REINFORCEMENT LEARNING APPROACHES FOR COMBINATORIAL OPTIMIZATION SURVEY REPORT

Guntas Singh Saran*
Department of Computer Science and Engineering
Indian Institute of Technology Gandhinagar
Palaj, GJ 382355, India

Prof. Anirban Dasgupta
Department of Computer Science and Engineering
Indian Institute of Technology Gandhinagar
Palaj, GJ 382355, India

1 INTRODUCTION

Combinatorial Optimization (CO) problems are fundamental to a wide range of applications in domains such as logistics, telecommunications, computational biology, and finance. These problems aim to find an optimal solution from a finite set of possibilities, often under constraints. Classical examples include the Traveling Salesman Problem (TSP), the Maximum Cut Problem (Max-Cut), and the Bin Packing Problem (BPP). Many CO problems are NP-hard, meaning that they cannot be solved efficiently using exact algorithms for large-scale instances.

Traditional approaches to CO problems often rely on hand-crafted heuristics and approximation algorithms designed by domain experts. Although these methods provide practical solutions, they may not generalize to diverse problem instances or deliver near-optimal results. The inherent complexity and variability of CO problems require more adaptable and automated methods.

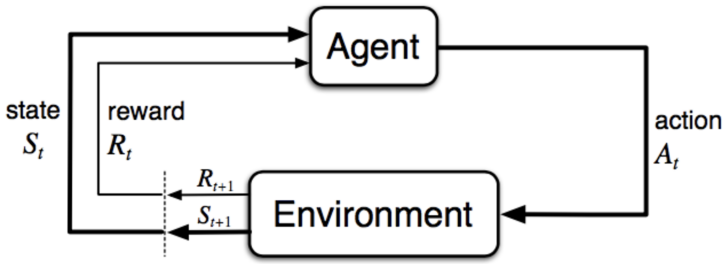

Figure 1: The agent-environment interaction in a Markov decision process. [Sutton & Barto (2018)]

Reinforcement learning (RL), a subfield of machine learning, has emerged as a promising alternative to solve CO problems. Unlike supervised learning, which requires labeled datasets, RL models learn by interacting with an environment to maximize cumulative rewards. By formulating CO problems as sequential decision-making tasks, RL enables the automation of heuristic design and optimization. Recent advances, such as deep reinforcement learning (DRL), which integrates neural networks with traditional RL techniques, have demonstrated significant success in tackling large-scale and complex CO problems.

This survey explores the intersection of RL and CO, focusing on how RL frameworks can be applied to solve canonical optimization problems. We delve into the formulation of CO problems as Markov Decision Processes (MDPs), highlighting key RL algorithms such as Deep Q-Networks (DQN), Policy Gradient Methods, and Actor-Critic Methods. We discuss the main advances in this field, compare traditional approaches, and outline open directions for future research.

*Guntas Singh Saran - 22110089 - guntassingh.saran@iitgn.ac.in

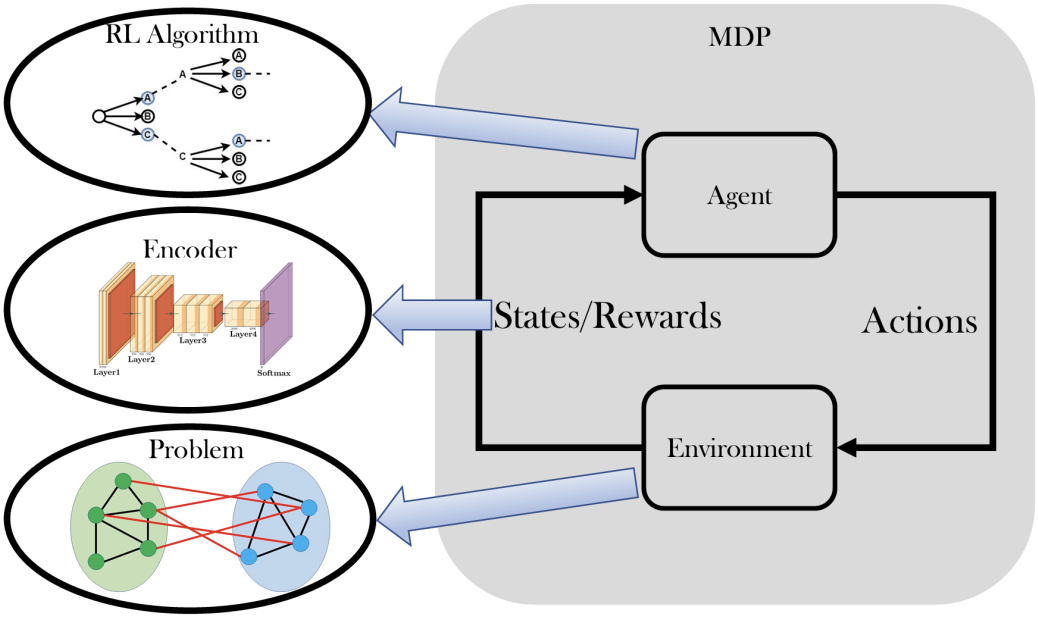

Figure 2: Solving a CO problem with the RL approach requires formulating MDP. The environment is defined by a particular instance of CO problem (e.g. Max-Cut problem). States are encoded with a neural network model (e.g. every node has a vector representation encoded by a graph neural network). The agent is driven by an RL algorithm (e.g. Monte-Carlo Tree Search) and makes decisions that move the environment to the next state (e.g. removing a vertex from a solution set).[Mazyavkina et al. (2020)]

