Engineering Applications of Artificial Intelligence 105 (2021) 104422

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

Learning to traverse over graphs with a Monte Carlo tree search-based self-play framework

Qi Wang[a,*], Yongsheng Hao[b], Jie Cao[c]

[a] *School of Computer Science, Fudan University, China*
[b] *Network Center, Nanjing University of Information Science and Technology, China*
[c] *Xuzhou University of Technology, China*

A R T I C L E   I N F O

Keywords:
Combinatorial optimization
Deep learning
Reinforcement learning
Monte Carlo tree search

A B S T R A C T

The combinatorial optimization (CO) problems on the graph are the core and classic problems in artificial intelligence (AI) and operations research (OR). For example, the Vehicle Routing Problem (VRP) and Traveling Salesman Problem (TSP) are fascinating NP-hard problems and have important significance for the existing transportation system. Traditional methods such as heuristics methods, exact algorithms, and solution solvers can already find approximate solutions on small-scale graphs. However, they are helpless for large-scale graphs and other problems with similar structures. Moreover, traditional methods often require artificially designed heuristic functions to aid decision-making. In recent years, more and more work has focused on applying deep learning and reinforcement learning (RL) to learn heuristics, which allows us to learn the internal structure of the graph end-to-end and find the optimal path under the guidance of heuristic rules. However, most of these still need manual assistance, and the RL method used has the problems of low sampling efficiency and small searchable space. This paper proposes a novel framework (called OmegaZero) based on Alphago Zero, which does not prescribe expert experience or label data but is trained through self-play. We divide the learning into two stages: in the first stage, we employ graph attention network (GAT) and GRU to learn node representations and memory history trajectories. In the second stage, we employ Monte Carlo tree search (MCTS) and deep RL to search for the solution space and train the model.

1. Introduction

Combinatorial optimization (CO) is an essential tool to resolve some core problems in information theory, management science, computer science, AI, and other disciplines. Known classic NP-hard problems such as the vehicle routing problem (VRP) (Mor and Speranza, 2020), the traveling salesman problem (TSP) (Goyal, 2010), and so on belong to the category of combinatorial optimization problems. It is of central theoretical significance and practical application value to quickly solve large-scale CO problems.

Over the years, many traditional algorithms such as approximation algorithms and heuristic algorithms (Huang et al., 2021; Aqil and Allali, 2021; Meng et al., 2021; Dehghan-Sanej et al., 2021) have been designed to solve CO problems, including simulated annealing algorithms, genetic algorithms, nearest neighbor algorithms, and heuristic solution solvers such as “OR-Tools”, “LKH3”, and “Concorde”. The above algorithms are often well targeted and accurate for specific problems, but we need to design new algorithms again and again for different instances of similar problems. That is, the previous solving experience is not helpful for the problems to be resolved. Traditional algorithms do not make good use of the fact that in practical applications, most CO problems in the same scenario have similar combinatorial structures, and the difference lies only in values and variables. (Dai et al., 2017). Therefore, researchers hope to design a general method that can excavate the essential information of the problem through learning and improve the quality and efficiency of problem solutions by constantly updating the solution policy iteratively.

In recent years, with the rapid development of big data and artificial intelligence (AI) technology, machine learning, especially its deep learning (Lecun et al., 2015) and reinforcement learning (RL) (Botvinick et al., 2019), is increasingly applied to solve CO problems. In the face of enormous search space and data points, it should be a reasonable scheme to combine the perceptual ability of deep learning with the reasoning ability of RL. Deep RL has made a revolutionary breakthrough and extensive influence in the field of artificial intelligence, such as AlphaGo (Silver et al., 2016a), AlphaGo Zero (Silver et al., 2017b), and AlphaZero (Silver et al., 2018), etc. The fundamental motivation of applying deep learning and RL to CO lies in the discovery and reasoning of new policies. Compared with traditional

* Corresponding author.
E-mail address: 17110240039@fudan.edu.cn (Q. Wang).

https://doi.org/10.1016/j.engappai.2021.104422
Received 31 March 2021; Received in revised form 11 July 2021; Accepted 4 August 2021
Available online 19 August 2021

algorithms, machine learning can discover the inherent characteristics of the instances to guide future instances by learning and applying the solving experience of existing instances (Romero-Hdz et al., 2020). It also makes it possible for NP-hard problems that were not easy to solve in the past.

The application of deep learning and RL in combinational optimization has obtained some preliminary results, but it is still in the experimental stage. The problems they currently solve, including VRP, TSP, and MVC (Dai et al., 2017), are often limited to the scale of hundreds of nodes. And they still need to adjust the policy network, reward function, and decoding to solve similar problems with similar structures, which have not reached complete universality. Most of the existing learning-based approaches are essentially learning heuristics, but they cannot have the ability to learn heuristics independently and require artificial design heuristics to assist learning. Besides, it remains challenging to learn heuristics more cheaply, intelligently, and efficiently and integrate state-of-the-art models with their training methods. The present RL has some problems, such as sparse rewards, low sampling efficiency, and limited space exploration.

CO is similar to GO since they all look for feasible solutions or optimal solutions under constraints in a vast combinatorial space. Even in terms of search scale, the CO problem on the graph can have a much larger search space than go, so the CO on the graph is not low in complexity. AlphaGo Zero won by thinking smarter rather than faster, by discovering the rules of board play on its own to develop a way of playing that reflects the truth of the game rather than programmer priorities and biases. It reveals the fact and advocates a direction that in intelligent optimization, we are not just competing for computing power, algorithms, and scale, but also for intelligence and insights. Inspired by AlphaGo Zero, we designed a lightweight framework for combinational optimization that applies the Q-Learning algorithm with MCTS (Browne et al., 2012) to model the policy network and the value network. We first design a memory component that combines GAT (Veličković et al., 2018) and GRU (Cho et al., 2014) to process the input graph. It can selectively memory state trajectories avoiding pre-training. Then we employ the history trajectories to the joint modeling policy and value. Finally, employ the Q-learning algorithm with MCTS to optimize and update the policy network and value network.

To sum up, the main contributions of this paper are as follows:

- We design a memory component based on GAT and GRU, which can selectively remember historical trajectories for pathfinding and reasoning to omit the pre-training in previous work.
- We employ the history trajectories generated by the memory component to jointly model the policy network and the value network, which can be updated and promoted simultaneously.
- We employ the Q-learning with MCTS to optimize the policy network and the value network more efficiently and more conducive to expanding the exploration space and avoiding sparse rewards than the REINFORCE algorithm and actor-critic algorithms.
- We design a scoring function based on MCTS to predict the nodes to be searched and reasoned about more efficiently in the testing stage.

2. Related work

The Hopfield network (Hopfield and Tank, 1985) should be the earliest application of neural networks in combinatorial optimization (CO). With the rapid development of artificial intelligence technology and hardware devices (such as GPU, TPU, etc.), more and more researchers are committed to applying machine learning (Jordan and Mitchell, 2015) to traditional CO problems in recent years. Inspired by the great success of deep RL in the game, some researchers have tried to transfer it to CO. To make this paper self-consistent, we first introduce the application of deep RL in games, sort out the representative learning-based methods used in CO in recent years, and finally explain the inspiration of the AlphaGo series for solving CO problems.

2.1. Games with deep RL

Deep RL (Ivanov and D'yakonov, 2019), which integrates deep learning and RL (Littman, 2015), has become one of the most mainstream directions of artificial intelligence. Mnih et al. proposed DQN, which combined with the deep neural network, Q-learning, and experience playback to achieve the human-level performance of Atari games (Mnih et al., 2015). DreamerV2 was the first to achieve human-level performance on 55 Atari benchmarks by learning behavior in a single trained world model (Hafner et al., 2020). Agent 57 was the first deep RL agent to exceed the standard human benchmark in all 57 Atari games (Badia et al., 2020). DeepMind designed a series of algorithms for the more complex games of Go, chess, and so on, such as AlphaGo, AlphaGo Zero, and AlphaZero, and applied them to defeat professional human masters, officially ushering in the era of artificial intelligence (Silver et al., 2017a). MCTS (Guez et al., 2018) is one of the core techniques of these go/chess algorithms, which is applied to search for the vast state space while using UCB to make decision choices to balance exploration (Browne et al., 2012). AlphaGo employs a two-stage training pipeline consisting of supervised learning and RL, expert experience-assisted training for a hot start in the initial phase, and RL to update and optimize policy (Drori et al., 2020) iteratively. AlphaGo Zero and AlphaZero are updated versions of AlphaGo, which do not require any human experience. Still, only the rules of go, then train themselves with the data generated by self-play and easily beat AlphaGo in tests (Silver et al., 2017b). Deep RL has been hugely successful in increasingly complex single-agent environments and two-player rotational games (Jaderberg et al., 2019). However, the real world is more complex and consists of multiple agents, each learning and acting independently and then cooperating and competing. DeepMind applied tournament-style evaluations to verify that agents could achieve human-level performance in 3D multiplayer first-person video games such as "Quake III Arena in Capture the Flag Mode" (Jaderberg et al., 2019). DeepMind's Alpha Star was rated grandmaster in all three Starcraft tournaments, with over 99.8% of the officially ranked human players (Vinyals et al., 2019).

2.2. Combinatorial optimization (CO) with machine learning

Integrating machine learning (Jordan and Mitchell, 2015) into combinatorial optimization problems is rapidly developing in recent years (Bengio et al., 2021; Emami et al., 2020). Compared with traditional methods, it has many potentials, such as good scalability, generalization, and versatility. At present, most of the methods based on machine learning are first to use deep learning (Lecun et al., 2015) to encode the problem, then use reinforcement learning (Littman, 2015) to train the model, and finally use heuristic search to decode one by one to construct a solution. We differentiate the current learning-based methods into attention-based and graph neural network (GNN (Xu et al., 2019))-based methods.

