NODE SELECTION FOR SUBDIVISION WITH REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Subdivision is an important method for finding all the real solutions of nonlinear equations in a given box, through a recursive process of contracting and bisecting. In certain applications like constraint satisfaction, it suffices to find a single solution efficiently. This turns out to be a challenging node selection problem when the underlying subdivision tree is huge. In this paper, we propose to employ reinforcement learning to learn an efficient node selection policy for solving parametric nonlinear equations. Under certain assumptions, we prove that there exists an optimal node selection policy solely based on selecting child nodes. We train a Double Deep Q-Network by exploring the node selection space at many different parameter values offline, and integrate the learned policy into a state-of-the-art subdivision solver. During the online phase, to efficiently find one solution of the system at new values of parameters, we apply the learned policy to select promising child nodes and backtrack when necessary based on maximum Q-value estimates to continue the search. Experiments on several parametric systems show that our method significantly outperforms default heuristic strategies.

1 INTRODUCTION

Subdivision methods (Chabert & Jaulin, 2009) provide a reliable approach for finding all the real solutions of a system of zero-dimensional nonlinear equations. Given a system f and an initial box B_0, it repeatedly applies operators to contract the box into smaller ones, bisects the box and prunes the subboxes not containing solutions until all the regions are rejected or small boxes enclosing all the solutions are obtained; when certification operator succeeds, different solutions can be isolated. In many applications, however, computing all the solutions is not necessary. For instance, in constraint satisfaction problems, one may only need to find a feasible solution satisfying all constraints (Kumar, 1992). In such cases, it is desirable to find one solution as quickly as possible whenever a solution exists. This problem falls in the broader category of tree search (Russell & Norvig, 2020), for which a variety of heuristic and learning-based methods have been developed for different applications (Zhang et al., 2026; Gasse et al., 2019; Cappart et al., 2023).

The tree search problem aims to construct and explore a search tree efficiently in order to reach a goal node while avoiding unnecessary exploration. Classical strategies include depth-first search, breadth-first search, and best-first search, while more recent studies have also considered learning-based methods for guiding the search process (Zhao et al., 2026). In combinatorial optimization, branch-and-bound (B&B) is a representative tree-search framework, in which branching-variable selection determines how child nodes are generated, while node selection determines the order in which active nodes are explored (Lodi & Zarpellon, 2017; Scavuzzo et al., 2026).

There have been a few works on node selection for B&B framework, in particular aiming for solving mixed inter linear programming (MILP) problems. The work by He et al. (2014) proposed an imitation learning based method for node selection and pruning. The power of the method is largely attributed to employ an oracle knowing the optimal solution as an expert, which however also limits its applicability on solving large scale problems. To transfer the ability learned on small problems to larger ones, the work of Song et al. (2018) proposed retrospective imitation learning based on improving initial search tree by tracing backwards from the terminal states. Instead of learning to select from a list of unexplored frontier nodes, Yilmaz & Yorke-Smith (2021) employed imitation learning to select only the children nodes and relied on a separate backtrack strategy for exact solving. MILP

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

problem instances can be encoded by bipartite graphs and one can employ graph neural network to automatically learn their representation instead of manually designing features. This representation has been widely used for branching variable selection for MILP (Gasse et al., 2019). Based on this representation, the work by Labassi et al. (2022) handled the node selection problem by learning to compare two different nodes. The performance of imitation learning strongly relies on the optimal solution information of the training instances, which is difficult to be obtained for larger size problems. For this reason, there have been alternative attempts to apply reinforcement learning for the node selection problem (Mattick & Mutschler, 2024; Zhang et al., 2025). For both work, the state is treated as a dynamically growing B&B tree and the action is to select a node from unexplored leaf nodes. Although this strategy has the potential to select the best node, in practice, the effectiveness of both approaches are limited by the complex state space and a large dynamic action space. The work by Zhang et al. (2025) heuristically restricted the action space to a small candidate set.

In this work, we propose a reinforcement learning based approach targeting on solving parametric nonlinear equations by subdivision methods. Given a parametric nonlinear system $\boldsymbol{f}(\boldsymbol{u}, \boldsymbol{x}) = 0$ and an initial box B_0, the goal is to learn a node selection policy π such that for any given value of parameters u, one solution of $\boldsymbol{f}(u, \boldsymbol{x}) = 0$ inside B_0 can be isolated as fast as possible by π. We emphasize that it is worth paying a fixed cost for offline training as a well learned policy may benefit the solving for infinitely many new parameter values in the online phase. To achieve this, we reset the parameters $\boldsymbol{u}$ to different values and at each value u, train an agent to autonomously construct a path by subdivision method from the initial box B_0 to a sufficiently small leaf box $B_T(u)$, inside which we can test if $\boldsymbol{f}(u, \boldsymbol{x})$ has exactly one solution. During such a process, the agent is penalized for each step it takes towards the leaf box and receives a larger penalty (resp. nonnegative award) if it cannot find (resp. can find) an isolated solution in the leaf box.

Note that unlike the B&B method for MILP, there are no bounds associated with different nodes to be compared to help making the decision. Instead, the solver needs to identify promising subbox early, even though it may take several subsequent subdivision steps to determine whether a subbox contains a solution. Therefore, unlike some learning-based MILP approaches that encode the search tree in the state, we use the current box together with the parameter values as the state, providing a more natural local representation. Compared with MILP, subdivision branches are largely independent once the solver rules are fixed, making it more natural to continue along a promising path rather than repeatedly switching among frontier nodes. The unselected siblings are retained for backtracking, so that another branch can be explored if the current path terminates without finding a solution. As a result, for our problem, learning a policy to directly select a child and properly backtrack is more suitable. Indeed, we prove that a shortest path towards a solution can be found in this way if it exists based on the theory of Bellman equation.