2 Problem Definition

2.1 Combinatorial Optimization Problems

Combinatorial Optimization (CO) problems involve finding an optimal solution from a finite or countable set of discrete possibilities. These problems are widely encountered in operations research, computer science, and industrial applications. Formally, a CO problem can be defined as:

> Let V be a finite set of elements and $f : V \to \mathbb{R}$ be a cost function. The goal of a CO problem is to find an element $v^* \in V$ that minimizes or maximizes f, i.e.,
>
> $$v^* = \arg\min_{v \in V} f(v) \quad \text{or} \quad v^* = \arg\max_{v \in V} f(v).$$

Due to their discrete nature, many CO problems are classified as NP-hard, meaning that no polynomial-time algorithm is known to solve them exactly. Examples of CO problems include the Travelling Salesman Problem (TSP), Maximum Cut Problem (Max-Cut), and Mixed-Integer Linear Programs (MILPs). Below, we formally define some of these problems.

As can be seen, RL algorithms depend on the functions that take as input the states of MDP and outputs the actions' values or actions. States represent some information about the problem such as the given graph or the current tour of TSP, while Q-values or actions are numbers. Therefore an RL algorithm has to include an encoder, i.e., a function that encodes a state to a number. Many encoders were proposed for CO problems including recurrent neural networks, graph neural networks, attention-based networks, and multi-layer perceptrons.

To sumup,a pipeline for solving CO problem with RL is presented in Figure 2. A CO problem is first reformulated in terms of MDP, i.e., we define the states, actions, and rewards for a given problem. We then define an encoder of the states, i.e. a parametric function that encodes the input states and outputs a numerical vector (Q-values or probabilities of each action). The next step is the

actual RL algorithm that determines how the agent learns the parameters of the encoder and makes the decisions for a given MDP. After the agent has selected an action, the environment moves to a new state and the agent receives a reward for the action it has made. The process then repeats from a new state within the allocated time budget. Once the parameters of the model have been trained, the agent is capable of searching the solutions for unseen instances of the problem.

2.2 Examples of CO Problems

2.2.1 Maximum Cut Problem (Max-Cut)

Given a graph $G = (V, E)$ with a set of vertices V and edges E, where each edge $(i, j) \in E$ has a weight $w_{ij} \geq 0$, the Max-Cut problem aims to partition V into two subsets S and $V \setminus S$ such that the total weight of edges between the two subsets is maximized. Mathematically, the objective is:

$$\text{Maximize: } C(S, G) = \sum_{i \in S, j \in V \setminus S} w_{ij}.$$

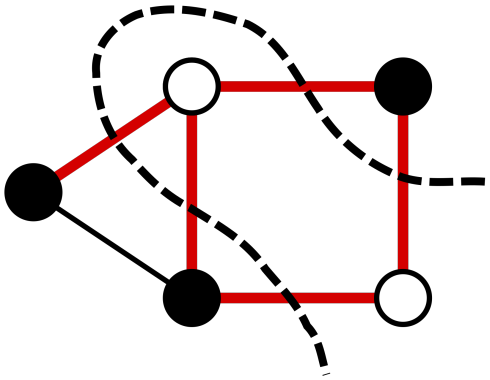

Figure 3: An Example of the Max-Cut S^* in the Graph.

2.2.2 Mixed-Integer Linear Program (MILP)

MILPs [Wolsey (1998)] represent a class of CO problems involving both continuous and integer variables. An MILP can be formulated as:

$$\text{Minimize: } \{c^\top x \quad \text{subject to } Ax \leq b,\ x \in \mathbb{Z}^p \times \mathbb{R}^{n-p}\},$$

where:

- $x \in \mathbb{Z}^p \times \mathbb{R}^{n-p}$ is the decision variable, comprising both integer and continuous components.
- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $b \in \mathbb{R}^m$ is the constraint vector.
- $c \in \mathbb{R}^n$ is the cost coefficient vector.

2.2.3 Travelling Salesman Problem (TSP)

Given a complete weighted graph $G = (V, E)$, find a tour of minimum total weight, i.e. a cycle of minimum length that visits each node of the graph exactly once.