Attention-based Methods. The sequence-to-sequence (Seq2Seq) model has become a popular method for processing sequence data tasks such as translation and chatbots due to its excellent effects (Wiseman and Rush, 2016). Seq2Seq generally uses an encoder–decoder architecture, in which the encoder network encodes the input sequence into a fixed-length vector (Vinyals et al., 2016). Then the decoder network decodes it into the output target sequence. Pointer Network (Ptr-Net) (Vinyals et al., 2015) applies the attention mask directly to indicate the current output node. The attention mask is normalized with softmax. It can be regarded as a distribution, logically like a pointer to a node in the input. The pointer network lays a good foundation for subsequent learning-based work. It uses the commonly used supervised learning with cross-entropy as the loss function.

The limitation of supervised learning for combinatorial optimization is evident because it is difficult for us to obtain many optimal solutions as label data. It is also not conducive to the generalization of the model.

Reinforcement learning is different from supervised learning in that it does not require an explicitly labeled data set but learns through feedback signals from continuous interaction with the environment. Bello et al. proposed neural combinatorial optimization (NCO) (Bello et al., 2017), which uses the Ptr-Net as a policy model, and then applies a gradient-based reinforcement learning method (REINFORCE algorithm) to learn the parameters of the policy model.

Nazari et al. (2018) pointed out that the RNN encoder is only helpful for input order such as translation. Still, it is unnecessary for TSP, sorting tasks, and other input order irrelevant situations. So they removed the encoder part and only kept the decoder part. They replaced the LSTM encoder in Ptr-Net with the embedding of all the inputs to make the model's output independent of the input order. The entire network mainly consists of two parts: embedding based on the two parts of static (coordinates) and dynamic (demand) states; the other part is the decoder, which points to the input node in each step. The decoder uses a context-based attention mechanism with a glimpse.

Kool et al. (2019) employ the transformer (Vaswani et al., 2017) model as a policy network. Their encoder does not use positional encoding, making node embedding independent of the input order compared with the original transformer model. The decoder network is also based on the attention mechanism. A big difference from the previous work is that it introduces a context node to represent the context of decoding, which includes graph embedding and the embedding of the first node and the previous output node. The training still uses the classic REINFORCE algorithm. The difference here is that the baseline function is obtained through the rollout method, the deterministic greedy rollout is obtained based on the current best policy model. In addition, methods based on attention and RNN also include (Pierrot et al., 2019; Selsam et al., 2019; Deudon et al., 2018).

GNN-based Methods. For combinatorial optimization problems, one challenge is that we often need to solve problems of the same type but with different data over and over again. Therefore Dai et al. (2017) put forward a question: "Given a graph optimization problem and a distribution of problem instances, is there a way to generalize the heuristic to problems that have never been seen before heuristically?" It also pointed out that many previous machine learning-based methods did not fully exploit the particular structure of graphs. To this end, they combined structure2vec and RL to propose the S2V-DQN algorithm. In terms of training, considering the inefficiency of the policy gradient commonly used in the previous work, they used Q-learning (Ecoffet et al., 2021). They use a greedy algorithm to decode the solution. At each step, they select the node that maximizes the evaluation function based on the current partial solution. To optimize the parameters in the evaluation function, they took the evaluation function as the Q function in Q-learning and then trained it.

Manchanda et al. (2019) proposed a deep reinforcement learning framework called GCOMB for learning combinatorial optimization problems on large-scale graphs. The whole algorithm includes two stages of training and inference. They divided the training phase into two sub-phases: in the first step, they use GCN (Xu et al., 2019) to get the embedding of the node and then learn the parameters in GCN through supervised learning. It essentially allows embedding to reflect the importance of each node to the solution; in the second step, they use decoding with Q-learning to form a solution set. They set the reward as the marginal benefit after joining a node and then train. In the reasoning stage, for a given graph, they use the trained GCN for embedding and then iteratively calculate the solution through the Q function learned by Q-learning. They used the greedy algorithm to construct the solution incrementally.

Graph attention networks (GAT) (Veličković et al., 2018) introduce the attention mechanism to GNN, which allows nodes to pay attention to specific adjacent nodes. Drori et al. (2020) used reinforcement learning to train GAT on a training set, used the trained neural network on a new graph instance, and then produced an approximate solution. They introduced the edge-to-vertex line graph for problems that require edge selection (such as MST) and kept the calculation at linear complexity. They use GAT to learn the representation of the graph and output the policy of selecting nodes, which is based on the classic encoder–decoder architecture. This method can be applied to a variety of graph-based combinatorial optimization problems through different objective functions. In addition, GNN-based methods also include (Barrett et al., 2020; Ma et al., 2019a; Nowak et al., 2018; Li et al., 2018; Duan et al., 2020; Sobieczky, 2020).

2.3. AlphaGo Zero's inspiration for CO

AlphaGo Zero (Silver et al., 2017b) only applies one deep neural network to replace the two independent networks in AlphaGo (Silver et al., 2016a), namely the policy network and the value network. It outputs the playing policy and the winning rate value of the current go board situation, saving the computing space and reducing operation consumption. The mixed two networks can better adapt to a variety of different situations.

Monte Carlo Tree Search (MCTS) (Browne et al., 2012) is one of the core technologies of the AlphaGo series (Schrittwieser et al., 2020). A significant feature of board games is that they can be represented by a decision tree linked by a move. The number of nodes in the tree depends on the number of branches and the depth of the tree. The purpose of MCTS is to collect as many wins and losses as possible through experimental simulations (rollout, playout) when there are a lot of tree nodes. Based on this statistical information, it balances the focus of the search on the new nodes and the nodes worth exploring to reduce the simulation resource input on the nodes with a high probability of losing. MCTS has four processes: selection, expansion, simulation, and backpropagation.

Inspired by the success of AlphaGo Zero in Go, some scholars have transferred it to CO to achieve higher accuracy and efficiency (Huang et al., 2019; Laterre et al., 2018). However, a common problem of these methods is the excessive demand for an experimental environment. Besides, the policy or value network in the previous MCTS-based methods was trained independently without MCTS participation and only applied to assist MCTS after training. In contrast, our method improved the policy by learning the trajectory generated by MCTS and using the entire MCTS search tree information.

3. Methodology

3.1. Problem setup and Markov decision process

VRP is a generic term for a class of problems in which a fleet based on one or more warehouses must determine a set of routes for a geographically dispersed city or customer (Mor and Speranza, 2020). The goal of VRP is to present a set of customers with known requirements with the lowest-cost vehicle routes starting and ending in a warehouse (Fig. 1). TSP is a simple, particular case of VRP that describes starting from one city and coming back to the starting point after traversing all the other cities with the lowest total cost (Goyal, 2010). VRP is among the most challenging combinational optimization (NP-hard) problems, and much of the interest over the years stems from its practical significance and difficulty.

We adopt the MCTS algorithm in the framework, aiming to transform the routing optimization problem on the graph into the path with the least search cost in the tree. Therefore, we formally define VRP and TSP on the graph. (Xing et al., 2020). $G = (V, E, W)$ is used to represent an undirected and unlabeled graph, where V is a set of vertices and E represents each node and the optimal path is composed of a sequence of nodes $v_0, \ldots, v_i, \ldots, v_0$ where v_0 represents the starting node, E is a set of edges, and W is a set of weights on the edges $e_{ij} = (v_i, v_j)$.

Unify VRP and TSP. In theory, we can apply distributed multi-agents to simulate multi-vehicles to find the paths on the graphs. That is, each

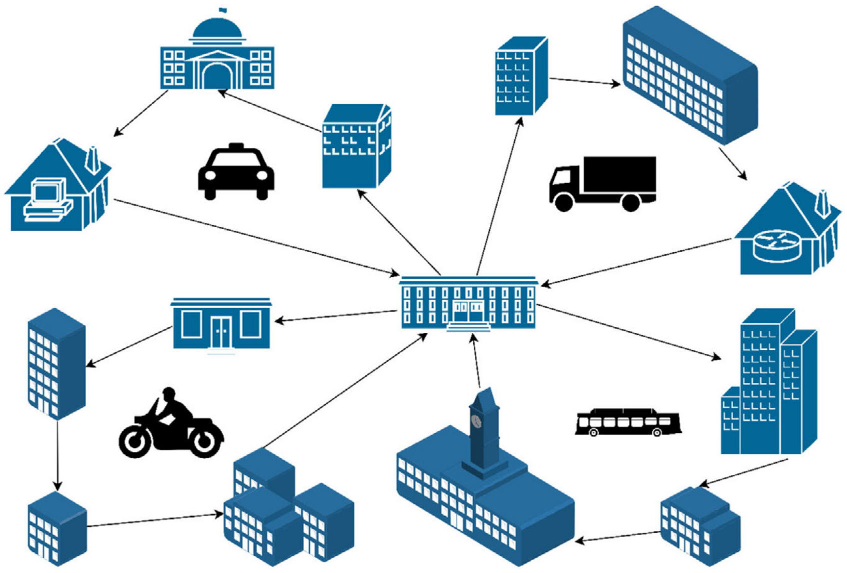


Fig. 1. The demonstration of VRP in the practical application scenario, in which each node can be different facilities or cities are containing their demand. Different paths correspond to different miles. A vehicle can be a truck, a motorbike, a taxi, or a shuttle bus with their respective load capacity. The goal of VRP is to optimize a set of paths to minimize the total cost.

