57º SBPO
SIMPÓSIO BRASILEIRO DE PESQUISA OPERACIONAL
Gramado • RS
DE 5 A 9 DE OUTUBRO DE 2025

Tackling combinatorial optimization with Hybrid DQL: A case study with the Minimum Dominating Set Problem

Osniel Lopes Teixeira
Fluminense Federal University
Av. Gal. Milton Tavares de Souza, s/n – Boa Viagem - Niterói/RJ
osnielt@id.uff.br

Fabio Protti
Fluminense Federal University
Av. Gal. Milton Tavares de Souza, s/n – Boa Viagem - Niterói/RJ
fabio@ic.uff.br

Aline Paes
Fluminense Federal University
Av. Gal. Milton Tavares de Souza, s/n – Boa Viagem - Niterói/RJ
alinepaes@ic.uff.br

ABSTRACT

This paper proposes combining reinforcement learning —- specifically, deep Q–learning —- with a local search procedure to tackle NP-hard combinatorial optimization problems. We select the Minimum Dominating Set problem, a classic combinatorial optimization problem from Graph Theory, to assess the proposed approach. Each model was trained on graphs sampled from a fixed Erdős–Rényi $G(n, p)$ distribution, and the ground truth solutions for the minimization problem were computed via Integer Linear Programming with the Gurobi Optimizer. We report results in terms of average optimization ratios and the average sizes of the produced solutions. Additionally, a greedy algorithm was implemented to serve as a performance baseline. Our findings indicate that the Deep Q-Network alone was insufficient to discover optimal solutions; however, when combined with the local search procedure, the proposed method achieved competitive performance on small graph instances.

KEYWORDS. Minimum Dominating Set. Deep Q-Learning. Graph-Based Learning. Combinatorial Optimization

Paper topics: AI (OR and AI), TAG (Graph Theory and Related Algorithms).

1. Introduction

Combinatorial Optimization (CO) problems play a vital role in numerous practical applications, such as traffic, routing, network design, task parallelization, and transportation planning Paschos [2010]. They also have significant theoretical importance in Computer Science, as many of these problems are known to be NP-hard [Papadimitriou and Steiglitz, 1998]. A classic example is the *Dominating Set* problem, with applications ranging from facility location and social network analysis to coding theory [Rani, 2019]. In this work, we specifically investigate the *Minimum Dominating Set problem* (MDS).

The MDS aims at finding a set of vertices that "dominate" the graph and has minimum size. Let $G = (V, E)$ be a graph where V is the set of vertices and E is the set of edges. Let $\mathcal{N}(\cdot)$ be

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br

the neighborhood function that takes a set $S \subseteq V$ of vertices as input and returns the set of adjacent vertices to any of them. Then we can state the MDS as the problem of finding the minimum set of vertices S such that

$$S \subseteq V \text{ and } \forall v \in V,\ v \in S \text{ or } v \in \mathcal{N}(S) \tag{1}$$

Garey [1979] showed that the decision version of MDS is NP-complete. Because of that, it is believed that there is no efficient algorithm to solve the MDS. Since then, many heuristics and approximation algorithms have been proposed to approximate the solution. More information on the MDS can be found in [Hedetniemi and Laskar, 1990; Crescenzi et al., 2000].

Designing effective approximation algorithms for CO problems often requires deep domain expertise and extensive trial and error. Recently, however, researchers have turned to Machine Learning (ML) to tackle these challenges more efficiently, leveraging its pattern recognition capabilities [Vinyals et al., 2017]. This work investigates how Reinforcement Learning (RL), a family of ML algorithms, can automate heuristic discovery, offering a data-driven alternative to traditional algorithm design. Although existing studies have focused mainly on well-known CO problems like the Traveling Salesperson Problem (TSP) and Minimum Vertex Cover [Peng et al., 2021], MDS has received relatively little attention - a gap we aim to address.

Specifically, we employ Deep Q-Learning (DQL) to train a neural network-based agent explicitly tailored for the MDS. The framework learns to construct candidate solutions through repeated interactions with graph environments. We integrate a local search post-processing step that systematically improves the model's outputs by exploring neighboring solution spaces to refine these solutions further. This hybrid methodology combines the exploratory strength of reinforcement learning embedded in a graph neural network [Scarselli et al., 2009] with the precision of classical combinatorial optimization techniques.