2.3 Markov Decision Process Formulation

Reinforcement Learning (RL) approaches model CO problems as Markov Decision Processes (MDPs) [Bellman (1957)]. An MDP is defined by the tuple (S, A, R, T, γ, H), where:

- S (*State Space*): The set of all possible states of the environment. In CO problems, S typically represents partial solutions or configurations.
- A (*Action Space*): The set of all actions available to the agent. Actions correspond to decisions such as selecting the next vertex in Max-Cut or adding a variable in MILPs.
- R (*Reward Function*): A mapping $R : S \times A \to \mathbb{R}$ that assigns a reward to each state-action pair. In CO, rewards are often designed to reflect improvements in the objective function (e.g., increasing the cut value in Max-Cut).
- T (*Transition Function*): A deterministic or stochastic mapping $T : S \times A \to S$ that describes how the state evolves in response to an action.
- γ (*Discount Factor*): A scalar $\gamma \in [0, 1]$ that determines the importance of future rewards relative to immediate rewards.
- H (*Horizon*): The length of an episode, which is the number of actions the agent can take before the process terminates.

The objective in an MDP is to find an optimal policy $\pi^*(s)$, which maps states to actions and maximizes the expected cumulative discounted reward:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{H} \gamma^t R(s_t, a_t)\right].$$

2.4 Formulating CO Problems as MDPs

To apply RL to CO problems, the graph or solution space is encoded as an MDP:

- **States**: Represent the current partial solution (e.g., subset of vertices selected in Max-Cut).
- **Actions**: Define the possible modifications or extensions to the current solution (e.g., adding a node to a subset in Max-Cut or selecting a variable in MILP).
- **Rewards**: Quantify the improvement in the solution quality (e.g., incremental change in cut weight for Max-Cut).
- **Transitions**: Governed by the rules of the CO problem (e.g., updating the graph structure or partition after an action).

This framework enables RL algorithms to learn effective policies for constructing or improving solutions to CO problems.

2.5 Encoders

In order to process the input structure (e.g. graphs) of CO problems, we must present a mapping from S to a d-dimensional space $\mathbb{R}^d$. We call such a mapping an encoder as it encodes the original input space. The encoders vary depending on the particular type of the space but there are some common architectures that researchers have developed over the last years to solve CO problems.

In order to model long-range dependencies, there exist *Attention Models* in which similarity scores (e.g. dot product) are computed between the input element and each of the previous elements, and these scores are used to determine the weights of the importance of each of the previous elements to the current element. Attention models has recently gained the superior performance on language modeling tasks (e.g. language translation) [Vaswani et al. (2023)] and have been applied to solving CO problems (e.g. for building incrementally a tour for TSP).

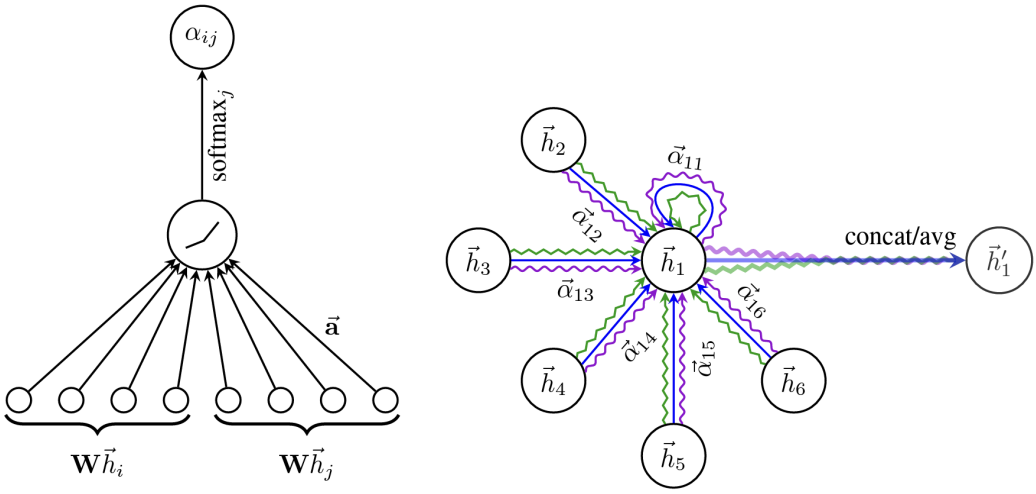

Figure 4: Graph Attention Mechanism $a(\mathbf{W}\mathbf{h}_i, \mathbf{W}\mathbf{h}_j)$ parameterized by a weight vector $\mathbf{a} \in \mathbb{R}^{2F'}$. Different swiggly arrows represent the Multi-Head Attention. [Veličković et al. (2018)]

Graph Neural Networks (GNNs) are a natural extension of attention models to graph-based combinatorial optimization problems. These models represent nodes as initial vectors (e.g., unit vectors) and iteratively update these representations based on the local neighborhood structure using message-passing paradigms. In this framework, nodes exchange information with their neighbors to refine their embeddings, focusing only on connected elements rather than the entire graph as in attention models.