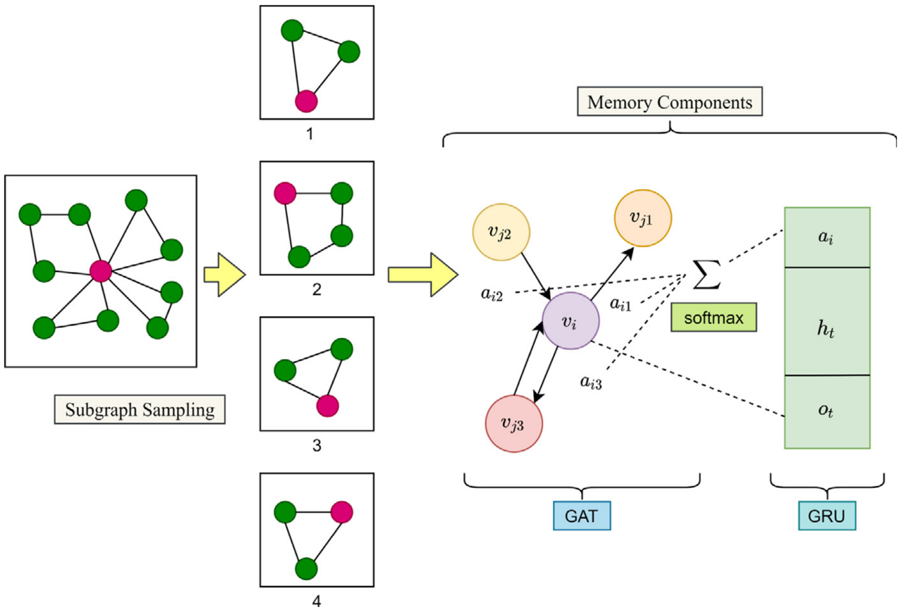


Fig. 2. Flowchart of memory components with GAT and GRU. We take the graph as input and apply subgraph sampling to divide it into k subgraphs with central nodes, then iteratively process each subgraph using the memory component. The subgraph sampling process in the figure is just a simple example, while the actual graph is much more complex. The memory component combining GAT and GRU can effectively infer the importance of neighbor nodes and remember and update their status and trajectories, which enables the model to avoid pre-training.

agent represents a vehicle. However, due to our limited implementation capacity, computing resources, etc., we are always committed to designing a generic and lightweight framework for finding and reasoning paths. If we apply the subgraph sampling algorithm (Zhang et al., 2018) to divide the VRP on the graph into subgraphs of the same size, and each subgraph contains a central node, then the VRP can be transformed into TSP. In this way, we can solve both VRP and TSP without defining two Markov decision processes for them, respectively, and subgraph sampling is equivalent to greedy probability distribution (Manchanda et al., 2019) and Graph reduction (Li et al., 2018) to preprocess large-scale graphs effectively.

Heuristic Functions. The learning-based method to solve combinatorial optimization problems is essential to learn the corresponding heuristics. We can construct a heuristic function for the combinatorial optimization problems on the graph to guide the learning. At the same time, inspired by the Q&A system (Ma et al., 2019b), we can also turn the combinatorial optimization problem into finding an answer to a specific question. For example, for TSP, we can construct a function $f(G, v_0, q)$, whose functional form is generally unknown, where q is a relational query or target such as “shortest distance”, “longest distance”, and “lowest cost”, etc. We need to use something like $(v_0, \dots, v_0)$ such samples (the optimal solution sets) to constitute a training data set to learn $f(G, v_0, q)$. Therefore, in theory, we can solve other combinatorial optimization problems (Li et al., 2018), such as Minimum Vertex Cover (MVC), Maximum Cut (MC), and Maximal Independent Set (MIS), by setting different relational queries and targets to make the heuristic function $f(\cdot)$ more universal.

3.2. Markov decision process

In this paper, $f_\theta(G, v_0, q)$ is modeled by a graph-walking agent, which can intelligently traverse all the remaining nodes in the graph and return to the starting point, and requires the shortest sum of traversed paths. The agent must learn a search policy from the training set so that when the training is completed, the agent understands how to traverse all nodes in the graph with minimal cost. The agent is not supervised from start to end but receives only delayed evaluation feedback: the agent is rewarded positively when correctly predicting the optimal path in the training set. For this purpose, we develop the optimal path problem as an MDP to train the agent through RL.

We define the MDP by the tuple (S, A, P, R), where S is the set of states, A is the set of actions, P is the state transition probability, and R is the reward function. Our environment is a limited range, deterministic, and partially observed GAT located on a VRP graph.

Actions. Action space A refers to all actions carried out by the agent in the graph. Every time the agent selects a node v_i, it means that it executes an action $a_t(a_t \in A)$ in the time step t.

States. S represents the state space, and s_t represents the node-set traversed by the agent up to time step t. At s_t, the agent can perform two actions: (1) select an edge e_{ij} and move to the next node. (2) Terminate the walk after returning the start node v_0 (the state becomes s_{end}). The agent needs to make a correct decision based on the traversed node sequence trajectory and the target, so we recursively define s_t:

$$s_t = s_{t-1} \cup a_{t-1}, v_{i,t}, N_{v_{i,t}}, E_{v_{i,t}} \tag{1}$$

Where $v_{i,t}$ is the node visited by the agent in time step t, $N_{v_{i,t}}$ represents the set of adjacent neighbor nodes with $v_{i,t}$, and $E_{v_{i,t}}$ represents the set of edges connected with $v_{i,t}$.

Transition. P refers to the probability that the state changes after the agent take action. $P\left(s_{t-1}, a_{t-1}, s_t\right) = 1$ if and only if s_t represents a partial tour produced by adding $v_{i,t}$ to the partial tour in s_{t-1}.

Rewards. We define the reward function $R(s)$ as the length of the tour, if $s_t = s_{end}$ and $a_t = v_{0,t}$ (represents the complete tour), otherwise $R(s) = 0$.

3.3. Memory components with GRU encoder and graph attention

To solve the finite time domain deterministic part of the observable GAT, we introduce the GRU (Cho et al., 2014) to memorize previously traveled paths (states or actions) for learning random history-dependent policies instead of using supervised learning in AlphaGo (Silver et al., 2016a) and GCOMB (Manchanda et al., 2019) for pre-training. Pre-training with supervised learning requires many known optimal paths for model training, and such "hot starts" may cause the model to overfit on given paths.

The agent-based GRU encodes the state s_t as a continuous vector h_t, $h_t = ENC(s_t)$. We dynamically update the history embedding by the GRU:

$$h_t = GRU(h_{t-1}, [a_{t-1}; o_t]) \tag{2}$$

Where o_t represents a vector representation of observation and $[;]$ denotes vector connection.

Based on historical embedding h_t, the policy network decides to take any action from all available actions A based on the query q. Each possible action represents a node or an outgoing edge with labels and information. To make the graph the input and enable the agent to have a larger field of vision during traversal to judge the weights of paths and nodes better, we introduce the GAT (Veličković et al., 2018) to process the information on the graph.

We employ the complete graph as input rather than a sequence of nodes because we can apply GNN to learn the internal structure of the graph so that we can aggregate and update the information on the graph even when the nodes and edges in the graph are changing dynamically (Fig. 2). As an encoder of the graph, we first obtain the original feature of the node on the graph $e_i = Embed\left(v_i\right), e_i \in R^F$, where F is the dimension of the feature vector, and $Embed()$ can be a fully connected neural network such as FCN, MLP, etc. (we choose MLP in this paper). Then we apply each layer of the GAT to update the feature vector of the $i-th$ node, and calculate the attention weights from node i to node j as follows:

$$a_{ij} = a(We_i, We_j) \tag{3}$$

Where a is a mapping of $R^{F'} \times R^{F'} \rightarrow R$, and $W \in R^{F' \times F}$ is a weight matrix. To ensure that the structural information on the graph is not lost, we apply masked attention, which assigns attention to the adjacent node set of the node v_i. Normalized self-attention shows below:

$$\alpha_{ij} = softmax_j\left(a_{ij}\right) = \frac{\exp(a_{ij})}{\sum_{k \in N_i} \exp(a_{ik})} \tag{4}$$

We employ a single layer neural network a, and the specific calculation process is as follows:

$$\alpha_{ij} = \frac{\exp(LeakyReLU(\vec{a}^T[We_i \parallel We_j]))}{\sum_{k \in N_{v_i}} \exp(LeakyReLU(\vec{a}^T[We_i \parallel We_k]))} \tag{5}$$

Where $\vec{a}^T \in R^{2F'}$ is the parameter of the feedforward neural network a and $LeakyReLU$ is the activation function. We can get the updated vector:

$$e_i' = \sigma(\sum_{j \in N_{v_i}} \alpha_{ij} We_j) \tag{6}$$

To improve the representative ability of the model, we apply multiple W^k to calculate self-attention at the same time, and then combine the results obtained by each W^k:

$$e_i' = \parallel_{k=1}^{K} \sigma(\sum_{j \in N_{v_i}} \alpha_{ij}^k W^k e_j) \tag{7}$$

Where $\parallel$ represents the connection, and α_{ij}^k and W^k represent the calculation result of the $k-th$ head. We can also take the summation to get e_i':

$$e_i' = \sigma(\frac{1}{K}\sum_{k=1}^{K} \sum_{j \in N_{v_i}} \alpha_{ij}^k W^k e_j) \tag{8}$$

We can also replace GAT with the state-of-the-art DAGN (Wang et al., 2020), a principled approach incorporating multi-hop adjacent contexts into attention calculations, supporting remote interaction at each level. The state vector of the node v_i at the time t can be calculated using the following formula:

$$s_{i,t} = [[a_{t-1}; o_t]; h_t; e_i'] \tag{9}$$

3.4. Joint modeling policy and value with self-play neural network

Parameter sharing is meaningful for reinforcement learning algorithms with policy networks and value networks because it can effectively improve the efficiency of training and sampling. For example, previous reinforcement learning methods (Bello et al., 2017; Nazari et al., 2018) based on the actor-critic framework for combinatorial optimization often directly model the encoders of the policy network and the value network as precisely the same. In this way, they can share parameters so that the output of the value network can be used as a baseline to evaluate policies. However, compared to only one encoder network, two independent networks using the same architecture are more costly and more unstable for training. More importantly, it often requires a two-stage method of pre-training and fine-tuning to learn parameters, which will reduce the efficiency of parameter learning compared to end-to-end training.