More precisely, we make the following three contributions:

- (i) We formulate the node selection for subdivision as a deterministic Markov decision process (MDP) and prove that an optimal policy exists solely based on selecting children nodes.
- (ii) We propose a double DQN based offline-online framework for learning node selection to accelerate the search of the first solutions of parametric nonlinear equations.
- (iii) We integrate the learned policy into a state-of-the-art subdivision solver and show that it outperforms the default search strategies of the solver.

2 Background on subdivision methods

Let $B_{\boldsymbol{u}} \subset \mathbb{R}^m$ and $B_0 \subset \mathbb{R}^n$ be bounded boxes. Let $\boldsymbol{f} = (\boldsymbol{f}_1, \ldots, \boldsymbol{f}_n) : B_{\boldsymbol{u}} \times B_0 \to \mathbb{R}^n$ be a continuously differentiable function admitting computable interval extension. We assume that for any $u \in B_{\boldsymbol{u}}$, the system $\boldsymbol{f}(u, \boldsymbol{x}) = \mathbf{0}$ has only finitely many real solutions contained in B_0. The goal in this paper is to quickly find one such real solution if it exists by subdivision methods. Algorithm 1 provides an overview of the subdivision framework.

At each iteration of Algorithm 1, one box is selected from the working list $\mathcal{W}$ for further processing. The rule used for this selection determines the order in which the subdivision tree is explored. When all solutions are required, the solver must eventually process all regions that cannot be discarded. When only one solution is sought, however, the order of exploration becomes particularly important, since the computation terminates as soon as one certified solution boxes has been obtained. A

suitable node-selection strategy can therefore substantially reduce the number of boxes that need to be processed before termination. Here we search for only one solution and terminate as soon as a certified solution box is found.

Algorithm 1: General Subdivision Framework

Input: An initial box $B_0 \subseteq \mathbb{R}^n$, the system $\boldsymbol{f}(u, \boldsymbol{x}) = 0$ and a box-width tolerance $\epsilon > 0$.
Output: A certified solution box B (with width $\leq \epsilon$ and containing exactly one real solution) contained in B_0, or a list of unresolved boxes $\mathcal{O}$ if no certified solution is found.

```
W ← {B_0}; O ← ∅;
while W ≠ ∅ do
    select and remove a box B from W; Contract B;
    if B ≠ ∅ then
        if the width of B is no greater than ε then
            if B is certified to contain exactly one zero of f then return B; else add B into O;
        else
            bisect B into two subboxes B⁻ and B⁺; add B⁻ and B⁺ to W;
return O;
```

3 METHOD

Let $\boldsymbol{f}(\boldsymbol{u}, \boldsymbol{x})$ be the given parametric system. Given a vector of parameter values u, let $\boldsymbol{f}(u, \boldsymbol{x})$ be the specialized system at u. Let B_0 be the initial box. Let $\mathcal{SD}$ be a solver based on subdivision method. Let ϵ (with the default value 10^{-6}) be a given threshold. The goal of subdivision is run $\mathcal{SD}$ to find all zeros of f in B_0 such that each zero is contained in a subbox of B_0 of width $\leq \epsilon$.

The solving process naturally induces a binary tree T whose leaf nodes are subboxes of B_0. Throughout this paper, each node in T corresponds to a box after contraction. In particular, the root of T corresponds to the box obtained by contracting B_0. A node B is a leaf if $B = \emptyset$ or B has width $\leq \epsilon$. An empty leaf is called a dead leaf. For a leaf $B \neq \emptyset$, if B is certified to contain exactly one zero of f, it is called a solution leaf; otherwise, it is called an unknown leaf. Notice that any zero of f in B_0 is contained in a solution leaf or an unknown leaf.

The goal of this work is to learn a policy that finds a solution leaf using as few subdivision steps as possible. If a solution exists and can be isolated by the subdivision method, the ideal policy should generate a shortest path from the root to a solution leaf. In practice, as it is difficult to make every node selection optimal, one would expect to generate a subtree as small as possible. Fig. 1 provides an overview of the framework. Here we use Ibex (more precisely, IbexSolve) as the default solver, which provides contraction, bisection, and solution certification operators (Chabert & Jaulin, 2009).

3.1 A THEORETICALLY OPTIMAL NODE SELECTION POLICY

In this subsection, we provide a theoretical solution to the node selection problem for subdivision. We first recall some necessary background theory of Bellman equation and then state our main theoretical results in Theorem 1 and Theorem 2.

3.1.1 BACKGROUND ON THE THEORY OF BELLMAN EQUATION

In this work, we consider a deterministic MDP defined by $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the deterministic transition function, i.e., $s_{t+1} = \mathcal{T}(s_t, a_t)$, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the deterministic reward function, i.e., $r_t = \mathcal{R}(s_t, a_t)$, and $\gamma \in (0, 1)$ is the discount factor. We emphasize that the Markov property implies that s_{t+1} and r_t depend only on the current state s_t and action a_t, not on the historical states.

Let $\pi(a|s)$, or simply π, be a policy defining the probability distribution over the action space $\mathcal{A}$ given a state s. For a policy π, the *state-value function* $V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,\middle|\, s_t = s \right]$ is the expected discounted return starting from state s and following π thereafter. The *action-value function* $Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,\middle|\, s_t = s, a_t = a \right]$ is the expected discounted return starting from state s, taking action a, and then following π. Note that we have $V^\pi(s) = \sum_a \pi(a|s) Q^\pi(s, a)$.