Popular GNN variants include Graph Convolutional Networks (GCN) [Kipf & Welling (2017)], Graph Attention Networks (GAT) [Veličković et al. (2018)], Graph Isomorphism Network (GIN) [Xu et al. (2019)], and Structure-to-Vector Networks (S2V) [Dai et al. (2020)]. Despite differences in implementation, all these models are differentiable and learn node or graph representations through gradient descent, which can then be utilized by reinforcement learning agents [Mazyavkina et al. (2020)].

3 Main Directions

3.1 Reinforcement Learning Algorithms

Reinforcement Learning (RL) provides a framework for solving sequential decision-making problems by training an agent to maximize cumulative rewards through interaction with an environment. RL algorithms can be broadly classified into *value-based methods* and *policy-based methods*, with each having distinct approaches for finding the optimal policy.

3.1.1 Value-Based Methods

Value-based methods aim to estimate the value of states or state-action pairs, which guide the agent to select actions that maximize future rewards. The core functions in value-based methods are [Weng (2018)]:

State Value Function ($V(s)$) The value of a state s under a policy π is defined as the expected cumulative discounted reward starting from s:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s\right],$$

where $\gamma \in [0, 1]$ is the discount factor, and $R(s_t, a_t)$ is the reward at time t.

Action-Value Function ($Q(s,a)$) The action-value function estimates the expected reward of taking an action a in state s, followed by the policy π:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t,a_t) \mid s_0 = s, a_0 = a\right].$$

Bellman Equations The value functions satisfy the recursive Bellman equations [Bellman (1957)]:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R(s,a) + \gamma V^{\pi}(s')\right],$$
$$Q^{\pi}(s,a) = R(s,a) + \gamma \mathbb{E}_{\pi}\left[Q^{\pi}(s',a')\right],$$

where s' is the next state and a' is the next action.

Q-Learning Q-learning is a model-free RL algorithm that iteratively updates the Q-values using the Bellman optimality equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right],$$

where α is the learning rate. Q-learning converges to the optimal action-value function $Q^*(s,a)$, from which the optimal policy $\pi^*(s) = \arg\max_a Q^*(s,a)$ can be derived.

Deep Q-Networks (DQN) DQN [Mnih et al. (2015)] extends Q-learning by approximating the Q-function $Q(s,a)$ with a neural network. The network is trained using a loss function:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')}\left[\left(r + \gamma \max_{a'} Q_{\theta^-}(s',a') - Q_{\theta}(s,a)\right)^2\right],$$

where θ represents the network parameters and θ^- are the parameters of a target network updated periodically. DQN introduces techniques such as experience replay and target networks to stabilize training.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t),a;\theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 5: Algorithm for DQN with experience replay and occasionally frozen optimization target. The prepossessed sequence is the output of some processes running on the input images of Atari games. [Mnih et al. (2015)]

3.1.2 Policy-Based Methods

Policy-based methods directly optimize the policy $\pi_\theta(s, a)$, parameterized by θ, by maximizing the expected cumulative reward:

$$J(\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \right].$$

Policy Gradient Methods The policy gradient theorem [Sutton & Barto (2018)] provides the gradient of the objective $J(\theta)$ with respect to θ:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\nabla_\theta \log \pi_\theta(a \mid s) Q^\pi(s, a) \right].$$

Actor-Critic Methods Actor-critic algorithms (A2C/A3C) [Mnih et al. (2016)] combine value-based and policy-based approaches by maintaining two models:

- The *actor*, which updates the policy $\pi_\theta(s, a)$.
- The *critic*, which estimates the value function $V^\pi(s)$ or $Q^\pi(s, a)$.

The critic evaluates the actor's performance and provides feedback to improve the policy.

Proximal Policy Optimization (PPO) PPO [Schulman et al. (2017)] is a policy-gradient method that improves training stability by enforcing a constraint on the policy update. The objective function is:

$$L^{\text{PPO}}(\theta) = \mathbb{E} \left[\min \left(r_t(\theta) A(s, a), \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) A(s, a) \right) \right],$$

where $r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$ is the probability ratio, and $A(s, a)$ is the advantage function.

3.2 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) [Browne et al. (2012)] is a model-based reinforcement learning technique that combines tree search with Monte Carlo simulations. It is particularly effective for decision-making in domains with large action spaces and deterministic or stochastic transitions. MCTS iteratively builds a search tree by exploring the state-action space and simulating potential outcomes to guide future decisions.

