartite graph denoted by $G \equiv (N_v, N_c, A, E)$ as input, where the graph G is represented by the set of variable nodes N_v, set of constraint nodes N_c, the adjacency matrix A, and the edge features E. It should be noted that the variable nodes N_v and constraint nodes N_c have different sizes of attributes or feature vectors. Previous works in literature that rely on GNNs for MILP branching do not consider the information stored in the edge features or the coefficients of the variable in a particular constraint. However, we choose to use this information and perform edge-conditioned convolutions in our MPNN model. Edge-conditioned filters in graph convolutional networks have been demonstrated to have better generalization capabilities [9]. The output of edge-conditioned MPNN is obtained as shown in Eq. 1. Embeddings of the node features are generated by a multi-layer perceptron (MLP) operation with parameters W_{root} and b. These embeddings or hidden layer features are further added to the aggregate sum of messages from its neighboring nodes. The messages are represented as the product of the neighboring node features and the embeddings of the edge features generated by a similar MLP operation. Due to the feature size mismatch of variable and constraint nodes, we perform two half convolutions on the variable and constraint nodes. Additionally, layer wise normalization is performed while generating the hidden layer embeddings to prevent issues like gradient explosion or overfitting. The output of the developed MPNN model are the final single-valued embeddings for each variable node of the input bipartite graph. These output Q-value function values denote the state-action value for the input state and the particular action or the variable.

$$x_i^{'} = x_i W_{root} + \sum_{j \in N(i)} x_j MLP\left(e_{ji}\right) + b \tag{1}$$

2.2. Algorithms

To solve the above sequential decision problem, we learn estimates of the optimal Q-value function by implementing the deep Q network (DQN) technique [10]. The parameterized Q-value function is denoted by $Q\left(s_t, \cdot \mid \theta\right)$. In DQN, the two primary components are the use of a target network and the use of experience replay. The target MPNN network is parameterized by parameters θ'. The target MPNN model is the same as the online or local MPNN network but is updated by performing a soft update step. The experience replay is performed by storing the transitions $\left(s_t, a_t, r_t, s_{t+1}\right)$ or agent's experiences in a fixed size buffer also termed as the agent's memory. During the agent's training, Q-learning updates are performed on the local MPNN network by uniformly drawing a batch of experiences at random from the agent's memory. With the sampled batch of collected experiences, the targets used by DQN are shown in Eq. 2. The Q-learning update for DQN is performed by minimizing the loss function in Eq. 3 to update the online MPNN network parameters θ. The target MPNN is updated by performing a soft update $\theta' \leftarrow \tau\theta + \left(1-\tau\right)\theta'$ with τ as a pre-specified hyperparameter.

$$Y_t^{DQN} = r_{t+1} + \gamma \max Q\left(s_{t+1}, a; \theta'\right) \tag{2}$$

$$L_{DQN} = E_{(s,a,r,s')}\left[\left(Q\left(s_t, a_t; \theta\right) - Y_t^{DQN}\right)^2\right] \tag{3}$$

DQN's learning steps are similar to the standard Q-learning approach and use the max operator to select and evaluate a particular action. This has been shown to result in over-optimistic value estimates. To prevent this issue, decoupling of the evaluation and selection is necessary and is the main idea behind double DQN (DDQN) [11]. Similar to the DQN approach, DDQN also uses two networks to estimate the optimal Q-values. To this end, the targets used by DQNN for experiences uniformly sampled from the agent's memory can be calculated in Eq. 4. These targets are consistent with the decoupling idea behind DDQN, wherein the selection of actions in the argmax is due to the local network while the evaluation of the Q-value estimates is done using the target network parameters θ'. The online and target MPNN networks can then either be updated using the sift update rule or symmetrically by switching their roles. In this work, we implement both DQN and DDQN to compare the efficacy of the algorithms and the edge-conditioned MPNN network on the branching problem.

$$Y_t^{DDQN} = r_{t+1} + \gamma Q\left(s_{t+1}, \underset{a}{\arg\max}\, Q\left(s_{t+1}, a; \theta\right); \theta'\right) \tag{4}$$