Our findings reveal that for graphs with 10 to 50 nodes and edge density ranging from 15% to 45% the model was able to maintain the learned approximation heuristics under 2 times the ground truth. A local search routine was included to achieve a tighter approximation ratio. On average and compared to a greedy procedure, this routine was able to find a better or equal level of success in 67% of the test cases.

In this way, we contribute to the study of hybrid machine learning approaches for combinatorial optimization by investigating how learned heuristics can be effectively combined with traditional local search techniques. Our results demonstrate that even without complex tuning, the model can produce good solutions that are further improved by a simple local refinement step. This highlights the potential of such hybrid strategies to bridge the gap between data-driven and algorithmic methods. By reporting empirical results and methodological insights, we hope to support further exploration of machine learning as a practical tool for solving classical optimization problems and to encourage future research to refine and expand this line of work.

The remainder of the paper is organized as follows. Section 2 describes the current status of the literature on related approaches for MDS and CO with ML. Section 3 presents the preliminary technical definitions. In Section 4, we describe the methodology used. Section 6 reports on the results found. Finally, Section 7 summarizes our findings and discusses potential directions for future work.

2. Related Work

In Vinyals et al. [2017] the authors used ML to approximate CO solutions, introduzing Pointer Networks, a neural architecture that uses attention mechanisms to output discrete tokens corresponding to positions in the input sequence. Unlike traditional sequence-to-sequence models, Pointer Networks are especially suited for solving combinatorial optimization problems where the

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br

output sequence length depends on the input length, such as the Travelling Salesman Problem and Convex Hull computation.

Previous works have already investigated reinforcement learning to tackle combinatorial optimization problems. Dai et al. [2017] proposed a learning framework that automatically combines graph embeddings and reinforcement learning to construct greedy heuristics for combinatorial optimization problems on graphs. Their method, called S2V-DQN, represents partial solutions using a learned embedding (Structure2Vec) and trains a Q-function using fitted Q-learning to guide node selection. This framework is applied to classic problems such as Minimum Vertex Cover, Maximum Cut, and the Traveling Salesperson Problem, achieving near-optimal performance and outperforming hand-crafted heuristics in many cases. However, they have not selected MDS in their investigation.

Other works have studied MDS, but using more sophisticated architectures. For example, Sato et al. [2019] proposed a method called Consistent Port Numbering Neural Networks, which gives a way to mark the nodes from which each message is coming. Based on port numbering models and distributed local algorithms, they showed that their model can approximate the MDS with a ratio of $\frac{\Delta G+1}{2}$, where ΔG is the maximum degree in the graph G. They also showed that their model was representatively more efficient than Graph Isomorphism Networks, which were considered the most powerful GNN at that time [Xu et al., 2019]. Chen et al. [2024] also presented results for the MDS using RL, but relying on the Double Deep Q-Network model, trained on grid graphs, triangular lattice graphs, and hexagonal lattice graphs. Here, we focus on investigating the performance of traditional GNNs refined with local search to tackle the MDS problem. In our solution, the nodes are anonymous with respect to message-passing. Zhang et al. [2023] introduced a novel unsupervised learning framework based on Generative Flow Networks (GFlowNets) to solve NP-hard graph combinatorial optimization problems (e.g., Maximum Independent Set, Maximum Clique, Minimum Dominating Set, and Maximum Cut). By designing problem-specific Markov Decision Processes (MDPs), the authors trained conditional GFlowNets to sample solutions proportionally to their quality. Key contributions include a transition-based training algorithm for improved efficiency and scalability, and the use of intermediate learning signals to accelerate credit assignment.

3. Preliminaries

This section presents key concepts necessary for our proposal. First, we briefly describe Reinforcement Learning, which is the framework chosen to learn the heuristic for the MDS via trajectories experienced in the graph environment. Secondly, we give a summary of Graph Neural Networks, which are used in this work as a way of bridging the gap between the topological structure of the graph and the inference of the Q function. Finally, we formalize the concept of Domination Potential, an attribute devised with the intention of providing the network with more information about how interesting a node can be in the context of graph domination. This attribute is also used as a guide for the greedy procedure we implemented.

3.1. Reinforcement Learning

In Reinforcement Learning (RL), problems are typically modeled as Markov Decision Processes (MDPs), which provide a formal description of the environment where agents interact. An MDP is defined as a 5-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where:

- $\mathcal{S}$ (state space) is the set of all possible states s the environment can be in.
- $\mathcal{A}$ (action space) is the set of all possible actions a the agent can take.
- $P(s' \mid s, a)$ (transition probability function) is the probability of transitioning to state s' from state s while taking action a, encoding the dynamics of the environment.