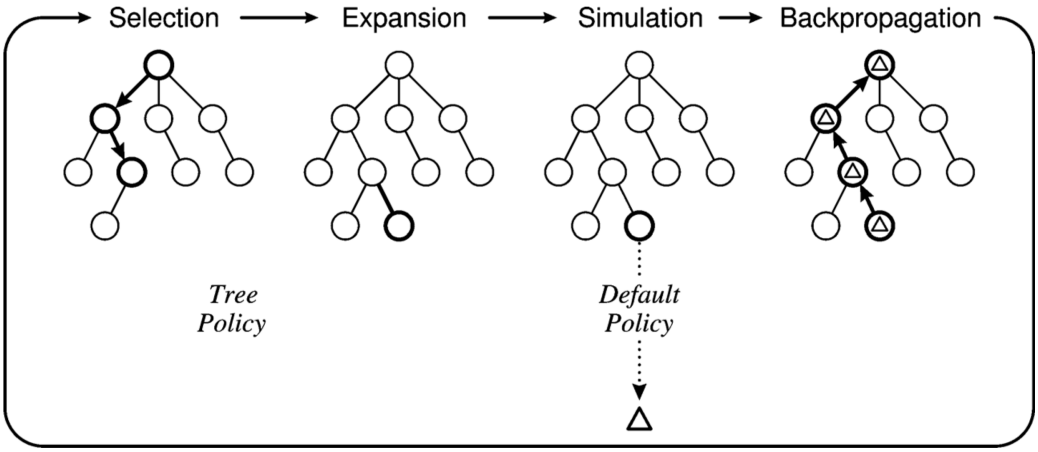

Figure 6: One iteration of the general MCTS approach

Key Components of MCTS MCTS operates through four key steps during each iteration:

1. **Selection:** Starting from the root node, a policy is used to traverse the tree and select child nodes until a leaf node is reached. Typically, the Upper Confidence Bound (UCB) criterion is used:
$$UCB(s,a) = Q(s,a) + c \cdot P(s,a) \cdot \sqrt{\frac{\sum_{a'} N(s,a')}{1+N(s,a)}},$$
where:
   - $Q(s,a)$ is the action-value of a in state s.
   - $P(s,a)$ is the prior probability of selecting action a.
   - $N(s,a)$ is the visit count of action a in state s.
   - c is a constant controlling the exploration-exploitation tradeoff.
2. **Expansion:** When a leaf node is reached, new child nodes corresponding to available actions are added to the tree. The state transitions are simulated to generate the child nodes.
3. **Simulation:** A random or heuristic policy is used to simulate a complete trajectory from the newly expanded node to a terminal state. The total reward obtained in this simulation is recorded.
4. **Backpropagation:** The reward from the simulation is propagated back up the tree, updating the action-value estimates $Q(s,a)$ and visit counts $N(s,a)$ for each node along the path.

Policy and Value Estimation MCTS can be enhanced by leveraging learned policies and value functions. Neural networks are often used to estimate:

- $P(s,a)$: Prior probabilities for action selection.
- $V(s)$: Value of the state, used to initialize action-value estimates during backpropagation.

For example, in AlphaGo Zero [Silver et al. (2016)] [Silver et al. (2017)], a neural network provides both $P(s,a)$ and $V(s)$, integrating MCTS with deep learning.

Algorithm Output After a predefined number of iterations or time, MCTS selects the action with the highest visit count from the root node:
$$\pi(a \mid s_0) = \frac{N(s_0,a)}{\sum_{a'} N(s_0,a')}.$$
This action is executed in the environment, and the tree is updated accordingly.

3.3 Learning Combinatorial Optimization Algorithms over Graphs [Dai et al. (2018)]

This paper introduces a framework that leverages reinforcement learning and graph embedding techniques to automate the design of greedy heuristics for NP-hard combinatorial optimization (CO) problems on graphs. The key contributions of the paper are as follows:

3.3.1 Framework Description

The proposed framework combines the following:

- **Graph Embedding:** The *Structure2Vec (S2V)* network [Dai et al. (2020)] is used to encode graph structure and the current state of partial solutions. The embedding incorporates neighborhood and global graph information.
- **Reinforcement Learning:** The framework employs a greedy policy, trained using Q-learning, to construct solutions incrementally. At each step, the policy selects the next node to add based on the learned evaluation function $Q(h(S), v; \Theta)$, where $h(S)$ represents the helper function maintaining problem constraints, and v is the candidate node.

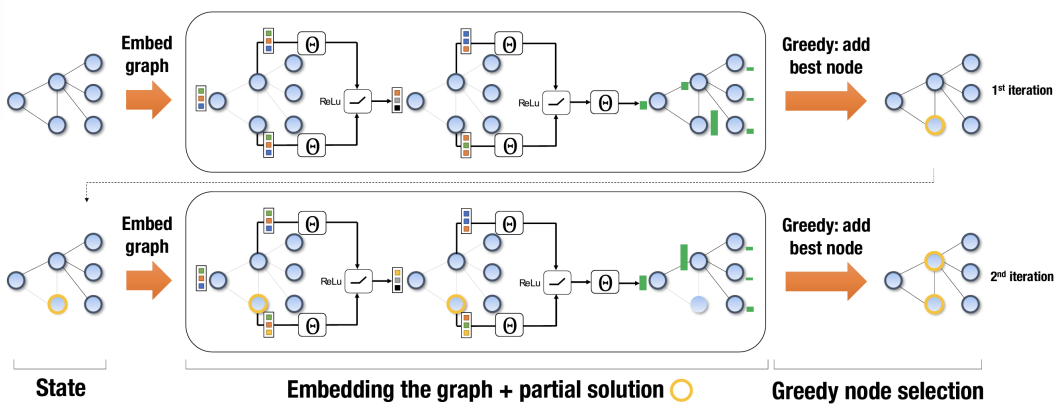

Figure 7: Illustration of the proposed framework as applied to an instance of Minimum Vertex Cover. [Dai et al. (2018)]