To solve these challenges, we combine GRU encoding and the MCTS model the policy network and value network of parameter sharing in a self-playing way. We apply both the policy $\pi_\theta(a_t|s_t)$ and the Q_θ, where

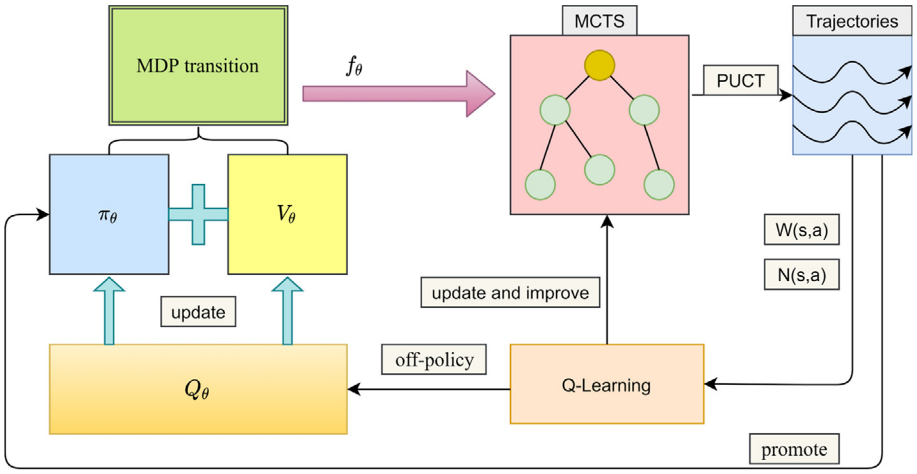


Fig. 3. Q-learning with MCTS is applied to simultaneously model and train the policy network and the value network.

θ is the model parameter. $\pi_\theta(a_t|s_t)$ represents the probability that the agent performs the action a_t in the current state s_t, which is regarded as prior to the bias of the MCTS. Q_θ defines the long-term reward obtained by following the optimal policy and taking the action a_t at the state s_t. We aim to learn a policy that can maximize long-term rewards so that the agent can predict nearby target nodes with high probability and eventually return to the original node to form a tour (Fig. 3 shows the complete process of our modeling and training).

As described in Section 3.3, h_t is composed of the following parts: vector $h_{S,t}$ encodes $(s_{t-1}, a_{t-1}, v_{i,t})$, i.e., the state, action, and current node at the last time; $h_{v'_{i,t},t}(v'_{i,t} \in N_{v_{i,t}})$ integrates neighbor nodes $N_{v_{i,t}}$ and edges $E_{v_{i,t}}$; $h_{A,t}$ judges whether the agent arrives at the destination based on $N_{v_{i,t}}$ and $E_{v_{i,t}}$. We can model π_θ and Q_θ by the following formula:

$$x_0 = f_{\theta_\pi}([h_{S,t}, h_{A,t}]) \tag{10}$$

$$x_{v'_{i,t},t} = \langle h_{S,t}, h_{v'_{i,t},t} \rangle \tag{11}$$

$$V(s_t) = Q_\theta\left(s_t, \cdot\right) = Sigmoid(x_0, x_{v'_1}, \ldots, x_{v'_m}) \tag{12}$$

$$P(s_t, a_t) = \pi_\theta\left(\cdot|s_t\right) = Softmax_\tau(x_0, x_{v'_1}, \ldots, x_{v'_m}) \tag{13}$$

Where x_0 is obtained by stitching $h_{S,t}$ and $h_{A,t}$ together through a full-connected neural network $f_{\theta_\pi}()$. $x_{v'_{i,t},t}$ is given by the inner product of $h_{S,t}$ and $h_{v'_{i,t},t}$. $x_{v'_m}$ refers to the neighbor's score, Q is obtained by the $Sigmoid$ function, and π is obtained by the $softmax$ function with the "temperature" parameter τ (Hinton et al., 2015).

3.5. Key components of Alphago Zero

Given a state s, Alphago Zero (Silver et al., 2017b) trains a neural network f_θ with parameters θ by applying deep reinforcement learning. The output of the neural network is

$$f_\theta(s) = (\boldsymbol{p}, v) \tag{14}$$

Alphago Zero improves its network through self-play. It uses the two predictions in Eq. (15) in the Monte Carlo Tree Search (MCTS) to refine the evaluation of a board position. The policy network P assigns weights to candidate moves when looking at the board "at first glance" and imitates the enhanced policy by MCTS $\pi^{MCTS} = \alpha_\theta(s)$. The output v is the neural network evaluation function of the position s, which simulates the player's reward z in the game (for example, if the player wins, then $z = 1$, otherwise $z = 1$). Alphago Zero statistically estimates the game results after each move through MCTS and repeatedly simulates how to expand to a certain ply depth. In each MCTS simulation, f_θ is recursively applied to a sequence of positions (or nodes) until a certain layer depth. Alphago Zero uses Eq. (15) to evaluate and "backup" the evaluation to the root node at the maximum layer depth. Each node adjusts its "action selection rules" to change the moves that will be selected and expanded in the next MCTS simulation. After many MCTS simulations, the most frequently accessed root move will be performed. Specifically, Alphago Zero learns to minimize the following loss:

$$L = (z - v)^2 - \pi^{MCTS^T} log\boldsymbol{p} + c\,\|\theta\|_2^2 \tag{15}$$

Where c is the non-negative constant of L_2 regularization to prevent overfitting.

Each node in the Monte Carlo tree corresponds to a state s, and the root denotes the initial state. Each edge (s, a) represents an action a on state s and stores a tuple:

$$\{N(s,a), W(s,a), Q(s,a), P(s,a)\} \tag{16}$$

Here $N(s,a)$ is the number of visits, $P(s,a)$ is the prior probability, $W(s,a)$ and $Q(s,a)$ are the total action value and the average action value, respectively.

An iteration of MCTS consists of three parts: select, expand, and backup. A core problem of combining reinforcement learning and search algorithms is maintaining a balance between the exploration of fewer simulations moves and the exploration of moves with a high average win rate.

Select. The first in-tree stage of each simulation starts from the root node s_0 of the search tree, and ends when the simulation reaches a leaf node s_L at time step L, and $t < L$. AlphaGo Zero (Silver et al., 2016b) choose an action by the statistics $a_t = argmax_a(Q\left(s_t, a\right) + U(s_t, a))$, and applies a variant of the PUCT algorithm (PUCB) (Rosin, 2011) to calculate $U(s,a)$ and handle this balance.

$$U(s,a) = c_{PUCT} * P(s,a) * \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \tag{17}$$

Here c_{PUCT} is the non-negative constant in the selection process. The MCTS+UCB combined algorithm initially focuses on moves with high prior probability and low number of visits, and finally prefers moves with high $Q(s_t, a)$ value.

Expand. If a leaf node s_L for exploration is encountered, the expansion process uses network prediction to initialize the edge value (Eq. (17)):

$$\{N\left(s_L, a\right) = 0, W\left(s_L, a\right) = 0, Q\left(s_L, a\right) = 0, P\left(s_L, a\right) = p_a\} \tag{18}$$

Backup. After expanding a new leaf node, we traverse each visited edge and update its edge value so that Q maintains the average value of the state evaluation in the simulation:

$$N\left(s_t, a_t\right) = N\left(s_t, a_t\right) + 1 \tag{19}$$

$$W\left(s_t, a_t\right) = W\left(s_t, a_t\right) + v \tag{20}$$

$$Q(s,a) = W\left(s_t,a_t\right)/N\left(s_t,a_t\right) \tag{21}$$

Play. After taking an action a, the sum of the states we get from s. And π^{MCTS} proportional to the number of moves each time:

$$\pi^{MCTS}\left(a|s_0\right) = N(s_0,a)^{1/\tau} / \sum_b N(s_0,b)^{1/\tau} \tag{22}$$

Where τ is the "temperature parameter" that controls the level of exploration.

3.6. Training with MCTS

We can understand the AlphaGo Zero self-play algorithm (Silver et al., 2017b) as an approximate policy iteration scheme, where MCTS is used for policy improvement and policy evaluation. Policy improvement starts with a neural network policy, implements an MCTS according to the guidance of the policy, and then projects a more assertive search policy back to the function space of the neural network. Policy evaluation (value network) evaluates the search strategy that has been improved and projects the results of self-playing games back into the function space of the neural network. AlphaGo Zero implements these projection steps by training the neural network parameters to match the search probabilities and self-play results, respectively.