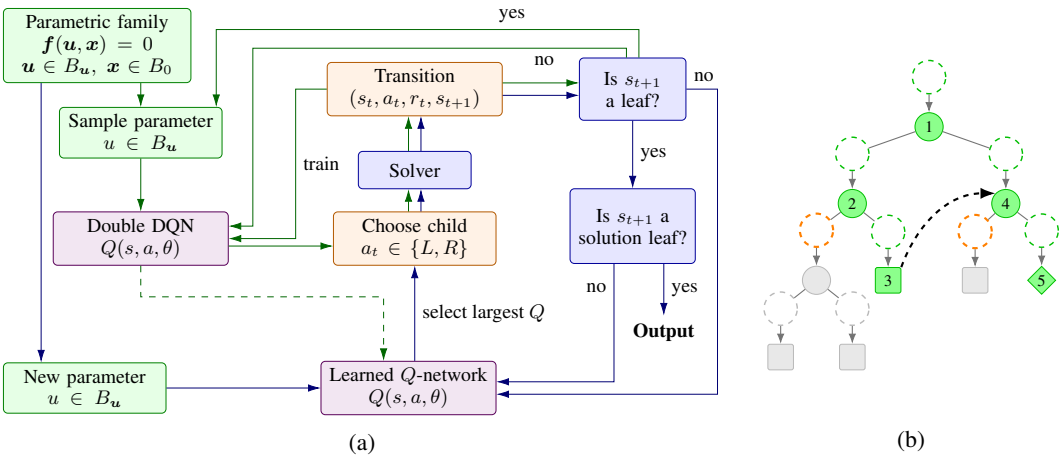


Figure 1: (a) Overview of the offline training and online search process. Green arrows indicates offline training while blue indicates online search. (b) Illustration of the online search process. A diamond denotes a solution leaf, while a square denotes a non-solution (dead or unknown) leaf. A dashed circle represents a box before contraction, which is not considered as a tree node. Green, gray, and orange indicate explored, unseen, and retained unexplored boxes, respectively. The numbers indicate the order in which nodes are explored.

Given two policies π_1 and π_2, we say that π_1 is better than π_2, if for any state $s \in \mathcal{S}$, we have $V^{\pi_1}(s) \geq V^{\pi_2}(s)$. A policy π^* is called *optimal* if π^* is better than any other policy π. The state-value function can be recursively defined by the Bellman equation:

$$V^{\pi}(s) = \sum_a \pi(a|s) \left(\mathcal{R}(s,a) + \gamma V^{\pi}(\mathcal{T}(s,a))\right).$$

Let $V^*(s) = \max_\pi V^\pi(s)$ and $Q^*(s,a) = \max_\pi Q^\pi(s,a)$. The theory of Bellman equation reveals that the optimal policy π^* exists, that is

$$V^{\pi^*}(s) = V^*(s) = \max_a \left(\mathcal{R}(s,a) + \gamma V^*(\mathcal{T}(s,a))\right) = \max_a Q^*(s,a). \tag{1}$$

Moreover, let $a^*(s) = \arg\max_a Q^*(s,a)$ and define $\pi(a|s) = 1$ if $a = a^*(s)$ and $\pi(a|s) = 0$ otherwise. Then such a deterministic policy is the optimal policy π^*.

Since the action-value function satisfies $Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma V^\pi(\mathcal{T}(s,a))$, combining with Eqs (1), one deduces that

$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma V^*(\mathcal{T}(s,a)) = \mathcal{R}(s,a) + \gamma \max_{a'} Q^*(\mathcal{T}(s,a), a'). \tag{2}$$

3.1.2 Main Result

Suppose that starting from a state-action pair (s_1, a_1) (resp. (s_2, a_2)), we apply the deterministic policy π_1 (resp. π_2) until reaching a termination state. Suppose that the trajectory consists of $N_1 \geq 1$ (resp. $N_2 \geq 1$) steps. Assume further that the intermediate reward is r°, the termination rewards are respectively r_1 and r_2. The awards r_1 and r_2 are only allowed to take values from $\{r^+, r^-\}$, where r^+ (resp. r^-) represents the award (penalty). For the awards, we assume that $r^+ \geq 0 > r^\circ > r^-$ or $r^+ > 0 = r^\circ > r^-$.

Lemma 1. *Under the above setting, $Q^{\pi_1}(s_1,a_1) \geq Q^{\pi_2}(s_2,a_2)$ if and only if*

$$\gamma^{N_1-1}\left(r_1 - \frac{r^\circ}{1-\gamma}\right) \geq \gamma^{N_2-1}\left(r_2 - \frac{r^\circ}{1-\gamma}\right). \tag{3}$$

Let $c_i = r_i - \frac{r^\circ}{1-\gamma}$ and $c^- = r^- - \frac{r^\circ}{1-\gamma}$. We further have the following properties depending on the signs of r_1 and r_2.

(*i*) *When $r_1 \geq 0$ and $r_2 < 0$, Eq. (3) holds for any N_1 and N_2 if and only if $c^- \leq 0$.*
(*ii*) *When $r_1 \geq 0$ and $r_2 \geq 0$, Eq. (3) holds if and only if $N_1 \leq N_2$.*
(*iii*) *When $r_1 < 0$ and $r_2 \geq 0$, Eq. (3) does not hold for any N_1 and N_2 if and only if $c^- \leq 0$.*
(*iv*) *When $r_1 < 0$ and $r_2 < 0$, if $c^- = 0$, then Eq. (3) always holds; if $c^- > 0$, Eq. (3) holds if and only if $N_1 \leq N_2$; if $c^- < 0$, Eq. (3) holds if and only if $N_1 \geq N_2$.*

Proof. To prove Eq. (3), it suffices to observe that for (s_i, a_i), $i = 1, 2$,

$$Q^{\pi_i}(s_i, a_i) = \sum_{k=0}^{N_i-2} \gamma^k r^\circ + \gamma^{N_i-1} r_i = r^\circ \frac{1-\gamma^{N_i-1}}{1-\gamma} + \gamma^{N_i-1} r_i. \tag{4}$$

We next prove the four properties one by one.