3.3.2 Helper Functions for Specific Problems

The framework incorporates problem-specific helper functions $h(S)$, which maintain the constraints and evaluate the incremental impact of adding a node to the current partial solution S. These helper functions are integral to formulating the reinforcement learning policy.

Minimum Vertex Cover (MVC): In the MVC problem, the helper function $h(S)$ ensures that all edges in the graph are covered by the selected vertices in S. For a given node v, the function evaluates the impact of adding v to S by checking:

$$h_{\text{MVC}}(S, v) = \text{Number of uncovered edges incident to } v.$$

This function prioritizes nodes that cover the maximum number of remaining edges, facilitating the construction of a minimal vertex cover.

Maximum Cut (Max-Cut): For the Max-Cut problem, the helper function $h(S)$ computes the incremental increase in the cut value if a node v is added to a subset S. The evaluation is given by:

$$h_{\text{Max-Cut}}(S, v) = \sum_{u \in V \setminus S} w_{uv},$$

where w_{uv} is the weight of the edge between u and v. The function guides the policy to partition nodes in a way that maximizes the cut value.

Traveling Salesman Problem (TSP): For TSP, the helper function $h(S)$ evaluates the cost of extending the current tour S by adding a new node v. The cost is calculated as: The reward, in this case, is defined as the difference in the cost functions after transitioning from the state s to the state s' when taking some action a: $r(s, a) = c(h(s'), G) - c(h(s), G)$, where h is the graph embedding function of the partial solutions s and s', G is the whole graph, c is the cost function. Because the weighted variant of the TSP is solved, the authors define a cost function $c(h(s), G)$ as the negative weighted sum of the tour length.

3.3.3 Results

A variant of the Q-learning algorithm was used to learn to construct the solution, that was trained on randomly generated instances of graphs. The S2V network updates node embeddings iteratively based on graph topology and node features, capturing long-range interactions. This approach achieves better approximation ratios compared to the commonly used heuristic solutions of the problem, as well as the generalization ability, which has been shown by **training** on graphs of consisting of **50-100 nodes** and **tested** on graphs with up to **1000-1200 nodes**, achieving very good approximation ratios to exact solutions.

3.4 Reinforcement Learning for Integer Programming: Learning to Cut [Tang et al. (2020)]

This paper proposes a reinforcement learning (RL) approach to enhance the cutting plane method in integer programming (IP), which is a crucial component in optimization solvers. The method models the selection of cutting planes as a Markov Decision Process (MDP) and integrates it with the branch-and-cut algorithm to improve solver efficiency. The cutting plane method adds linear constraints iteratively to an LP relaxation to tighten the feasible region and reduce the integrality gap. However, selecting effective cuts remains a challenge. This paper adapts RL to optimize cut selection, specifically focusing on learning heuristics for Gomory cuts, a widely used cutting plane in branch-and-cut solvers.

3.4.1 MDP Formulation and Gomory Cuts

The RL-based approach is modeled as an MDP where:

- **State Space** (S): Includes the original linear constraints, the set of cuts already applied, and the current solution from the LP relaxation.
- **Action Space** (A): The set of candidate Gomory cuts that can be generated and added to the problem.
- **Transition Function**: Describes the state change when a chosen cut is added to the current LP relaxation, resulting in a new feasible region.
- **Reward Function** (R): Defined as the difference in the objective value between two consecutive LP relaxations after adding a cut $R(s, a) = \text{Objective}(s') - \text{Objective}(s)$, where s' and s are the new and previous states, respectively. This reward quantifies the improvement in the solution quality due to the cut.

3.4.2 Policy Network Architecture

- An LSTM network to encode the state representation, capturing a variable number of variables.
- An attention mechanism to handle a variable number of constraints, enabling the model to focus on relevant parts of the problem during cut selection.

The policy gradient algorithm is used for training, optimizing the network to select Gomory cuts that maximize the reward function.

3.4.3 Applications and Results

The method was evaluated on benchmark IP instances, including packing, planning, and Max-Cut problems. Key findings include:

- Improved efficiency of cut selection, with reductions in the number of cuts needed compared to traditional heuristics.
- Significant closure of the integrality gap in LP relaxations, enhancing the overall performance of the solver.
- Generalization to various problem instances, demonstrating that the approach can adapt to different types of IPs.
- Beneficial synergy with branch-and-cut strategies, leading to faster convergence in practical experiments.