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br

- $R(s, a)$ (reward function) is the expected immediate reward received after taking action a in state s. Alternatively, some formulations define $R(s, a, s')$ as the reward for a transition.
- $\gamma \in [0, 1]$ is the discount factor, determining the importance of future rewards. This way, when $\gamma = 0$ only immediate rewards matter, and when $\gamma \to 1$, future rewards are nearly as important as immediate ones.

The goal of the agent is usually to learn a policy $\pi(a \mid s)$ (which is another form of the transition probability function) through consecutive episodic interactions with the environment that maximizes the expected cumulative reward G, defined as

$$G = \sum_{k=0}^{\infty} \gamma^k R(s_k, a_k). \tag{2}$$

Beyond learning the policy, it is also possible for an agent to learn: (i) the Value function, which is the expected return if it starts in a state or state-action pair, and then acts according to a particular policy forever after: $V(s) = \mathbb{E}[G \mid s_0 = s]$; (ii) the Action-Value (also named Quality) function $Q(s, a) = \mathbb{E}[G \mid s_0 = s,\ a_0 = a]$, which gives the expected return if it starts in state s, takes an arbitrary action a (which may not have come from the policy), and then acts according to policy π. The Action-Value function is more "specific" than the Value function.

Diverse algorithms take this model and try to learn the best policy. Those algorithms can use different criteria of optimality, and some examples are brute force search, simulated annealing, evolutionary computation, fitted Q-function, and Q-learning.

Deep Q-Learning We leverage Deep Q-Learning (DQL) [Mnih et al., 2015] as an RL algorithm to tackle the MDS problem. Q-Learning is an approach in Reinforcement Learning that focuses on learning the action-value function $Q_\pi(s, a)$, that is, the expected return starting from state s, taking action a, and then following policy $\pi : Q_\pi(s, a) = \mathbb{E}_\pi[G \mid s_t = s, a_t = a]$. The term "Deep Q-Learning" (DQL) refers to the use of a Deep Neural Network to approximate such Q-function — that is, the network learns to estimate the expected return for each pair of (graph state, action). We will use the Deep Q-Network (DQN) nomenclature when referring to a model instance of this framework. We use the notation Q_Θ when referring to a network trained with parameters Θ to emulate the Q-function.

3.2. Graph Neural Networks

Graph Neural Networks (GNNs) represent a powerful class of neural network architectures specifically designed to process and learn from graph-structured data by exploiting topological relationships [Scarselli et al., 2009]. At their core, GNNs employ a message-passing framework, where the nodes of a graph iteratively aggregate and update their representations by exchanging information with their local neighborhoods. Each message-passing iteration (corresponding to a network layer) progressively propagates information one hop further through the graph structure, enabling nodes to capture increasingly broader structural contexts.

In this work, we used the implementation of Graph Convolutional Operator [Kipf and Welling, 2017] provided by the PyTorch Geometric library. This operator is described as

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}\Theta^{(l)}\right), \tag{3}$$

where H is the matrix of latent attributes with each line holding the representation for one vertex, l is the layer index, $\tilde{A}$ the adjacency matrix with added self-loops $A + I$, $\tilde{D}$ is the diagonal degree

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br

matrix of $\tilde{A}$, Θ the trainable weights of the network layer and σ an activation function. At the start of the process, we have $l = 0$, $H^0 = X$. Here, X is the matrix of original node attributes provided as input to the network.

3.3. Domination Potential

To allow the network to have some hint on which nodes are more useful when building a DS, we devised the concept of *Domination Potential* (DP). The idea is to have a value that grows in proportion to the number of nodes to be dominated by a vertex if it were added to some partial solution. Note that it is a dynamic attribute that depends on the partial solution of the MDS; as S increases, the DP of neighbors of recently incorporated vertices decreases. It is a real value in the interval $[0, 1]$, and, independently of the value of ΔG, it is a well-behaved attribute. Let S be a partial solution to MDS, and let D be the set of dominated vertices. We have:

$$D = S \cup \mathcal{N}(S), \tag{4}$$

$$DP(v) = \frac{|\,\mathcal{N}(v) \,\cup\, \{v\} \setminus D\,|}{\Delta G + 1}. \tag{5}$$