2.3. Prioritized Replay

The original proposals of the DQN and DDQN algorithms use an experience replay where the experiences are uniformly selected from the agent's memory. However, this approach does not consider the significance of the experiences and treats each experience with the same importance. For the branching problem, branching on some nodes may lead to pruned branches earlier than others. Therefore, we investigate whether prioritizing the experiences can make the experience replay much more effective by implementing a greedy prioritized experience replay (PER) memory for both DQN and DDQN agents [12]. This importance or ranking of the experiences can be provided by the TD error. The goal of greedy prioritized replay is to sample experiences with maximal TD error. The practical implementation of greedy prioritized replay is computationally expensive due to the sorting and updating of the underlying experience. That is why a binary heap structure is used which has low sorting and sampling complexities.

2.4. Training and Implementation details

The MPNN models are implemented using a Tensorflow based GNN library called Spektral. The DQN and DDQN agents are also trained with optimizers based in Tensorflow. An epsilon greedy strategy is implemented for action selection for these agents wherein the epsilon or probability of exploration is annealed consistently at a decay rate of 0.995. The maximum size of the episode is set to 500 with the size of the agent's replay memory of 5000. The size of a sampled batch for performing Q-learning updates is 120 while the agent is updated every 50 timesteps. The discount factor is set to 0.99 while the soft-update factor τ is set to 0.001. Adam optimizer is used to perform loss minimization and updating the MPNN model parameters. Replay memory is implemented using deque as the primary data structure while the prioritized experience replay uses binary heaps. Values like rewards, losses, timesteps are recorded using the Tensorboard utility for easier visualization. The Ecole library provides support for the branching environment used in this work.

3. Results and Discussion

To evaluate the models and algorithms used in this work, we perform several experiments using the set cover problems for MILP optimization. We use the negative difference of the number of nodes which is defined as the total number of nodes processed since the previous state. The Ecole library also provides support for other reward functions like the number of LP iterations, solving time, and single reward on terminal states. We train the DQN and DDQN agents on the branching environment created with Ecole using both experience replay and prioritized experience replay. The training curves for DQN and DDQN with experience replay along with DDQN with PER are shown in Figure 3. The figure represents the smoothened average reward per episode during the iterative learning process. DQN with PER results in a decrease in rewards overtime followed by an intermittent sharp increase and decrease in average score. The heap-based PER has the disadvantage of sampling initial transitions much more frequently than the later ones which may lead to some transitions not replayed at all. This combined with the over-optimistic nature of DQN may have resulted in this behavior. However, in the remaining three cases an increase in an average score over time is observed. Compared to DQN, a steep increase in the average score for DDQN is observed but is accompanied by further reduction. Additionally, DDQN and DDQN with PER exhibit some of the highest average score values.