For (i), if $c^- \leq 0$, as $c_2 = c^-$ and the left hand side is positive, Eq. (3) always holds. For another direction, if Eq. (3) holds for any N_1 and N_2, then we must have $r_2 - \frac{r^\circ}{1-\gamma} \leq 0$. Otherwise, Eq. (3) holds if and only if $N_1 - N_2 \leq ln(\frac{c_2}{c_1})/ln(\gamma)$. We can choose $N_1 - N_2 > ln(\frac{c_2}{c_1})/ln(\gamma)$ to make Eq. (3) not hold, contradiction. For (ii), it holds as both c_1 and c_2 are positive. For (iii), if $c_1 = c^- \leq 0$, Eq. (3) cannot be satisfied. For another direction, if Eq. (3) does not hold, we want to show that $c_1 \leq 0$. Otherwise, if $c_1 > 0$, based on the same arguments as (i), Eq. (3) holds if and only if $N_1 - N_2 \leq ln(\frac{c_2}{c_1})/ln(\gamma)$, whose solution set is not empty, contradiction. For (iv), it holds following the same argument as (ii). □

Remark 1. *This lemma suggests that if we choose the policy based on exact Q-values, when we set the value of penalties, it is recommended to make them satisfy $c^- \leq 0$.*

Next, we formulate the node selection in solving parametric systems by subdivision method as a deterministic MDP. Let $\mathcal{C}$ be the default contraction operator, which maps a box B into another box B' such that $B' \subseteq B$. For a given box $B = [\underline{x_1}, \overline{x_1}] \times \cdots \times [\underline{x_n}, \overline{x_n}]$, the bisection strategy chooses a branching variable $\boldsymbol{x}_i$ and divides B into two parts $B_L = [\underline{x_1}, \overline{x_1}] \times \cdots \times [\underline{x_i}, \frac{\underline{x_i}+\overline{x_i}}{2}] \times \cdots \times [\underline{x_n}, \overline{x_n}]$ and $B_R = [\underline{x_1}, \overline{x_1}] \times \cdots \times [\frac{\underline{x_i}+\overline{x_i}}{2}, \overline{x_i}] \times \cdots \times [\underline{x_n}, \overline{x_n}]$. Each state $s \in \mathcal{S}$ is represented by $s = (u, B)$, where $u \in B_{\boldsymbol{u}}$ and $B \subseteq B_0$ is the contracted box to be bisected at the current step. For every nonterminal state, the available action space is $\mathcal{A} = \{L, R\}$. For a state $s = (u, B)$ and an action $a \in \{L, R\}$, the box B is first bisected into B_L and B_R. The transition function $\mathcal{T}$ maps (s, a) to the next state $s' = (u, B')$, where $B' = \mathcal{C}(B_a)$. If $B' = \emptyset$, then B' is a dead leaf. Otherwise, we determine the type of B' based on the width of B' and the certification operator. If B' is not a leaf node, we set $\mathcal{R}(s, a) = r^\circ$. If B' is a solution leaf, we set $\mathcal{R}(s, a) = r^+$. Otherwise, that is s' is a dead leaf or unknown leaf, we set $\mathcal{R}(s, a) = r^-$. By the theory of Bellman equation, an optimal node selection π^* exists.

Theorem 1. *Assume that $c^- \leq 0$ and the tree T has at least one solution leaf. Then every optimal policy π^* generates a shortest path from the root to a solution leaf.*

Proof. Let $\pi_1 = \pi^*$ and π_2 be any other deterministic policy which finds a solution. Let s_0 be the state corresponding to the root node. Let $a^* \in \{L, R\}$ be the action such that $V^*(s_0) = \max_a Q^*(s_0, a) = Q^*(s_0, a^*) = Q^{\pi^*}(s_0, a^*)$. Let a be action taken by π_2 starting from s_0. By the definition of π^*, we have $Q^{\pi_1}(s_0, a^*) \geq Q^{\pi_2}(s_0, a)$. Since $r_2 \geq 0$, by (iii) of Lemma 1, we must have $r_1 \geq 0$. By (ii) of the same lemma, we must have $N_1 \leq N_2$. Since π_2 is any deterministic policy which finds a solution, π^* generates a shortest path from the root to a solution leaf. □

In Theorem 1, we assume that there exists an oracle policy which can always make the right decision. As a result, if there exist solutions, it suffices to follow a path to find one. In practice, when applying a policy to select a node until reaching a leaf, the leaf node may not be a solution leaf. When this occurs, we need a backtrack strategy to keep exploring the tree. More precisely, we have the following incremental procedure to build a subdivision tree T. (1) Base case: starting from the root, applying a policy $\pi^{(1)}$ to obtain a path ending with a solution/unknown/dead leaf; (2) Recursive step: assume that a tree T' has been constructed by applying the policies $\pi^{(1)}, \ldots, \pi^{(p-1)}$. If T' is not a complete subdivision tree, that is, there exists at least one internal node s such that at least one child node $s' = \mathcal{T}(s, a)$ has not been explored. Let SA be the set of all such state-action pairs.

Let $(s^*, a^*) = \arg\max_{(s,a)\in SA} Q^*(s,a)$. Starting from (s^*, a^*), we conduct state transitions by choosing the best policy $\pi^{(p)}$ associated with (s^*, a^*), until a leaf node is met; (3) Termination: If one solution is found or T is a complete subdivision tree, we are done.