Here are some examples: for a vertex v with $deg(v) = \Delta G$, and current solution $S = \{\}$, we have $DP(v) = \frac{|\mathcal{N}(v) \cup \{v\} \setminus \{\}|}{\Delta G+1} = \frac{\Delta G+1}{\Delta G+1} = 1$, meaning that this vertex has the maximum domination potential; on the other side, if $v \in S$, then $\mathcal{N}(v) \cup \{v\} \subseteq D$, which implies $DP(v) = \frac{|\{\}|}{\Delta G+1} = 0$, meaning that this node is already on the solution set and not a good choice, since it has the lowest possible domination potential.

This attribute is also used to guide the choices of the greedy procedure.

4. Proposed method

This section presents our proposed approach to tackle the MDS problem with reinforcement learning empowered by graph neural networks and a refinement local search step. We start by describing the problem formulation, followed by the RL algorithm we implemented and the GNN architecture that supports learning the RL approximation function.

4.1. Formalizing MDS as an instance of RL

We formulate the MDS as an RL agent as follows.

- Each state is represented as a tuple, $(\mathbf{A}, \mathbf{X})$, where $\mathbf{A}$ is the adjacency matrix that encodes the graph; $\mathbf{X}$ is the matrix of attributes, where each row holds the attributes of a node. In other words, $\mathbf{A}$ holds the topology of the graph and $\mathbf{X}$ the attribute information related to the nodes. Matrix $\mathbf{X}$ has two columns: in the first, it stores a binary value that represents which nodes were chosen to be on the DS, holding the value 1 if the node corresponding to that row is in the DS, and0 otherwise; in the second, we have the DP value corresponding to each vertex. Every time a new node is added to a partial solution, both columns are updated.

- Action set: The action set is the same as the vertex set of the graph, $A = V$. Every time an action is chosen, the corresponding node is added to the partial solution of the MDS. Nodes already in the solution cannot be selected again, meaning they are not viable actions to be taken on the environment.

- Reward function: Each node selected has a reward of $-1/n$. Using the RL formalism, $R(s, a) = -1/n$. We use n (the number of nodes of the graph) in the denominator to keep the total return normalized. Since the RL tries to maximize the cumulative reward, using a negative value will favor the least possible number of actions, aligning with the minimization objective of the MDS.

Using this formulation, a complete episode on the graph environment corresponds to constructing a DS one node at a time, and smaller DSs will receive the highest return.

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br

4.2. Reinforcement Learning Algorithm

Our DQL implementation works with three core components: (a) Q-network, a neural network $Q_\theta(s, a)$ parameterized by θ, whose goal is to approximate the Q-function; (b) a target network $Q_{\theta^-}(s, a)$, which is a periodically updated copy of the Q-network to stabilize training; (c) a replay buffer, a memory $\mathcal{M}$ to store past transitions (s, a, r, s') for experience replay. The algorithm starts by initializing both the Q-network and the target network with random weights. Then, random episodes are constructed and added to the memory buffer. After that, the training starts. For each epoch, it initializes the state s. It selects an action, executes it, observes the outcome, stores the observed transition (s, a, r, s') in the replay buffer $\mathcal{M}$, sample a batch of transitions (s_i, a_i, r_i, s'_i) from $\mathcal{M}$, and compute the targets $y_i = r_i + \gamma \max_{a'} Q_{\theta^-}(s'_i, a')$. Finally, it updates the Q-network using the sampled batch of transitions to minimize the loss function. At each C steps, the weights of the Q-network are copied to the target network. The algorithm is depicted in 1.

Algorithm 1 Deep Q-Learning Algorithm

Initialize Q-network Q_θ and target network $Q_{\theta-}$
Initialize replay buffer $\mathcal{M}$
for each epoch **do**
    initialize state s
    choose action a using $\mathcal{E}$-greedy(Q_θ)
    take action a, observe r, s'
    store (s, a, r, s') in $\mathcal{M}$
    sample minibatch from $\mathcal{M}$
    compute target: $y = r + \gamma \times max_a Q_{\theta-}(s'_i, a')$
    update Q_θ by minimizing loss $\mathcal{L}(\theta) = \frac{1}{N} \sum_i (Q_\theta(s_i, a_i) - y_i)^2$
    update $\theta- \leftarrow \theta$ every C steps
    $s \leftarrow s'$
end for