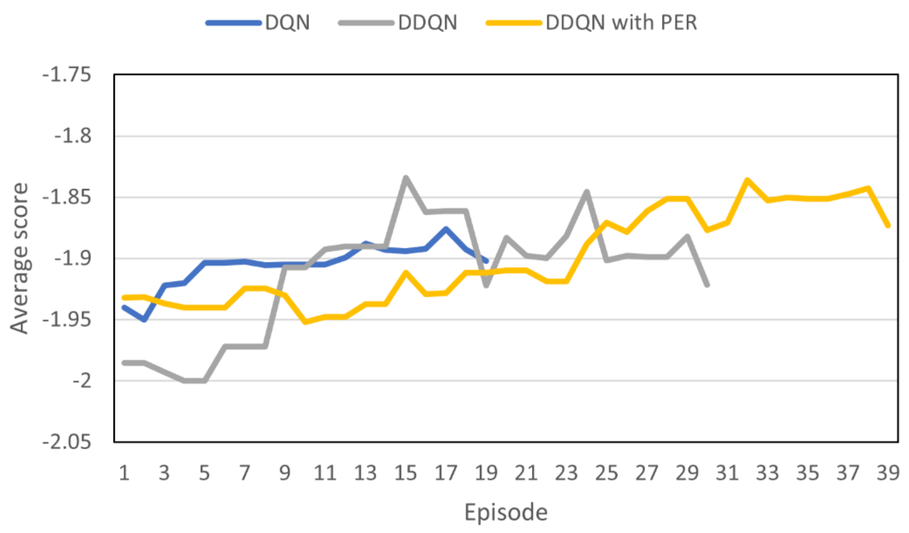

Figure 3. Training curves tracking the agent's average score

The trained agents are further used to evaluate their branching efficiency during the branch and bound procedure in MILP optimization. We generate random set cover instances of varying complexities for this purpose. The instances were generated with instance generators provided within Ecole comprising of 300 rows and 400 columns and varying densities. The computational experiments were conducted on a Google cloud virtual machine optimized for high performance and GPU support. We compare each of the DRL approaches with a baseline state-of-the-art branching rule that is a variant of hybrid branching and is used by the SCIP solver. The computational times required to solve the set cover instances along with the number of nodes are reported in Table 1. Compared to SCIP's default branching rule, each of the DRL algorithms performs better in terms of the resulting number of nodes. This outcome is expected due to the minimization of the number of nodes explored by the agent in the sequential decision process. In terms of timing performance, DQN with PER requires more computational time to solve set cover instances. This can be attributed to an increase in the number of LP iterations resulting in an increase in solution time. On the other hand, DQN with PER also performs worse than other DRL algorithms which could be due to the limitations of both DQN and PER. The remaining three algorithms perform much better than SCIP solver in terms of both computational times and resulting B&B nodes. Additionally, the best method in terms of nodes is not necessarily the best in terms of total solving time. The superior performance of the DRL algorithms being trained on smaller set cover instances and evaluated on larger and denser instances demonstrate the generalization capabilities of the edge-conditioned MPNN model to larger bipartite graphs.

Table 1. Evaluation on set cover instances in terms of solving time and number of resulting B&B nodes for each model

Model	**Time (s)**	**Nodes**
SCIP	1.21 ± 0.74	16 ± 25%
DQN	1.12 ± 0.3	5 ± 40%
DQN with PER	1.41 ± 0.88	10 ± 71%
DDQN	0.92 ± 0.23	5 ± 45%
DDQN with PER	1.04 ± 0.35	6 ± 60%

4. Conclusion

In this work, we conducted an analysis of the state-of-the-art DRL algorithms for branching in MILP optimization. The Q-value function was parameterized by an edge-conditioned MPNN network. DQN and DDQN algorithms were implemented with experience replay to solve the sequential decision problem. We also implemented a binary heap-based PER to overcome the limitations of uniform experience replay. Several computational experiments were conducted with the set cover problem instances to demonstrate the applicability and efficiency of the DRL approaches. The four DRL approaches perform better than off-the-shelf solver SCIP in terms of the number of nodes in the B&B tree. Generalization capabilities of the DRL approaches to larger set cover instances were also demonstrated. However, generalization of these techniques to MILP problems with different structures requires further analysis.

References

[1] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016, vol. 30, no. 1.

[2] A. M. Alvarez, Q. Louveaux, and L. Wehenkel, "A machine learning-based approximation of strong branching," *INFORMS Journal on Computing*, vol. 29, no. 1, pp. 185-195, 2017.

[3] C. Hansknecht, I. Joormann, and S. Stiller, "Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem," *arXiv preprint arXiv:1805.01415*, 2018.

[4] 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.

[5] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," *arXiv preprint arXiv:1704.01665,* 2017.

[6] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from single-bit supervision," *arXiv preprint arXiv:1802.03685*, 2018.

[7] G. Lederman, M. N. Rabe, and S. A. Seshia, "Learning heuristics for automated reasoning through deep reinforcement learning," *arXiv preprint arXiv:1807.08058,* vol. 57, 2018.

[8] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural Message Passing for Quantum Chemistry," presented at the Proceedings of the 34th International

Conference on Machine Learning, Proceedings of Machine Learning Research, 2017. Available: http://proceedings.mlr.press/v70/gilmer17a.html

[9] M. Simonovsky and N. Komodakis, "Dynamic edge-conditioned filters in convolutional neural networks on graphs," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 3693-3702.

[10] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529-533, 2015/02/01 2015.

[11] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016, vol. 30, no. 1.

[12] T. Schaul, J. Quan, I. Antonoglou, and D. J. a. p. a. Silver, "Prioritized experience replay," 2015.