3.5 Solving NP-Hard Problems on Graphs with Extended AlphaGo Zero [Abe et al. (2020)]

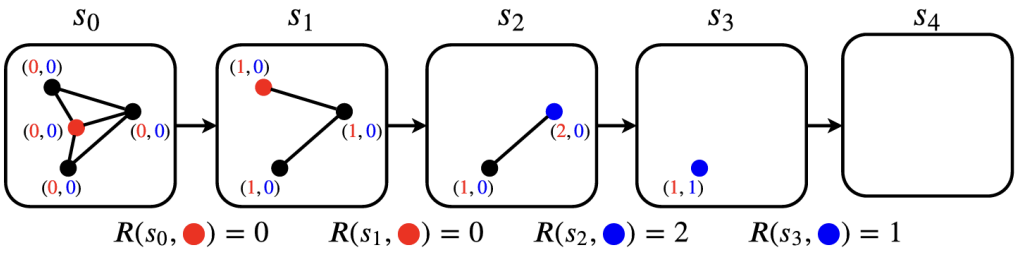

Figure 8: Example of MAXCUT MDP trainsitions. [Abe et al. (2020)]

Dai et al. (2018) proposed an end-to-end reinforcement learning framework, S2V-DQN, which automatically learns graph embeddings to construct solutions to a wide range of problems. To improve the generalization ability of their Q-learning method, we propose a novel learning strategy based on AlphaGo Zero [Silver et al. (2017)] which is a Go engine that achieved a superhuman level without the domain knowledge of the game. Their framework is redesigned for combinatorial problems, where the final reward might take any real number instead of a binary response, win/lose. They propose to replace Q-learning with our novel learning strategy named CombOpt Zero.

3.5.1 MDP Formulation for Max-Cut

In each action, we color a node by 0 or 1, and remove it while each node keeps track of how many adjacent nodes have been colored with each color. $A_s = \{(x, c) \mid x \in V, c \in \{1, 2\}\}$ denotes a set of possible coloring of a node, where (x, c) means coloring node x with color c. $L = \mathbb{N}^2$, representing the number of colored (and removed) nodes in each color (i.e., l_0 is the number of (previously) adjacent nodes of x colored with 0, and same for l_1, where $l = d(x)$). Init uses the same graph as G_0 and sets both of $d(x)$ to 0 for all $x \in V$. $T(s, (x, c))$ increases the c-th value of $d(x)$ by one for $x \in N(x)$ and removes x and neighboring edges from the graph. S_{end} is the state with the empty graph. $R(s, (x, c))$ is the $(3 - c)$-th (i.e., 2 if $c = 1$ and 1 if $c = 2$) value of $d(x)$, meaning the number of edges in the original graph which have turned out to be included in the cut set (i.e., colors of the two nodes are different).

3.5.2 Outcomes

Several GNNs were compared as the graph encoders, with GIN [Xu et al. (2019)] being shown to be the most performing. Also, the training procedure similar to AlphaGo Zero [Silver et al. (2017)] was employed with the modification to accommodate for a numeric rather than win/lose solution. The experiments were performed with a vast variety of generated and real-world graphs. The extensive comparison of the method with several heuristics and with previously described S2V-DQN [Dai et al. (2018)] showed the superior performance as well as the better generalization ability to larger graphs, yet they didn't report any comparison with the exact methods.

3.6 A Deep Learning Algorithm for the Max-Cut Problem based on Pointer Network Structure with Supervised Learning and Reinforcement Learning Strategies [Gu & Yang (2020)]

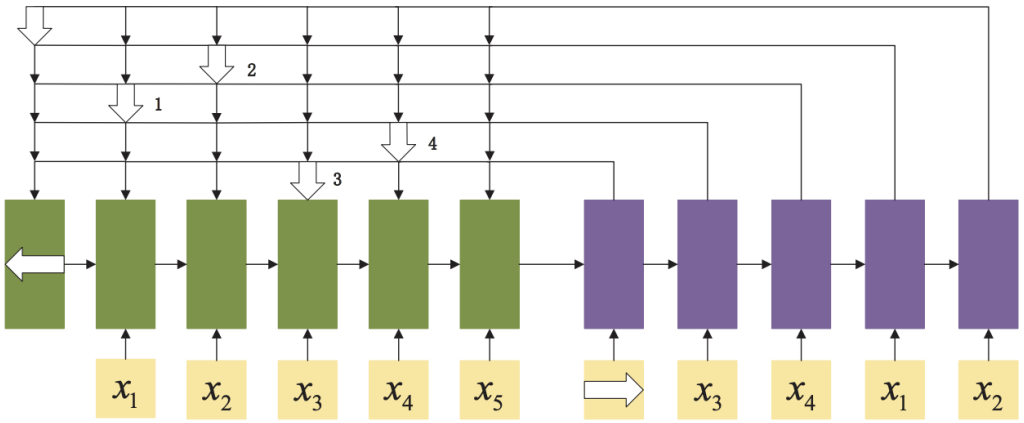

Figure 9: The architecture of pointer network (encoder in green, decoder in purple). [Vinyals et al. (2017)]