4.3. Graph Neural Network Architecture

Our neural network architecture is composed of ten layers of the graph convolution operator previously described. Each layer output has a channel of size 64, which means that we have a latent vector of size 64 for each node. After those ten passages, we have a vector that works like a signature that characterizes each node on the graph with its local 10-hop neighborhood. The signatures of all the nodes are then aggregated via summation, so that we have a vector of 64 positions, representative of the whole graph. This graph signature is then passed through a layer of perceptrons, keeping the vector size. This vector is appended to each node representation so that we include the processed global information with the local information, leaving us with a vector of 128 positions for each node. Finally, we use again a perceptron layer that takes each node vector and maps it into a single output value: the estimated Q function value. The activation function in intermediate layers was the ReLU, and a hyperbolic tangent was used for the final layer. The schematic representation of this network can be seen in Figure 1.

4.4. The hybrid method

Having trained a network to emulate the Q-function for MDS (Q_Θ), we use it to build the problem solution for a graph, following the Algorithm 2. This algorithm starts with an empty solution S, and, at each round, it asks for the Q values of the nodes. Then it selects the node with the highest Q value (that is not already part of the solution) and adds it to the solution. In this procedure, we use the auxiliary COMPUTE_ATTRIBUTE_MATRIX function, which is responsible for receiving

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br

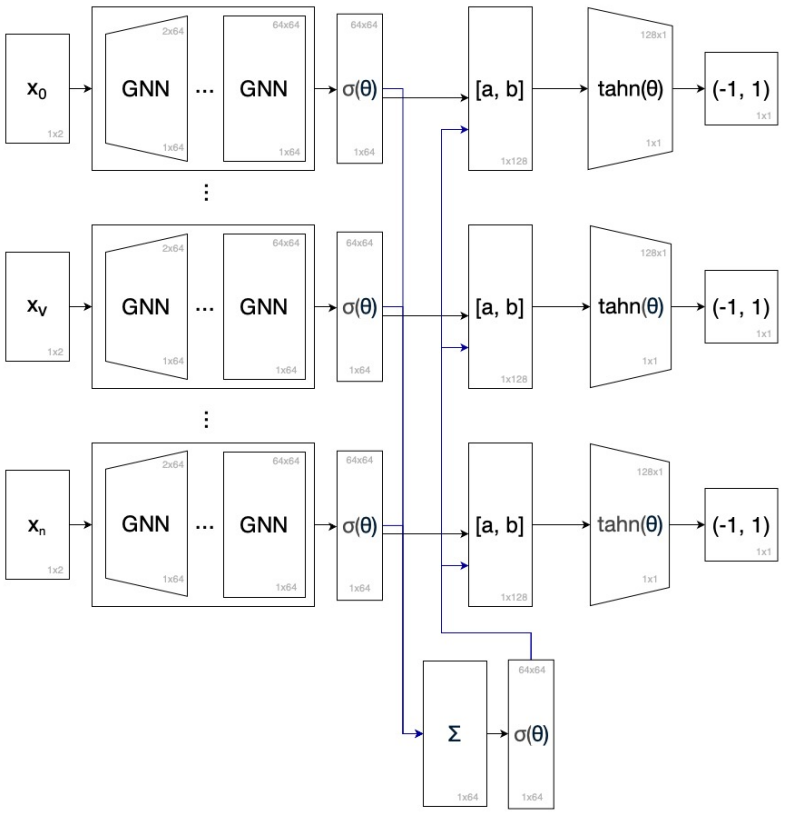


Figure 1: Schematic representation of the neural network: σ represents the ReLU activation function, [a, b] represents the concatenation of the two input vectors, and Θ is the learnable weights for each layer. Inside the boxes, at the top is the shape of the learnable parameters for each layer, and at the botton is the output shape.

an intermediate MDS solution and a graph in the form of an adjacency matrix, and computing the attribute matrix X. The first column of X encodes which nodes belong to the (intermediate) solution, and the second column stores the DP of each node, as explained before.

After a DS S is found via the greedy algorithm described above, we use a local search procedure to enhance the quality of the solution S. The method of the local search is formalized in Algorithm 3.