Based on the above discussion, we begin to introduce our training algorithm in detail (we summarize the complete process of the proposed algorithm in Algorithm 1). We also apply the self-play algorithm to train our collaborative network. In this approach, the current policy network is trained to fight against previous iterations of itself. We first generate a feasible complete tour by iteratively selecting the best node predicted by the policy network. Then at each time step t, Omega-Zero runs the previous iteration of the policy network to get π, a distribution on unused nodes, and sample from this distribution to execute MCTS.

We apply the ranking reward mechanism (R2) (Laterre et al., 2018) to compare each agent's solution with its recent rather than initial performance. It must surpass its current self to receive positive rewards. We also use the buffer B to record the most recent Markov decision process (MDP) reward and calculate the threshold r_α based on the given percentile $\alpha \in (0, 100)$. Recalling the previous MDP reward $R(S)$, we reshape it into a graded reward $z \in \pm 1\}$ according to whether it exceeds the threshold value:

$$z = \begin{cases} 1, R_t(S) > r_\alpha \\ -1, R_t(S) < r_\alpha \\ b \sim B, R_t(S) = r_\alpha \end{cases} \tag{23}$$

Here $z = b$ is the sample sampled in the binary variable B, when $R_t(S) = r_\alpha$, z is randomly equal to 1. We compare the instant reward $R_t(S)$ at time step t with the threshold r_α to obtain the current instance sample of the agent and use random variables to break the ties to ensure continuous learning.

Following AlphaGo Zero's self-play training pipeline (Silver et al., 2017b), OmegaZero's training consists of three main components: data generator, learner, and evaluator. The data generator generates new self-play data based on the MCTS of the current best model, which is the self-play record (s, a, π, z) for a randomly generated graph input. The learner randomly selects a small batch of samples from the generator's data and updates the best model parameters to minimize the loss in Eq. (16). OmegaZero does not need to evaluate the performance of different players but instead evaluates the average performance of each generated route.

Recalling the parameter-sharing joint network (including policy network and value network) that we built using GAT and GRU, we define its output P and V based on Eq. (15) as $f_\theta\left(s_t\right) = (P(s_t,a_t), V(s_t))$. Given the initial state s_0, we repeat the iteration of MCTS: "select", "expand", and "backup". Unlike AlphaGo Zero, in the "select" phase, we designed a novel reverse link algorithm that adopts MCTS to leverage MDP transfer functions. Specifically, in each MCTS simulation, we expand the trajectory from the root state s_0 by selecting actions based on the PUCB algorithm (Rosin, 2011):

$$a_t = argmax_a\{\beta \cdot \frac{\pi_\theta\left(a|s_t\right)^\mu \sqrt{\sum_{a'} N\left(s_t,a'\right)}}{1+N\left(s_t,a\right)} + \frac{W\left(s_t,a\right)}{N(s_t,a)}\} \tag{24}$$

Here β and μ are constants. OmegaZero aims to run multiple simulations to generate trajectories $(\rho = \langle s_t, a_t, z_{t+1}\rangle)$ with positive rewards, which is similar to the improvement MCTS policy π^{MCTS} in AlphaGo Zero. We can learn these trajectories while improving policies in reverse.

Our policy network and value network are parameter-sharing. The parameters of the policy network are updated with the update of the parameters of the value network. Compared to using the less stable actor-critic algorithm (Mousavi et al., 2018), we apply the Q-learning algorithm (Jin et al., 2018; Gölcük and Ozsoydan, 2021) to directly update the parameters of our Q-network in an off-policy manner. We apply the Temporal-Difference (TD) method (Silver et al., 2012) to update the Q value:

$$\theta \leftarrow \theta + \alpha \cdot \nabla_\theta Q_\theta(s_t,a_t) \times (z + \gamma max_{a'} Q_\theta\left(s',a'\right) - Q_\theta(s_t,a_t)) \tag{25}$$

Where α is the learning rate, γ is the rewarding decay coefficient. s' and a' are the state and action of the next time step, respectively.

3.7. Predict nodes with the MCTS scoring function

In the test phase, we generally apply the trained policy to predict the sequence of nodes (paths) on the unseen (unknown) graph. Previous methods predict the probability distribution of nodes on the unknown graph, such as the greedy probability distribution scoring function used in GCOMB (Manchanda et al., 2019). In contrast, our method combines the policy and value learned by MCTS to generate an MCTS tree during the training phase. In the testing phase, previous methods often predict multiple paths based on different rewards, but in practice, we often need an exact shortest path, which means that the result we need should be unique. When OmegaZero predicts multiple paths, leaf states in the MCTS tree correspond to nodes on different paths, and we need to combine the predicted results of MCTS leaf states into a score to sort the nodes, specifically:

$$Score(v_i) = \sum_{s \to v_i} \frac{N(s,a)}{N} \times Q_\theta(s, v_{0,t}) \tag{26}$$

Where N is the sum of all leaf states s corresponding to the same node v_i. $v_{0,t}$ is an action in the terminated state (see Section 3.2). $Score(v_i)$ is the weighted average value of terminal state values associated with the same candidate node v_i. Among the candidate nodes (paths) we choose the one with the highest score:

$$v_p = argmax_v Score\left(v_i\right) \tag{27}$$

4. Experiments

4.1. Data sets and settings

As mentioned earlier, we can apply the subgraph sampling algorithm to convert one VRP into multiple TSPs to reduce the complexity of problem-solving because VRP is generally tricky for TSP. The machine learning method we apply is essential to learn the data distribution in the training set. When the test set and the training set have the same distribution, we can train the model to predict the data distribution on the test set. However, the distribution of VRP on the graph is often not standardized, and it is challenging to ensure that the test set and the training set have the same distribution. We apply the subgraph sampling algorithm to preprocess the VRP instances to help improve the accuracy and interpretability of the learning data distribution and can play a role in transforming large problem instances into small ones.

Algorithm 1 Q-learning with MCTS

```
Input: Network $f_\theta$, initialize parameters $\theta_0$ of the network $f_{\theta_0}$, a mini-batch size $b$, root
(initial) node (state) $s_0$, a percentile $\alpha$, a query $q$, initialize fixed-size buffers $B = \{\}$
For episode = $1,\ldots,T$ do
    While $s_t$ is expanded before and $s_t \neq s_{end}$ do
        Perform an MCTS guided by $f_\theta$
        Extract MCTS-improved policy $\pi_\theta(\cdot\,|s_t)$ //Similar to $\pi^{MCTS}$ in AlphaGo Zero
        {Select}
            $a_t = argmax_a\{\beta \cdot \frac{\pi_\theta(a|s_t)^\mu \sqrt{\sum_{a'} N(s_t,a')}}{1+N(s_t,a)} + \frac{W(s_t,a)}{N(s_t,a)}\}$ // Equation (25)
         Take the action $a_t$ , observe next state $s_{t+1}$, and update $f_\theta$
    End while
         Compute MDP reward $R(S)$ and store it in $B$
         Compute threshold $r_\alpha$ based on the MDP rewards in $B$
         Reshape to ranked reward z as explained in Equation (24):
            $z = \begin{cases} 1, R_t(S) > r_\alpha \\ -1, R_t(S) < r_\alpha \\ b\sim B, \quad R_t(S) = r_\alpha \end{cases}$
         {expand}
      If $s_t \neq s_{end}$ then
         Initialize the edge value with Equation (19):
         $\{N(s_L,a) = 0, W(s_L,a) = 0, Q(s_L,a) = 0, P(s_L,a) = p_a\}$
      End if
        {backup}
      While $s_t \neq s_0$ do
           $a = previous\ action$
           $N(s_t,a_t) = N(s_t,a_t) + 1$ // Equation (20)
           $W(s_t,a_t) = W(s_t,a_t) + v$ // Equation (21)
           $Q(s,a) = W(s_t,a_t)/N(s_t,a_t)$ // Equation (22)
       End while
End for
For generating feasible tours, do
     if $s_t = s_{end}$ and $a_t = v_{0,t}$
     Minimize the loss with Q-learning:
   $L = (z-v)^2 - \pi^T logP + c\|\theta\|_2^2$ // Loss of routing problem based on Equation (16)
   $\theta \leftarrow \theta + \alpha \cdot \nabla_\theta Q_\theta(s_t,a_t) \times (z + \gamma max_{a'} Q_\theta(s',a') - Q_\theta(s_t,a_t))$ //Equation (26)
End for
```

Capacitated Vehicle Routing Problem (CVRP) (Xiang et al., 2020) is a more practical variant of VRP, so we generally prefer to study CVRP compared to VRP. Our application of the subgraph sampling algorithm is equivalent to transforming CVRP into processing TSP with capacity constraints in multiple subgraphs. OmegaZero does not require expert experience to learn heuristics through self-play, while data generators can generate data. This paper focuses on path optimization problems, and we can generate VRP or TSP instances to train the model as in previous works (Kool et al., 2019). We employ a data generator to generate Euclidean TSP and CVRP with 20, 50, and 100 nodes and larger-scale TSP instances with 500, 750, and 1000 nodes. We call them TSP20, CVRP20, etc. The coordinates of each node are randomly sampled within the unit square $[0,1] \times [0,1]$. For CVRP, we take the needs of each customer from $1,\ldots,9$; the capacities D of CVRP20, 50, and 100 are 30, 40, and 50, respectively.

GNN. For our memory graph neural network (including GRU, GAT, MLP), we set the GRU hidden dimension to 200 and the attention dimension in GAT to 100. We set the number of MLP layers used to obtain the original features to 5 and set the hidden dimension of each layer to 100.