Theorem 2. *Assuming that $c^- \leq 0$ and the complete subdivision tree T has at least one solution leaf. Let T' be a partial tree containing no solution leaves, and SA the set of all unexplored state-action pairs. Choose $(s^*, a^*) = \arg\max_{(s,a)\in SA} Q^*(s,a)$, and let π_1 be an optimal policy from (s^*, a^*). Let π_2 be any policy starting from a pair $(s_2, a_2) \in SA$ that reaches a solution leaf after N_2 steps. Then π_1 finds a solution leaf in $N_1 \leq N_2$ steps.*

Proof. Since π_1 is optimal from (s^*, a^*), we have $Q^{\pi_1}(s^*, a^*) = Q^*(s^*, a^*)$. By the definition of (s^*, a^*), $Q^*(s^*, a^*) \geq Q^*(s_2, a_2)$. Moreover, since π_2 is an arbitrary policy starting from (s_2, a_2), $Q^*(s_2, a_2) \geq Q^{\pi_2}(s_2, a_2)$. Therefore, $Q^{\pi_1}(s^*, a^*) \geq Q^{\pi_2}(s_2, a_2)$. Since $r_2 \geq 0$, by (iii) of Lemma 1, we must have $r_1 \geq 0$. By (ii) of the same lemma, we must have $N_1 \leq N_2$. □

The theorem implies that, among all unexplored branches that can lead to a solution, starting from (s^*, a^*) and following an optimal policy yields a shortest path to a solution.

3.2 A REINFORCEMENT LEARNING APPROACH FOR NODE SELECTION

From Eq. (2), we define the temporal difference (TD) for action-value function, which is the one of the most important notions in reinforcement learning.

$$\delta_t = \mathcal{R}(s,a) + \gamma \max_{a'} Q(\mathcal{T}(s,a), a') - Q(s,a). \tag{5}$$

In practice, we would use a deep neural network to approximate the function Q and the training loss is usually based on minimizing the TD error δ_t^2. That is, although our ultimate goal of deep Q-learning is to approximate Q^*, the intermediate goal is to reduce the TD error.

In particular, let $s' = \mathcal{T}(s,a)$, for the method based on double deep Q-learning network (double DQN), the training loss is based on the following TD error:

$$\delta_t = \mathcal{R}(s,a) + \gamma Q\left(s', \arg\max_{a'} Q(s', a', \theta), \theta^-\right) - Q(s,a,\theta), \tag{6}$$

where θ and θ^- are respectively the weights of the online and target network.

Algorithm 2 precisely describes the training steps of our framework RL4NS to learn node selection for subdivision methods based on double DQN. In the implementation, the training of the Q-network and the prediction of the action is conducted in Python. The rest is implemented in C++ by modifying the code of Ibex to add RL-based node selection strategy. During training, the inter-process communication between C++ and Python is conducted through pipe. For online deployment, to avoid the communication overhead, a purely C++ implementation is provided. Next, we provide a brief description on the state representation, Q-network and the setting of a few important hyperparameters. More implementation details can be found in Appendix A.

Representation of the state. As defined in Section 3, each state s is a pair (B, u). Here $u = (u_1, \ldots, u_d)$ and $B = [\underline{x_1}, \overline{x_1}] \times \cdots \times [\underline{x_n}, \overline{x_n}]$. So s is represented by a $2n + d$-dimensional real vector $(\underline{x_1}, \overline{x_1}, \ldots, \underline{x_n}, \overline{x_n}, u_1, \ldots, u_d)$. If the initial box B_0 is large, for numerical stability, the transformation $x \to \text{sign}(x)\log(|x|)$ will be applied to each endpoint of the box $B \subseteq B_0$, where $B \neq \emptyset$, associated with a state s. If $B = \emptyset$, then $s = (0, \ldots, 0, u_1, \ldots, u_d)$.

Q-network. The Q-network is implemented by a multi-layer perceptron, consisting of two hidden layers of dimension 128 with ReLU activation function, and an output layer of dimension 2. LayerNorm is applied to the first hidden layer.

The values of key hyperparameters. Remark 1 suggests that we set $c^- = r^- - \frac{r^\circ}{1-\gamma} \leq 0$. In the experiments, we choose $\gamma = 0.99, r^\circ = -0.1, r^- = -15$ and $r^+ = 10$ to make $c^- < 0$ and we briefly discuss the reason in Appendix B. A complete setting of the default values of hyperparameters is provided in Table 4 of Appendix A.

Algorithm 2: RL4NS: Learning Node Selection for Subdivision Methods via Double DQN

Input: The parametric system $\boldsymbol{f}(\boldsymbol{u}, \boldsymbol{x}) = 0$; the initial box B_0; a set of parametric values $\mathcal{U}$ for training; a neural network Q initialized with random weights θ.

Output: Learned weights θ of Q.

initialize target network parameters $\theta^- \leftarrow \theta$;
initialize local buffer $\mathcal{D}_\ell$ and replay buffer $\mathcal{D}_r$; set $k \leftarrow 0$, $\varepsilon \leftarrow 1$;
for $e = 0, \ldots, M-1$ **do**
  let $u \leftarrow \mathcal{U}_{e\%|\mathcal{U}|}$; let $\overline{B_0}$ be a contracted subbox of B_0 w.r.t. $\boldsymbol{f}(u, \boldsymbol{x})$;
  if $\overline{B_0}$ *is a leaf node* **then** next;
  Initialize state $s_0 = (u, \overline{B_0})$; $t \leftarrow 0$;
  while $t \geq 0$ **do**
    let $r \sim \text{Uniform}(0, 1)$;
    if $r \leq \varepsilon$ **then** $a_t \sim \text{Uniform}\{L, R\}$ **else** $a_t \leftarrow \arg\max_a Q(s_t, a; \theta)$;
    bisect the box $B := \overline{B_t}$ into B_L and B_R;
    let $\overline{B_{a_t}}$ be a contracted subbox of B_{a_t} w.r.t. $\boldsymbol{f}(u, \boldsymbol{x})$;
    determine the node type of $\overline{B_{a_t}}$ and obtain corresponding reward r_t;
    set $s_{t+1} = (u, \overline{B_{a_t}})$; store transition (s_t, a_t, r_t, s_{t+1}) in local buffer $\mathcal{D}_\ell$;
    if $\overline{B_{a_t}}$ *is a leaf node* **then** break;
    $t := t + 1$;
  end
  if $|\mathcal{D}_\ell| \geq b_\ell$ **then**
    copy data in $\mathcal{D}_\ell$ into the replay buffer $\mathcal{D}_r$ and then set $\mathcal{D}_\ell := \emptyset$;
    if $b_\ell \geq b$ **then**
      sample a mini-batch of b transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}_r$;
      for *each transition j in the mini-batch* **do**
        set $y_j \leftarrow r_j$ if s_{j+1} is a leaf state and otherwise set $y_j \leftarrow r_j + \gamma\, Q(s_{j+1}, \arg\max_{a'} Q(s_{j+1}, a'; \theta); \theta^-)$;
      end
      for $i = 1, \ldots, K$ **do**
        compute the loss $L(\theta) = \frac{1}{b} \sum_{j=1}^{b} \left(y_j - Q(s_j, a_j; \theta)\right)^2$;
        update θ using gradient descent: $\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$;
        $k \leftarrow k + 1$; **if** $k \bmod C = 0$ **then** $\theta^- \leftarrow \theta$; $k \leftarrow 0$;
      end
    end
  end
  $\varepsilon \leftarrow \max(\varepsilon * c_e, \varepsilon_M)$;
end
return θ