5. Experimental Setup

To assess our solution, we generate graphs using the Erdős–Rényi $G(n, p)$ distribution. This model generates graphs with n nodes, and each edge has a probability p of being included in the graph. We used $n \in \{10, 20, 30, 40, 50\}$ and $p \in \{0.15, 0.30, 0.45\}$. For each combination of (n, p), 10,000 graphs were generated to be used as the training mass. This gives a total of $15 \times 10,000$ graphs.

Graphs were randomly selected from the training mass, and some episodes were played on them with actions chosen randomly. From those episodes, the transitions (s, a, r, s') were collected. Those transitions were stored in the replay buffer M until it reached its full capacity, which was set to 100,000 transitions.

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br

Algorithm 2 Greedy MDS Via Q-Function

Input: Graph as an adjacency matrix A, trained Q_Θ function
Output: Dominating set S

$S \leftarrow \{\}$
$X \leftarrow$ COMPUTE_ATTRIBUTE_MATRIX(A, S)
while not IS_DS(A, S) **do** ▷ IS_DS returns true if S is a DS of the graph defined by A
    $v^* \leftarrow \arg\max\{Q_\Theta(A, X, v) \mid \forall v \in V \setminus S\}$
    $S \leftarrow S \cup \{v^*\}$
    $X \leftarrow$ COMPUTE_ATTRIBUTE_MATRIX(A, S)
end while
return S

Algorithm 3 Local Search

Input: Graph G, initial solution S: indexable set
Output: A (locally) improved dominating set S

for all $v \in S$ **do** ▷ Try to reduce the solution S one node at a time
    $S' \leftarrow S \setminus \{v\}$
    if IS_DS(G, S') **then**
        $S \leftarrow S'$
    end if
end for

for $i = 1$ to $|S| - 1$ **do** ▷ Two by one reductions
    for $j = i + 1$ to $|S|$ **do**
        $u \leftarrow S[i]$
        $v \leftarrow S[j]$
        if $u \notin S$ or $v \notin S$ **then**
            continue
        end if
        for all $w \in N(u) \cap N(v)$ **do**
            $S' \leftarrow S \setminus \{u, v\} \cup \{w\}$
            if IS_DS(G, S') **then**
                $S \leftarrow S'$
                break
            end if
        end for
    end for
end for
return S

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br

The Algorithm 1 runs up to a maximum of 2,500 epochs, meaning that it runs for 2,500 total iteration steps with the environment. Every time an episode reached the end, a DS was built; then the environment was reset by choosing a new random graph from the training mass. The γ parameter in DQL is set to 0.99, which makes future rewards have almost as much impact as the most recent ones. The ϵ parameter linearly decayed from 100% in the first epoch to 5% in the 2,000th epoch, remaining at that level until the end of the training.

The batch size of the training was 5,000, the max learning rate was 0.02, and the loss minimization algorithm used was Adam Optimizer. Since DQL is quite unstable because of the correlations present in the sequence of observations, and since the updates to Q_{Θ^-} may significantly impact the policy of the agent, we adopted the following schedule for the learning rate: linear warmup from 0 to the maximum value of 5% of the duration of the wait time of C epochs for the network replace, then a cosine annealing on the 95% left of the C epochs hiatus.

Also, a value of C that is too low can compromise the training, preventing the network from learning sufficiently before being replaced, thus creating instability. On the other hand, a value of C that is too large needs more extensive total training periods. To strike a balance on the period of update C, we make the first two replacements of the Q_{Θ^-} on the 100th epoch, and then 100 epochs later. After that, we find the next C value as the sum of the two previous ones, meaning that C grows following the Fibonacci sequence $F(i+1) = F(i)+F(i-1)$ where $F(0) = F(1) = 100$.

To choose the best model from the training history, we used a validation set of 300 $G(n, p)$ graphs, attached with the ground truth values for the MDS. These were computed using Integer Linear Programming (ILP) with the Gurobi Optimizer. The model for the ILP is as follows: let $\mathbf{x} \in \{0, 1\}^n$ be the vector encoding the MDS solution, where $x_i = 1$ if and only if node i belongs to the solution.

$$\text{MDS:} \qquad \min_{\mathbf{x}} \sum_i x_i \tag{6}$$

$$s.t. \qquad x_i + \sum_{j \in \mathcal{N}(i)} x_j \geq 1, \quad \forall i \in [n]. \tag{7}$$

The best model from the training history was selected following the lowest average approximation ratio (apx-ratio) found. The average approximation ratio was computed according to the following formula, where S^* is the solution found via ILP, S_{DQN} is the solution found by the DQN, and G is one of the graphs from the validation set:

$$\frac{\sum_G \frac{|S_{DQN}(G)|}{|S^*(G)|}}{\text{validation set size}} \tag{8}$$

To create the test data mass, we used $G(n, p)$ distributions, with $n \in \{10, 20, 30, 40, 50\}$ and $p \in \{0.15, 0.30, 0.45\}$. From each (n, p) combination, 800 graphs were sampled.