Gu & Yang (2020) applied the Pointer Network [Vinyals et al. (2017)] along with the Actor-Critic algorithm to iteratively construct a solution. The MDP formulation defines the state, S, as a symmetric matrix, Q, the values of which are the edge weights between nodes (0 for the disconnected nodes). Columns of this matrix are fed to the Pointer Network, which sequentially outputs the actions, A, in the form of pointers to input vectors along with a special end-of-sequence symbol `<EOS>`. The resulting sequence of nodes separated by the `<EOS>` symbol represents a solution to the problem, from which the reward is calculated. The authors conducted experiments with simulated graphs with up to 300 nodes and reported fairly good approximations ratios, but, unfortunately, didn't compare with the previous works or known heuristics.

4 Summary of the Approaches

Table 1: Summary of the Approaches of Maximum Cut Problem

Approach	Training	
	Encoder	RL
Dai et al. (2018)	S2V	DQN
Tang et al. (2020)	LSTM + Attention	Policy Gradient + ES
Abe et al. (2020)	GNN	Neural MCTS
Gu & Yang (2020)	Pointer Network	A3C

5 Open Directions

The previous sections have outlined several reinforcement learning (RL) approaches for solving combinatorial optimization (CO) problems, demonstrating performance on par with state-of-the-art heuristic methods and solvers. While promising, the field remains in its early stages, leaving multiple avenues for exploration:

5.1 Generalization to Other Problems

A key limitation in the current RL-CO field is the lack of generalization across problem domains and instances. Existing methods often require tailoring to specific CO problems, limiting their applicability. Future work could explore generalizing learned policies to unseen problem instances, such as smaller graphs, instances with different distributions, or even other types of CO problems. Advancements in this direction could broaden the scope and impact of RL in optimization.

5.2 Improving Solution Quality

Although RL-based methods have shown competitive performance, their success is often limited to less complex problem instances (e.g., graphs with fewer nodes). Improving solution quality for larger and more complex problems is an important challenge. Combining classical CO algorithms with RL, such as incorporating imitation learning, presents an opportunity to enhance both performance and scalability.

5.3 Interpretability of RL Solutions

One of the significant challenges in RL-CO methods is the lack of interpretability of the learned solutions. While RL algorithms are adept at finding near-optimal solutions, they often do not provide insights into the reasoning or methodology behind these solutions. Developing models and frameworks that not only generate high-quality solutions but also explain the decision-making process—such as identifying critical problem features or steps contributing to the solution—could enhance trust and usability, especially in high-stakes applications.

5.4 Filling Algorithmic Gaps

Current research reveals underexplored areas in algorithm design for certain CO problems. For instance, some problems, such as Bin Packing and Minimum Vertex Cover, lack algorithms that integrate both joint and constructive approaches. Developing novel methods in these unexplored areas could yield valuable insights and expand the toolbox for RL-CO research.

REFERENCES

Kenshin Abe, Zijian Xu, Issei Sato, and Masashi Sugiyama. Solving np-hard problems on graphs with extended alphago zero, 2020. URL https://arxiv.org/abs/1905.11623.

Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5): 679–684, 1957. URL http://www.jstor.org/stable/24900506.

Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810.

Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs, 2018. URL https://arxiv.org/abs/1704.01665.

Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data, 2020. URL https://arxiv.org/abs/1603.05629.

Shenshen Gu and Yue Yang. A deep learning algorithm for the max-cut problem based on pointer network structure with supervised learning and reinforcement learning strategies. *Mathematics*, 8:298, 02 2020. doi: 10.3390/math8020298.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017. URL https://arxiv.org/abs/1609.02907.

Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey, 2020. URL https://arxiv.org/abs/2003.03600.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015. doi: 10.1038/nature14236.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016. URL https://arxiv.org/abs/1602.01783.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016. doi: 10.1038/nature16961.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017. URL http://dx.doi.org/10.1038/nature24270.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut, 2020. URL https://arxiv.org/abs/1906.04859.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018. URL https://arxiv.org/abs/1710.10903.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks, 2017. URL https://arxiv.org/abs/1506.03134.

Lilian Weng. A (long) peek into reinforcement learning. *lilianweng.github.io*, 2018. URL https://lilianweng.github.io/posts/2018-02-19-rl-overview/.

L.A. Wolsey. *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 1998. ISBN 9780471283669. URL https://books.google.co.in/books?id=x7RvQgAACAAJ.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019. URL https://arxiv.org/abs/1810.00826.