4 EXPERIMENTATION

In this section, we report on the experimental results on a family of parametric systems.

Experimentation platform. Two machines have been used to conduct the experimentation. The training was conducted on a Ubuntu 24.04 server (Intel Core Ultra 7 265K CPU, 32 GB MEM; NVIDIA GeForce RTX 5080 GPU, 16 GB GDDR7), installed with Python 3.10.0, PyTorch 2.11, Ibex 2.9.1, and gcc 13.3.0. The rest was conducted on a Ubuntu 20.04 server (Intel Core i9-11900K, 128GB MEM), installed with Python 3.11.15, PyTorch 2.7.1, Ibex 2.9.1, and gcc 9.4.0. On both machines, we have modified Ibex to support learning policies or integrating learned policies.

Parametric systems for testing. The Kuramoto model Chen et al. (2019); Coss et al. (2018) is a classic model which describes the synchronization of coupled oscillators with different natural frequencies. It is given by $\frac{d\theta_i}{dt} = \omega_i - \frac{1}{N} \sum_{j=1}^{N} K_{i,j} \sin(\theta_i - \theta_j)$, for $i = 1, \ldots, N$, where N is the number of oscillators, θ_i and ω_i are respectively the phase and natural frequency of oscillator

i, and $K_{i,j}$ is the coupling strength between the i-th and j-th oscillators. We consider the equilibrium equations of the Kuramoto model. Since the system is invariant under a global phase shift, a reference phase is typically fixed by setting $\theta_N = 0$, with the corresponding equilibrium equation omitted. By adopting the polynomial formulation and the common assumption of uniform coupling, $K_{i,j} = 1$, we obtain $\{\omega_i - \frac{1}{N}\sum_{j=1}^{N}(s_i c_j - s_j c_i) = 0, c_i^2 + s_i^2 - 1 = 0 \mid i = 1, \ldots, N-1\}$, with $c_N = 1, s_N = 0$. This is a polynomial system with $2(N-1)$ equations and $2(N-1)$ unknowns. To generate instances, we fix ω_i for $i = 1, \ldots, N-1$ by sampling them independently from a zero-mean Gaussian distribution with standard deviation 0.3, i.e., $\omega_i \sim \mathcal{N}(0, 0.3^2)$. We generate instances for $N = 6, \ldots, 12$. For each N, the training, validation, and test sets contain 7000, 1000, and 1000 instances, respectively.

Evaluation of training. In Fig. 2, we show the reward curve (in blue) of RL during training for the Kuramoto model at $N = 6, 7$. Ten random seeds are used to generate this curve. For each seed, we record the weights of Q-network every 500 episodes and calculate the mean value of total rewards collected for examples in the validation dataset when running the policy learned so far to select child node from the root until reaching a leaf. The reward curve (in red) of random search is also shown in the figure for comparison. It is also generated with 10 random seeds, under each of which the mean value of total rewards for examples in the validation dataset by running random search from the root node to the first leaf node. The figure shows that the agent truly learns a node selection policy.

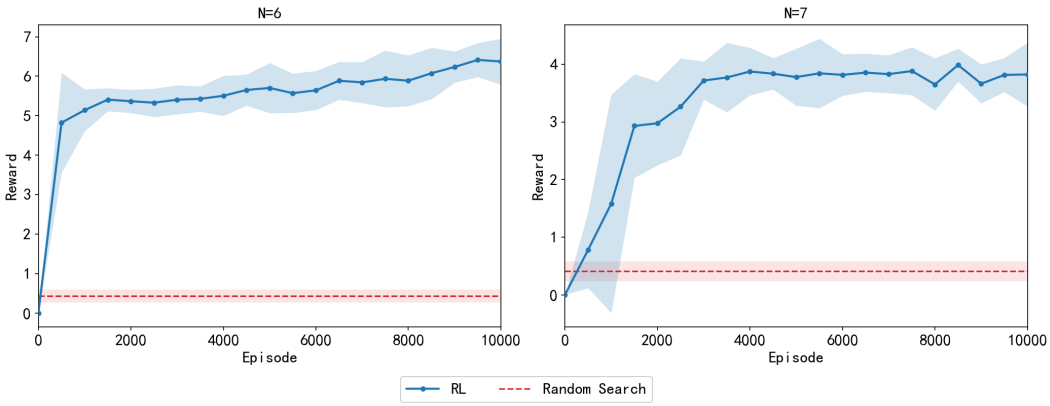


Figure 2: Reward curves for RL and random search.