MCTS. In the Monte Carlo tree search parameter setting, we follow the implementation of Alphgo Zero (Silver et al., 2017b) and ELF (Tian et al., 2019). We set the c in Eq. (16) to 4, and the c_{PUCT} in Eq. (18) to 1.5. MCTS instances perform 128 simulations (for TSP20, TSP50, TSP100) and 300 simulations (for other instances) each time they move. In each iteration, 50 instances are generated and solved by R2 with a reward buffer size of 250, which is used to define the threshold r_α. We set the softmax temperature parameter τ to a constant 1. Another option is to anneal it during training, such as $\pi: 1 \to 0$, but we did not observe significant differences in our experiments.

Training. During the training, each learner sampled 20 trajectories from the self-play record and conducted random gradient descent through Adam, with a learning rate of 0.001, a weight attenuation of 0.0001, and a batch size of 16. The learner saves the new model after 15 iterations. We set the number of data generators, learners, and model evaluators to 20,6 and 2, respectively. Meanwhile, we incorporate PBT (Wu et al., 2020) to optimize the parameters in the network, which is a population-based method for accelerating and improving AlphaZero. Each experiment is performed on eight NVIDIA GeForce GTX1080Ti GPUs for up to one hour.

Baseline. In terms of baseline selection, we choose some classic heuristic algorithms (Kool et al., 2019; Ma et al., 2019a) such as Nearest, MST, etc., solution solvers (Kool et al., 2019) such as Concorde, LKH3, OR-Tools, etc. Recent representative learning-based methods AM (Kool et al., 2019), GCN (Joshi et al., 2019), GPN (Ma et al., 2019a), S2V-DQN (Dai et al., 2017), etc. The current works based on AlphaGo Zero have not been published or focused on the path optimization problem. More importantly, most of them require too much computing resources and training time to reproduce, so we did not use them as a baseline. Since OmegaZero is based on self-play reinforcement learning, which is very different from the previous work in the training mechanism, so we cannot use the exact implementation as the previous work. The results of the baseline in the experiment all come from their papers.

4.2. Performance for small-scale TSP

We first verify the efficiency and effectiveness of OmegaZero on small TSP instances. The solution solvers' optimal solutions of TSP20/50/100 instances are relatively easy to obtain by the solution solvers (such as Concorde, LKH3, and OR Tools (Kool et al., 2019)). Then, the approximate ratio of each method to the optimal solution is applied to measure the quality of the solution obtained by the model, among which the method with the lower approximate ratio is better.

Fig. 4 shows the experimental results of different methods. From the trend, we can see that the solution obtained by OmegaZero is the closest to the optimal solution, which indicates its good effect on small-scale routing problems. Compared with previous methods based on GNN alone, OmegaZero applies GAT with GRU, which effectively integrates and remembers the information of neighboring nodes to form a complete state space and action space to guide agents to find nodes and reasoning paths better. We compared the memory components with GAT alone during the experiment. We found that using GRU to aid attention on a small scale is more accurate than just using attention (combining GRU and GAT can reduce the shortest path by 2%). We believe that attention can infer the importance of the surrounding neighbors so that the GRU can record the historical trajectories more efficiently and avoid redundant states and actions. Moreover, if the agent can remember the paths taken, it is helpful to infer the paths. Besides, the MCTS-based scoring function we designed in the testing phase is also conducive to improving the accuracy of OmegaZero in reasoning.

4.3. Performance for larger-scale TSP

Real-world transportation networks often contain hundreds of nodes, so it is necessary to verify OmegaZero's performance on a larger-scale TSP instance. We train on small-scale graphs and then reason on larger scale graphs, which is another way to demonstrate the model's generalization ability. The parameter amount in the GNN-based method may increase as the scale of the graph instance increases, which may eventually cause the model to be unable to be trained because the parameter amount is too large, which is common in the previous methods (Dai et al., 2017). At the same time, due to the OmegaZero using GRU, which is a variant of RNN, if all training in a large-scale graph could lead to a gradient explosion due to excessive parameters is too heavy to be trained (Manchanda et al., 2019). So for the super-large size graphs, we can apply the subgraph sampling algorithm (Zhang et al., 2018) to divide into subgraphs and iteratively deal with these subgraphs. If there is too much training data, the training time will be too long, or we do not know how long the training time is appropriate because if the training time is too long, it will lead to overfitting.

In contrast, the training time is short, the potential of the model cannot be fully released. To sum up, we trained for one hour on TSP20, TSP50, and TSP100, respectively, and then tested on TSP500, TSP750, and TSP1000. Table 1 shows the comparison results with baselines.

From the experimental results, we can see that OmegaZero achieves better results compared with the previous RL methods. However, it is still not as accurate as solution solvers. We can also observe that OmegaZero performs better when the training data set is larger, which indicates that the increase of training data within a specific range is conducive to improving its ability and unleashing its potential. At present, most learning-based methods are based on generation heuristics. That is, we generate solutions one by one through heuristic decoding. Their most significant advantage over solution solvers is a generalization. Learning-based methods can quickly generalize to similar problem instances or larger-scale instances as long as they learn the data distribution on a given instance. Most of the solution solvers are based on improvement heuristics. They often first produce a feasible solution and then iteratively improve it to get closer and closer to the optimal solution. In large-scale instances, the learning-based method is not as accurate as the solution solver. Still, it far exceeds the latter in time, which is very important for practical applications. Because the distribution learned by training on a small-scale instance is limited, it is reasonable that the accuracy is not as good as the solution-solving ability when testing on a large-scale instance. This experimental task also proves that MCTS-based self-play reinforcement learning can effectively learn heuristics and has good generalization ability.

Table 1
The model performance of OmegaZero trained on small TSP instances (TSP20, TSP50, TSP100) and then reasoned on larger instances compared to the results of the baseline approaches. The indicators measured are the optimal solution obtained by the model and the time required to obtain it when reasoning.

Method	TSP500 Obj. Time	TSP750 Obj. Time	TSP1000 Obj. Time
Concorde	**11.89**, 1894 s	**20.10**, 32993 s	**23.11**, 47804 s
LKH3	11.893, 9792 s	20.129,36840 s	23.130, 50680 s
OR Tools	12.289, 5000 s	22.395, 5000 s	26.477, 5000 s
Nearest Insertion	14.928, 25 s	25.219, 115 s	28.973, 136 s
2-opt	13.253, 303 s	22.668, 3296 s	26.111, 6153 s
Farthest Insertion	13.026, 33 s	22.342, 454 s	25.741, 945 s
GPN	12.942, 214 s	22.541, 2278 s	26.129, 4410 s
S2V-DQN	13.079, 476 s	22.550, 3182 s	26.046, 5600 s
AM	14.032, **2 s**	28.281, 42 s	34.055, 136 s
OmegaZero	(20) 12.857, 4 s	22.524, 35 s	25.698, 120 s
OmegaZero	(50) 12.798, 4 s	21.985, 33 s	24.856, 115 s
OmegaZero	(100) 12.109, 3 s	21.864, **32 s**	24.451, **109 s**

4.4. Performance on CVRP

In theory, we can apply OmegaZero to solve other types of combinational optimization (CO) problems on graphs by simply changing the Markov decision process (MDP) to fit other problems such as MVC, graph coloring, maximum cut problems, etc. (Huang et al., 2019). As mentioned above, this paper focuses on CVRP on the graph, which can be transformed into TSP through a subgraph sampling algorithm. In the CVRP experiment, k is set as 5, that is, the CVRP instance in the experiment is converted into 5 TSP instances. The model trained in TSP20 is used for CVRP100, TSP50 for CVRP200, and TSP100 for CVRP400. We try to establish a method system to solve these kinds of path problems based on AlphaGo Zero, so whether good effects can be maintained on other problems will be reserved for future work. The previous works based on AlphaGo Zero are mostly used in graph coloring problems, 3D packing problems, MVC, etc. (Huang et al., 2019; Laterre et al., 2018; Xing et al., 2020), which did not overlap with our methods in the experiment, and such methods were often too complex to be easily repeated, so we still chose the targeted method to solve VRP as the baselines.

From the experimental results, we can see that OmegaZero is still very competitive compared with baselines, which mainly proves the effectiveness of the subgraph sampling algorithm for the conversion of VRP to TSP because we have previously proved the efficiency of OmegaZero on TSP. We could have handled VRP directly by changing the MDP without the need for transformation, but we try to unlock the possibility of fast parallel processing of large-scale graphs. In the CO problems, a lot of problems have mathematical equivalence, for

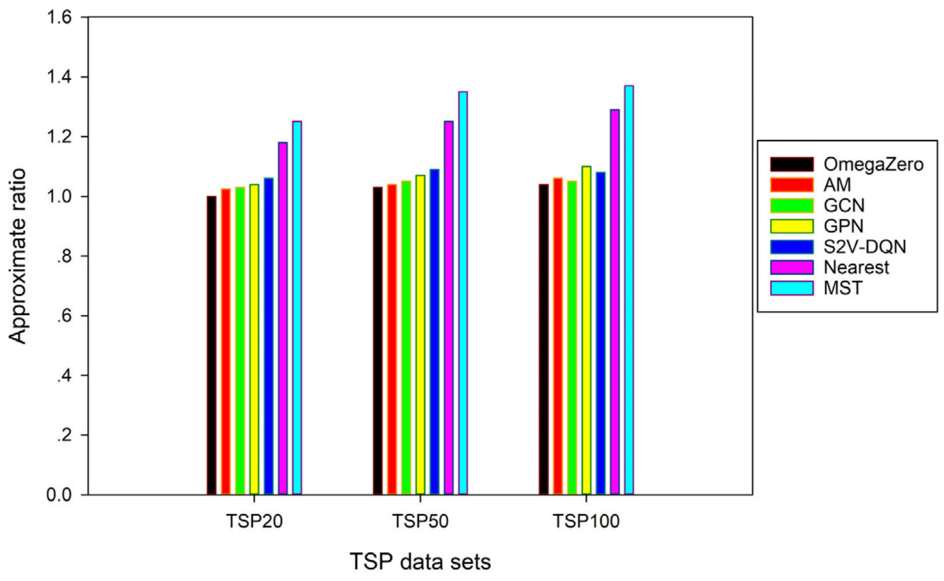


Fig. 4. The approximate ratio of the solutions generated by different methods on small TSP instances to the optimal solutions.