The experiments were conducted on an NVIDIA DGX-1 equipped with a Tesla Pascal P100 GPU (P100-SXM2-16GB), with 3584 CUDA cores. The system also featured an Intel Xeon E5-2698 v4 CPU with 512 GB of RAM.

Baseline

As a means to have a baseline for comparison, we implemented a greedy algorithm that builds dominating sets based on the Domination Potential attribute. Starting from an empty set S, the procedure chooses the node with the largest domination potential at each step and adds it to the set S. The procedure stops when the set S dominates the whole graph. This procedure is formalized in Algorithm 4.

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br

Algorithm 4 Greedy DS

Input: Graph G
Output: Dominating set S

$S \leftarrow \{\}$
while not IS_DS(G, S) **do** ▷ IS_DS returns true if S is a DS of G
    $v^* \leftarrow \arg\max\{DP(v) \mid \forall v \in V \setminus S\}$
    $S \leftarrow S \cup \{v^*\}$
end while
return S

6. Results

Table 1 shows the results found on our test instances. To compute the average apx-ratios, the same formula as presented in the previous section was used, only replacing the numerator of the fraction $\frac{|S_{DQN}(G)|}{|S^*(G)|}$ and keeping the denominator.

n	p	avg. $\lvert\mathbf{S_{DQN}}\rvert$	avg. $\lvert\mathbf{S^*}\rvert$	DQN apx-ratio	greedy apx-ratio	hybrid apx-ratio
10	0.15	4.11	3.78	1.090	1.058	**1.003**
20	0.15	6.84	5.94	1.161	1.138	**1.030**
30	0.15	9.29	6.84	1.367	1.146	**1.086**
40	0.15	13.72	7.31	1.874	1.157	**1.147**
50	0.15	14.33	7.76	1.850	**1.160**	**1.160**
10	0.30	3.63	3.03	1.216	1.106	**1.023**
20	0.30	5.34	3.71	1.454	1.136	**1.079**
30	0.30	7.04	4.08	1.727	**1.120**	1.123
40	0.30	8.23	4.33	1.908	**1.127**	1.173
50	0.30	9.14	4.69	1.958	**1.099**	1.147
10	0.45	2.72	2.28	1.202	1.111	**1.026**
20	0.45	4.08	2.81	1.462	1.075	**1.050**
30	0.45	4.92	3.00	1.639	**1.072**	**1.072**
40	0.45	5.28	3.04	1.739	**1.130**	1.131
50	0.45	6.16	3.21	1.936	**1.155**	1.200

Table 1: Results of the proposed method. From left to right, each column shows: the number of nodes, the graph density, the average solution size given by the DQN, the average solution size given by the Gurobi optimizer, the approximation ratio for the DQN solutions, the approximation ratio for the greedy algorithm, and the approximation ratio for the hybrid approach (DQN with local search).

Discussion

In general, considering the results shown in the column **DQN apx-ratio**, the DQN alone could not learn heuristics that made solutions as small as the ones provided by the Gurobi Optimizer. The best performance was found for the $G(10, 0.15)$ distribution, where it achieved a 1.090 approximation ratio. Despite that, it was capable of keeping the approximation ratio below 2.0 on all of the test instances. The local search procedure added a great advantage to the DQN performance, as can be seen in column **hybrid apx-ratio** it generated better or equivalent approximation ratios w.r.t. the greedy procedure for the $G(10, 0.15)$, $G(20, 0.15)$, $G(30, 0.15)$, $G(40, 0.15)$, $G(50, 0.15)$, $G(10, 0.30)$, $G(20, 0.30)$, $G(10, 0.45)$, $G(20, 0.45)$, $G(30, 0.45)$ distributions ($\sim 67\%$).