Comparison with default heuristics of the subdivision solver Ibex. By default, DFS is used for node selection in Ibex. Alternatively, one can choose breadth first search (BFS), which might take more steps to find the first solution than DFS but guarantees that the found solution has the minimum depth. Table 1 compares the performance of RL with default search strategies of Ibex. Table 2 further shows the speedup (in terms of wall time) and the reduction ratio (in terms of visited nodes) of RL over two default heuristic strategies. We have the following observations: (1) as expected, the tree size ($|\text{tree}|$, including both visited and unexplored nodes) of BFS is always larger than that of DFS; the number of visited nodes is larger (resp. smaller) than that of DFS when $N = 6, \ldots, 10$ (resp. $N = 11, 12$); the solving time is longer (resp. shorter) than that of DFS when $N = 6, \ldots, 9$ (resp. $N = 10, 11, 12$); (2) RL outperforms both DFS and BFS in terms of both tree size and the number of visited nodes (the geometric mean of the reduction factor is $7.41\times$) for all N; in addition, as N gets larger ($N > 7$), RL consistently outperforms both BFS and DFS in terms of running time (up to one order of magnitude); (3) in terms of the average depth of solution leaf, for all N, RL is shallower than DFS, and as expected, is deeper than BFS. Here the depth, denoted by $\mathfrak{D}$, counts the number of nodes in the path from the root node to the leaf node.

Evaluation of different methods with no backtracking. In our method, in the offline training phase, the agent selects a left or right child starting from the root node until a leaf node is met. During the online deployment, backtracking is applied if the leaf node is not a solution leaf. It is natural to ask the success rate of the agent without backtracking. In particular, does it perform better than the one based on random policies? Table 3 reports the success rate and the average depth on

Table 1: Experimental results on Kuramoto model for different N.

		N						
METHOD	**METRIC**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Ibex (BFS)	time (s)	0.06	0.19	0.52	1.32	3.32	7.89	19.92
	|tree|	40.78	79.67	157.03	304.18	612.40	1153.84	2256.04
	nodes	24.77	50.01	99.95	193.85	388.79	744.64	1481.28
	$\mathfrak{D}$	**3.1**	**4.1**	**5.2**	**6.1**	**7.1**	**8.1**	**9.1**
Ibex (DFS)	time (s)	**0.04**	**0.13**	0.40	1.19	3.81	11.12	31.72
	|tree|	20.97	40.85	82.93	169.82	398.40	895.23	1991.73
	nodes	15.84	34.35	75.07	160.71	387.90	883.82	1978.93
	$\mathfrak{D}$	6.0	8.5	11.0	13.2	15.4	17.3	19.6
RL	time (s)	0.27	0.54	**0.10**	**0.20**	**0.33**	**2.19**	**5.81**
	|tree|	**12.42**	**18.35**	**22.05**	**28.38**	**31.62**	**156.07**	**311.34**
	nodes	**6.91**	**12.42**	**12.19**	**15.84**	**17.78**	**117.29**	**240.92**
	$\mathfrak{D}$	5.6	7.6	8.7	10.4	12.0	13.2	15.1

Table 2: Speedup and nodes reduction factor of RL over heuristic node selection strategies.

		N							
BASELINE	**METRIC**	6	7	8	9	10	11	12	**GeoMean**
Ibex (BFS)	speedup	0.22	0.35	5.2	6.6	**10.06**	3.60	3.42	**2.29**
	red. factor	3.58	4.02	8.19	12.23	**21.86**	6.34	6.14	**7.41**
Ibex (DFS)	speedup	0.14	0.24	4.0	5.94	**11.54**	5.07	5.45	**2.22**
	red. factor	2.29	2.76	6.15	10.14	**21.81**	7.53	8.21	**6.58**

validation sets. From the table, we observe that: (1) as N grows, it gets more and more difficult for all the approaches to find a solution, which justifies the importance of backtracking; (2) random search has a higher success rate than depth first search; (3) the agent clearly outperforms both DFS and random search even without backtracking.

Table 3: Comparison of different methods without backtracking.

		N						
METHOD	**METRIC**	6	7	8	9	10	11	12
Ibex (DFS)	success rate	8.7	1.8	0.3	0.1	0	0	0
	average depth	5.1	6.5	9.6	10	0	0	0
Random Search	success rate	41.1	24.8	14.3	9	5.7	3.2	1.1
	average depth	5.5	7	8.3	10.1	11.5	13.1	14.4
RL	success rate	**85.1**	**79.3**	**67.6**	**56.0**	**51.5**	**30.1**	**17.7**
	average depth	5.4	7.5	8.4	10.3	11.8	13.5	15.3

5 Conclusion

In this work, we employed reinforcement learning to optimize the node selection policy in subdivision for finding one solution. We proved that an optimal policy exists solely based on selecting children nodes even one starts with non-perfect polices. Based on this theoretical result, we designed a double DQN based framework to efficiently search the first solution. We integrated this policy into a SOTA subdivision solver Ibex. Experiments on several parametric systems show that the learned policy significantly outperforms the default heuristics in Ibex.

AI USE STATEMENT

In this work, we used generative AI tools for polish writing, more precisely for seeking suggestions on phasing specific sentences, but not for generating a paragraph of the manuscript. We used generative AI tools for finding related work, but not for generating the description of the related work or the references. We used generative AI tools for suggestions on technical aspects of figure drawing, not for the conceptual design or content of the figure. We used generative AI to simply some formulas, such as Eq. (4). We have not used generative AI to generate the core theoretical results, such as the four properties of Lemma 1, Remark 1, Theorem 1 and 2 as well as their proofs.