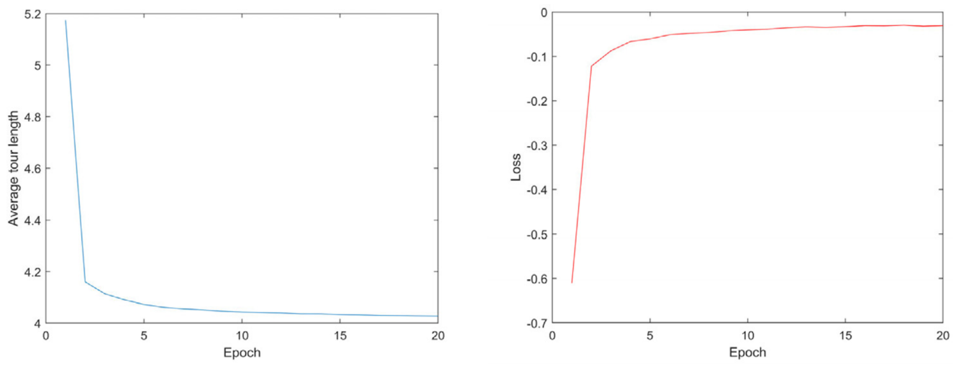


Fig. 5. The learning curve we got by running OmegaZero on TSP20 demonstrates the changes in the average route length and loss obtained by OmegaZero as iterations increase.

example, Independent Set (MIS), Minimum Vertex Cover (MVC), and Maximal Clique (MC) in the bipartite graph (Li et al., 2018). We can apply machine learning to convert them to each other to make it possible to solve multiple problems using only one model that does not need to be modified. Compared with other learning-based methods, the experimental results also prove our superiority in modeling and learning. Compared with simply using the combination of GNN and reinforcement learning, our memory graph neural network can capture the information of nodes and edges in the graph instance and assist end-to-end self-play learning through the memory state. And the linked policy network and value network also make parameter updates easier. These are probably the reasons why OmegaZero can learn better distribution (see Table 2).

Table 2
The performance of OmegaZero in the VRP instances. We employ the solution solver OR-Tools as a benchmark to calculate the performance of each model relative to it, where "Obj" represents the length of the optimal solution obtained, and "Gap" represents the Gap between each model and "OR-Tools". To facilitate the comparison, we choose the most representative methods recently. The performance of the methods with different search algorithms is different, and we choose the most effective version.

Method	VRP100		VRP200		VRP400	
	Obj	Gap	Obj	Gap	Obj	Gap
OR-Tools	11.348	0.00%	17.995	0.00%	26.095	0.00%
PRL (Nazari et al., 2018)	11.907	4.93%	19.521	8.48%	28.846	10.54%
AM (Kool et al., 2019)	11.746	3.51%	18.697	3.90%	27.143	4.02%
OmegaZero	**11.271**	**−0.68%**	**17.457**	**−3.00%**	**26.069**	**−0.1%**

4.5. Effect of training

So far, there are still some problems in the application of RL, such as sparse rewards, low sampling efficiency, limited search space, etc. Because, under normal circumstances, agents may find many useless nodes or paths in constant trial and error or enter an endless cycle in a specific state. Even the effect of reinforcement learning in the AlphaGo series is generally considered to be achieved due to MCTS and deep learning. We will then verify the learning effect of OmegaZero, whether it can learn valid paths and find the right nodes efficiently.

We train OmegaZero on TSP20, and it converges to the shortest path after 20 epochs. As shown in Fig. 5, OmegaZero converges quickly in the initial stage of training and gradually stabilizes as the training continues. The experiment proves the efficiency and effectiveness of our self-play reinforcement learning.

As mentioned earlier, we apply the R2 algorithm to reshape the reward function $R(s)$ to $z \in \{\pm 1\}$. We examined the actual reward situation of OmegaZero without the R2 algorithm. We ran 20 epochs on TSP10 to observe how OmegaZero gets rewards at each step, and the reward value is the shortest path that has been obtained.

Fig. 6 shows how the reward and loss in each epoch change as the time step increases. We choose the first six epochs because the learning curve has converged after the first epoch. From the figure, we can see that the reward changes significantly in the first epoch, and

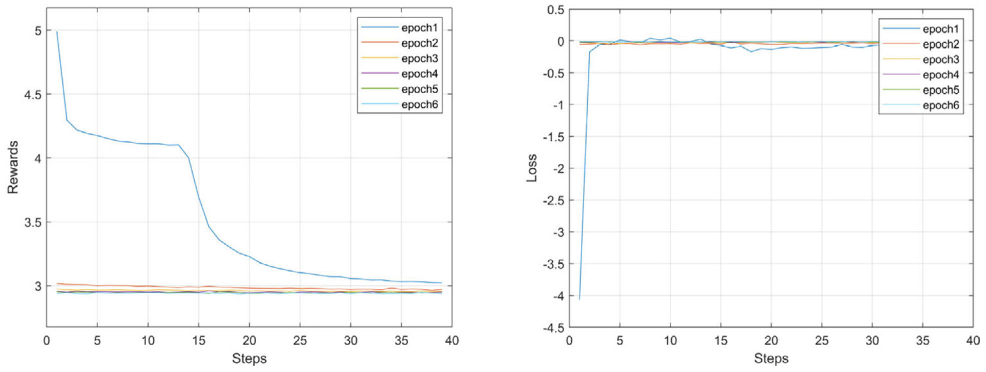


Fig. 6. We select the rewards and losses in the first six epochs of OmegaZero running on TSP10 to generate a learning curve. From the figure, we can see the changes in rewards and losses as time steps (100 batches in one time step) increase in each iteration.

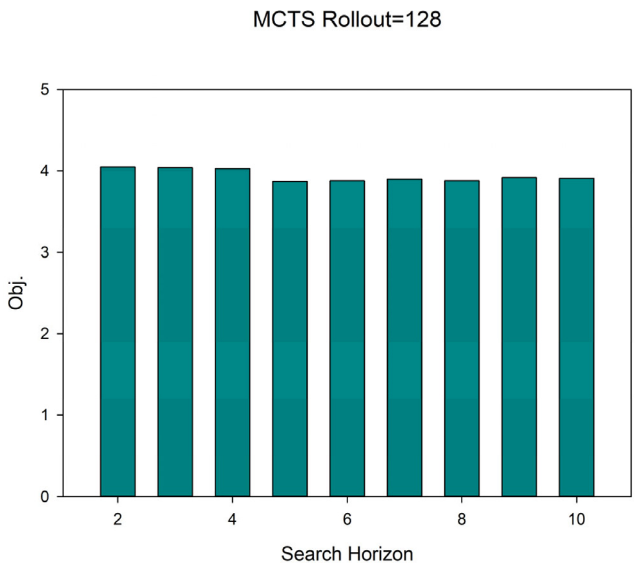


Fig. 7. We fixed the MCTS rollout to 128. The length of the optimal solution obtained by OmegaZero changes when the search horizon is 2, 3, 4, 5, 6, 7, 8, 9, and 10.

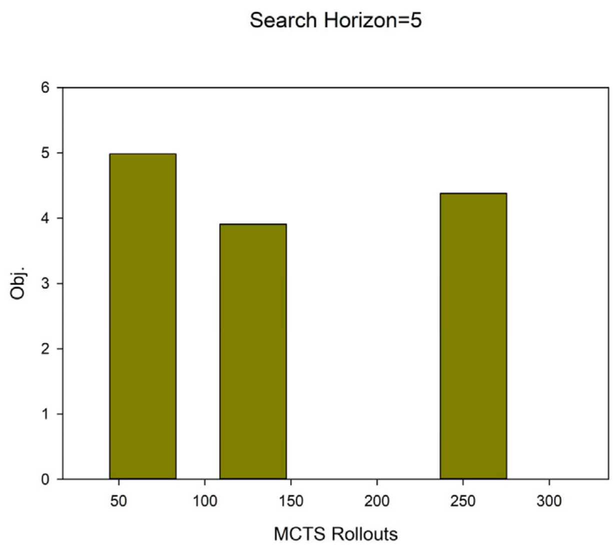


Fig. 8. We fixed Horizon to 5. The length of the optimal solution obtained by OmegaZero changes when the MCTS rollouts are 64, 128, and 256, respectively.

in subsequent epochs, it almost stabilizes near a constant value. It is also more conducive for us to apply the R2 algorithm to reshape the reward to a fixed value. Similarly, the loss also stabilizes after the first epoch, verifying that our self-play reinforcement learning can quickly converge.

Besides, we also conducted an MCTS hyperparameter analysis. We still analyzed the performance of OmegaZero under different numbers of MCTS rollout simulations and different search horizons on TSP20, and the results are shown in Figs. 7, 8. We observe that OmegaZero is more sensitive to the number of MCTS rollouts and less sensitive to searching for horizons.