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br

Those results indicate that the network finds it difficult to learn all the different combinatorial patterns exhibited in the solutions to the MDS problem. This indicates that the RL approach alone still needs refinement, maybe using more robust and refined models or different architectures that better suit the problem's combinatorial nature. Also, we can verify that hybrid machine learning with the local search method appears to be a promising approach to tackling combinatorial problems, despite also needing more advances to make them robust for larger and more complex problem instances.

7. Conclusions

In this work, we investigated the feasibility of applying the Deep Q-Learning (DQL) framework, combined with a local search refinement step, to approximate solutions for the Minimum Dominating Set (MDS) problem. Our results demonstrate that, although the DQL model alone struggles to consistently produce high-quality solutions, its performance can be significantly improved when complemented with a simple yet effective local search procedure. This hybrid approach proved to be competitive with a greedy baseline in approximately 67% of the test cases by varying graph sizes and densities, suggesting its potential as a viable method for small-scale instances.

Despite these promising results, the study presents some limitations. Most notably, we restricted our evaluation to synthetic graphs sampled from an Erdős–Rényi distribution, which may not capture the structural complexity of real-world graphs. Furthermore, the model's generalization capabilities remain unexplored, as no out-of-distribution testing was conducted.

As future work, we propose several directions to strengthen and extend this line of research. These include incorporating ablation studies to better understand the impact of different components (e.g., GNN layers and node attributes), exploring enhancements to the RL process tailored to the combinatorial nature of MDS, and expanding the evaluation to include real-world datasets and diverse graph topologies. We believe that with further refinement, hybrid learning methods such as the one proposed here can contribute meaningfully to the broader landscape of combinatorial optimization.

References

Chen, M., Liu, S., and He, W. (2024). Learn to solve dominating set problem with GNN and reinforcement learning. *Appl. Math. Comput.*, 474:128717. URL https://doi.org/10.1016/j.amc.2024.128717.

Crescenzi, P., Kann, V., Halldórsson, M., Karpinski, M., and Woeginger, G. (2000). Minimum dominating set. *A Compendium of NP Optimization Problems*.

Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., and Song, L. (2017). Learning combinatorial optimization algorithms over graphs. *NIPS*. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/d9896106ca98d3d05b8cbdf4fd8b13a1-Paper.pdf.

Garey, D. S., Michael R.; Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness. Series of Books in the Mathematical Sciences (1st ed.).* W. H. Freeman and Company., New York.

Hedetniemi, S. and Laskar, R. (1990). Bibliography on domination in graphs and some basic definitions of domination parameters. *Discrete Mathematics*, 86(1):257–277. ISSN

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br

0012-365X. URL https://www.sciencedirect.com/science/article/pii/0012365X90903650.

Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks. *ICLR 2017*. URL https://arxiv.org/abs/1609.02907.

Mnih, V., Kavukcuoglu, K., Silver, D., and et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–533. URL https://www.nature.com/articles/nature14236.

Papadimitriou, C. H. and Steiglitz, K. (1998). *Combinatorial optimization: algorithms and complexity*. Courier Corporation.

Paschos, V. T. (2010). *Applications of Combinatorial Optimization*. ISTE Ltd and John Wiley & Sons, Great Britain and the United States.

Peng, Y., Choi, B., and Xu, J. (2021). Graph learning for combinatorial optimization: A survey of state-of-the-art. *Data Sci. Eng.*, 6(2):119–141. URL https://doi.org/10.1007/s41019-021-00155-3.

Rani, P., J.; Mor (2019). Domination in graph and some of its applications in various fields. *International Journal of Emerging Technologies and Innovative Research*, 6:322–326.

Sato, R., Yamada, M., and Kashima, H. (2019). Approximation ratios of graph neural networks for combinatorial problems. *NeurIPS 2019*. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/635440afdfc39fe37995fed127d7df4f-Paper.pdf.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80.

Vinyals, O., Fortunato, M., and Jaitly, N. (2017). Pointer networks. *NIPS*. URL https://proceedings.neurips.cc/paper_files/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2019). How powerful are graph neural networks? *ICLR*. URL https://arxiv.org/abs/1810.00826.

Zhang, D., Dai, H., Malkin, N., Courville, A., Bengio, Y., and Pan, L. (2023). Let the flows tell: Solving graph combinatorial optimization problems with gflownets. *NeurIPS*. URL https://arxiv.org/abs/2305.17010.

DOI: 10.59254/sbpo-2025-212133
https://proceedings.science/p/212133?lang=pt-br