We used generative AI to assist with the implementation of several tasks. Specifically, AI was used to develop the code for aggregating, and storing solving results across multiple instances and methods. The data communication between Python and C (pipe mechanism), i.e., the module that handles requests from the C side, was implemented entirely by AI. The initial integration of RL with Ibex, and the modifications to ibex needed to construct the first working version of the code of the solver, were not AI-assisted. However, subsequent extensions for incorporating additional information during the solving process mainly relied on AI. For the Python code, only the default values for neural network initialization and training hyperparameters were based on AI suggestions; all other parts were not AI-assisted. Generative AI was not used to implement the single-instance solving procedures for various methods (e.g., random search), as well as the tracking information appended to the DFS/BFS solving workflows of Ibex. It was also not used to implement the reinforcement learning (RL) environment, the training procedure and the inference procedure. Other parts of the implementation not mentioned above were developed without AI.

We reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To enhance reproducibility, we have included detailed proofs for our theoretical results in Section 3.1. The offline training and online deployment are illustrated in Fig. 1. In addition, we provide a detailed description of the training procedure in Algorithm 2 and the default values of hyperparameters used in the implementation in Table 4 of Appendix B. Multiple random seeds are used in Fig. 2 of Section 4 and Table 5 of Appendix B to ensure robust conclusions.

REFERENCES

Quentin Cappart, Didier Chételat, Elias B Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.

Gilles Chabert and Luc Jaulin. Contractor programming. *Artificial Intelligence*, 173(11):1079–1100, 2009.

Tianran Chen, Jakub Mareček, Dhagash Mehta, and Matthew Niemerg. Three formulations of the kuramoto model as a system of polynomial equations. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 810–815. IEEE, 2019.

Owen Coss, Jonathan D Hauenstein, Hoon Hong, and Daniel K Molzahn. Locating and counting equilibria of the kuramoto model with rank-one coupling. *SIAM Journal on Applied Algebra and Geometry*, 2(1):45–71, 2018.

Maxime Gasse, Didier Chetelat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32, pp. 15554–15566. Curran Associates, Inc., 2019.

He He, Hal Daumé, and Jason Eisner. Learning to search in branch and bound algorithms. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32–44, 1992.

Abdel Ghani Labassi, Didier Chételat, and Andrea Lodi. Learning to compare nodes in branch and bound with graph neural networks. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 32000–32010. Curran Associates, Inc., 2022.

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

Alexander Julian Mattick and Christopher Mutschler. Reinforcement learning for node selection in branch-and-bound. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, Hoboken, NJ, 4 edition, 2020. ISBN 9780134610993.

Lara Scavuzzo, Karen Aardal, Andrea Lodi, and Neil Yorke-Smith. Machine learning augmented branch and bound for mixed integer linear programming. *Mathematical Programming*, 217(1-2): 123–166, 2026.

Jialin Song, Ravi Lanka, Albert Zhao, Aadyot Bhatnagar, Yisong Yue, and Masahiro Ono. Learning to search via retrospective imitation, 2018. arXiv preprint arXiv:1804.00846.

Kaan Yilmaz and Neil Yorke-Smith. A study of learning search approximation in mixed integer branch and bound: Node selection in scip. *AI*, 2(2):150–178, 2021. ISSN 2673-2688.

Sijia Zhang, Shuli Zeng, Shaoang Li, Feng Wu, and Xiangyang Li. Learning to select nodes in branch and bound with sufficient tree representation. In *The Thirteenth International Conference on Learning Representations*, 2025.

Xuefeng Zhang, Liangyu Chen, Zhengfeng Yang, and Zhenbing Zeng. Learning to select cutting planes in mixed integer linear programming solving. *Expert Systems with Applications*, 298: 129924, 2026. ISSN 0957-4174.

Dengwei Zhao, Shikui Tu, and Lei Xu. Neural-guided heuristic tree search for deep bidirectional decision-making: a survey. *Artificial Intelligence Review*, 2026. ISSN 1573-7462.

A More details on the implementation

Table 4 provides the default values of the hyperparameters used in our implementation.

Table 4: Default values of different hyperparameters.

Hyperparameters	Default Value
Discount factor γ	0.99
Final exploration rate ε_M	0.1
Scaling factor of exploration rate c_e	$\varepsilon_M^{\frac{1}{0.8M}}$
Number of consecutive gradient updates K	8
Target network update frequency C	10
Threshold for copying data from local buffer to replay buffer b_ℓ	64
Replay buffer capacity $\lvert\mathcal{D}_r\rvert$	20000
Mini-batch size b	128
Learning rate α	0.0003
Number of episodes M	10000
Solution found reward r^+	10
Solution missing penalty r^-	-15
Exploration penalty r°	-0.1

B IMPACT OF IMPORTANT HYPERPARAMETERS ON THE PERFORMANCE

Lemma 1 and Remark 1 suggest that in theory, when we design the awards, it is recommended to set $c^- \leq 0$. Table 5 reports the real performance of the policies for different values of c^-. From the table, we see that the effect of hyperparameters is largely offset by the influence of random seeds. Nevertheless, for both values of N, $r^- = -15$ exhibits the smallest standard deviation and is used as the default penalty for the leaf node.

Table 5: Impact of penalties on the performance (tested with 5 random seeds).

γ	r°	r^-	c^-	N	tree size	explored nodes	depth
0.99	-0.1	-5	5	6	13.320 ± 0.958	7.338 ± 0.495	6.9 ± 0.2
0.99	-0.1	-10	0	6	12.922 ± 0.656	7.174 ± 0.354	6.7 ± 0.2
0.99	-0.1	-15	-5	6	13.018 ± 0.356	7.174 ± 0.183	6.8 ± 0.1
0.99	-0.1	-5	5	7	16.332 ± 1.189	9.018 ± 0.596	8.1 ± 0.2
0.99	-0.1	-10	0	7	17.812 ± 1.451	9.760 ± 0.736	8.4 ± 0.2
0.99	-0.1	-15	-5	7	17.110 ± 0.661	9.406 ± 0.335	8.4 ± 0.2