5. Conclusion

We designed OmegaZero inspired by Alphago Zero, which does not require expert experience and can learn heuristics entirely from unlabeled data through self-play. Aiming at path optimization problems (including TSP, VRP, etc.), we have studied a lightweight self-playing framework based on MCTS, whose consumption is far lower than Alphago Zero. Since the memory component can record the history trajectories to assist the modeling of the policy network and value network later, this is equivalent to saving many computing resources in the early stage and avoiding the supervised pre-training in AlphaGo. We apply the Q-learning algorithm with MCTS to train the policy network and the value network to be iteratively updated simultaneously to produce the best model, which simultaneously effectively integrates learners, data generators, and model evaluators. Like Alphago Zero, which plays not just Go but a variety of board games, OmegaZero can be generalized to other similar combinatorial optimization problems on a much larger scale. It also shows that combinatorial optimization is one of the most cutting-edge problems in artificial intelligence. Its scientific value and practical application value make us try to apply state-of-the-art technology to it. In this paper, we propose a general RL framework in which components can be replaced by other models, such as the state-of-the-art GNNs and RL, which will be our work in the future.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Aqil, S., Allali, K., 2021. Two efficient nature inspired meta-heuristics solving blocking hybrid flow shop manufacturing problem. Eng. Appl. Artif. Intell. 100, 104196.

Badia, A.P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, D., Blundell, C., 2020. Agent57: Outperforming the atari human benchmark. ArXiv.

Barrett, T., Clements, W., Foerster, J., Lvovsky, A., 2020. Exploratory combinatorial optimization with reinforcement learning. In: Proc. AAAI Conf. Artif. Intell., vol. 34. pp. 3243–3250.

Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S., 2017. Neural combinatorial optimization with reinforcement learning. In: 5th Int. Conf. Learn. Represent. ICLR 2017 - Work. Track Proc. pp. 1–15.

Bengio, Y., Lodi, A., Prouvost, A., 2021. Machine learning for combinatorial optimization: A methodological tour d'horizon. European J. Oper. Res. 290, 405–421.

Botvinick, M., Ritter, S., Wang, J.X., Kurth-Nelson, Z., Blundell, C., Hassabis, D., 2019. Reinforcement learning, fast and slow. Trends Cogn. Sci. 23, 408–422.

Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S., 2012. A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intell. AI Games 4, 1–43.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In: EMNLP 2014 - 2014 Conf. Empir. Methods Nat. Lang. Process. Proc. Conf. pp. 1724–1734.

Dai, H., Khalil, E.B., Zhang, Y., Dilkina, B., Song, L., 2017. Learning combinatorial optimization algorithms over graphs. Adv. Neural Inf. Process. Syst. 2017-Decem, 6349–6359.

Dehghan-Sanej, K., Eghbali-Zarch, M., Tavakkoli-Moghaddam, R., Sajadi, S.M., Sadjadi, S.J., 2021. Solving a new robust reverse job shop scheduling problem by meta-heuristic algorithms. Eng. Appl. Artif. Intell. 101, 104207.

Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., Rousseau, L.M., 2018. Learning heuristics for the tsp by policy gradient. In: Lect. Notes Comput. Sci. (Including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics). In: 10848 LNCS, pp. 170–181.

Drori, I., Kharkar, A., Sickinger, W.R., Kates, B., Ma, Q., Ge, S., Dolev, E., Dietrich, B., Williamson, D.P., Udell, M., 2020. Learning to solve combinatorial optimization problems on real-world graphs in linear time. In: Proc. - 19th IEEE Int. Conf. Mach. Learn. Appl. ICMLA 2020. pp. 19–24.

Duan, L., Zhan, Y., Hu, H., Gong, Y., Wei, J., Zhang, X., Xu, Y., 2020. Efficiently solving the practical vehicle routing problem: A novel joint learning approach. In: Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min. pp. 3054–3063.

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K.O., Clune, J., 2021. First return, then explore. Nature 590, 580–586.

Emami, P., Pardalos, P.M., Elefteriadou, L., Ranka, S., 2020. Machine learning methods for data association in multi-object tracking. ACM Comput. Surv. 53, 1–35.

Gölcük, İ., Ozsoydan, F.B., 2021. Q-learning and hyper-heuristic based algorithm recommendation for changing environments. Eng. Appl. Artif. Intell. 102.

Goyal, S., 2010. A Survey on travelling salesman problem. In: Midwest Instr. Comput. Symp. pp. 1–9.

Guez, A., Weber, T., Antonoglou, I., Simonyan, K., Vinyals, O., Wierstra, D., Munos, R., Silver, D., 2018. Learning to search with MCTSnets. In: 35th Int. Conf. Mach. Learn., Vol. 4. ICML 2018. pp. 2920–2931.

Hafner, D., Lillicrap, T., Norouzi, M., Ba, J., 2020. Mastering atari with discrete world models.

Hinton, G., Vinyals, O., Dean, J., 2015. Distilling the knowledge in a neural network. pp. 1–9.

Hopfield, J.J., Tank, D.W., 1985. Neural computation of decisions in optimization problems. Biol. Cybernet. 52, 141–152.

Huang, J.P., Pan, Q.K., Miao, Z.H., Gao, L., 2021. Effective constructive heuristics and discrete bee colony optimization for distributed flowshop with setup times. Eng. Appl. Artif. Intell. 97, 104016.

Huang, J., Patwary, M., Diamos, G., 2019. Coloring big graphs with AlphaGoZero. ArXiv.

Ivanov, S., D'yakonov, A., 2019. Modern deep reinforcement learning algorithms. ArXiv.

Jaderberg, M., Czarnecki, W.M., Dunning, I., Marris, L., Lever, G., Castañeda, A.G., Beattie, C., Rabinowitz, N.C., Morcos, A.S., Ruderman, A., Sonnerat, N., Green, T., Deason, L., Leibo, J.Z., Silver, D., Hassabis, D., Kavukcuoglu, K., Graepel, T., 2019. Human-level performance in 3D multiplayer games with population-based reinforcement learning. Science 364 (80-.), 859–865.

Jin, C., Allen-Zhu, Z., Bubeck, S., Jordan, M.I., 2018. Is Q-learning provably efficient? Adv. Neural Inf. Process. Syst. 2018-Decem, 4863–4873.

Jordan, M.I., Mitchell, T.M., 2015. Machine learning: Trends, perspectives, and prospects. Nature 349.

Joshi, C.K., Laurent, T., Bresson, X., 2019. An efficient graph convolutional network technique for the travelling salesman problem. pp. 1–17, ArXiv.

Kool, W., Van Hoof, H., Welling, M., 2019. Attention, learn to solve routing problems! In: 7th Int. Conf. Learn. Represent. ICLR 2019. pp. 1–25.

Laterre, A., Fu, Y., Jabri, M.K., Cohen, A.-S., Kas, D., Hajjar, K., Dahl, T.S., Kerkeni, A., Beguir, K., 2018. Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization. ArXiv.

Lecun, Y., Bengio, Y., Hinton, G., 2015. Deep learning. Nature 521, 436–444.

Li, Z., Chen, Q., Koltun, V., 2018. Combinatorial optimization with graph convolutional networks and guided tree search. Adv. Neural Inf. Process. Syst. 2018-Decem, 539–548.

Littman, M.L., 2015. Reinforcement learning improves behaviour from evaluative feedback. Nature 521, 445–451.

Ma, Q., Ge, S., He, D., Thaker, D., Drori, I., 2019a. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. ArXiv.

Ma, X., Zhu, Q., Zhou, Y., Li, X., Wu, D., 2019b. Improving question generation with sentence-level semantic matching and answer position inferring. ArXiv.

Manchanda, S., Mittal, A., Dhawan, A., Medya, S., Ranu, S., Singh, A., 2019. Learning heuristics over large graphs via deep reinforcement learning. Assoc. Adv. Artif. Intell..

Meng, K., Tang, Q., Zhang, Z., Yu, C., 2021. Solving multi-objective model of assembly line balancing considering preventive maintenance scenarios using heuristic and grey wolf optimizer algorithm. Eng. Appl. Artif. Intell. 100, 104183.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D., 2015. Human-level control through deep reinforcement learning. Nature 518, 529–533.

Mor, A., Speranza, M.G., 2020. Vehicle routing problems over time: a survey. 4OR 18, 129–149.

Mousavi, S.S., Schukat, M., Howley, E., 2018. Deep reinforcement learning: An overview. Lect. Notes Netw. Syst. 16, 426–440.

Nazari, M., Oroojlooy, A., Takáč, M., Snyder, L.V., 2018. Reinforcement learning for solving the vehicle routing problem. Adv. Neural Inf. Process. Syst. 2018-Decem, 9839–9849.

Nowak, A., Villar, S., Bandeira, A.S., Bruna, J., 2018. Revised note on learning quadratic assignment with graph neural networks. In: 2018 IEEE Data Sci. Work. DSW 2018 - Proc., Vol. 1706. pp. 229–233.

Pierrot, T., Ligner, G., Reed, S., Sigaud, O., Perrin, N., Laterre, A., Kas, D., Beguir, K., de Freitas, N., 2019. Learning compositional neural programs with recursive tree search and planning. Adv. Neural Inf. Process. Syst. 32.

Romero-Hdz, J., Saha, B.N., Tstutsumi, S., Fincato, R., 2020. Incorporating domain knowledge into reinforcement learning to expedite welding sequence optimization. Eng. Appl. Artif. Intell. 91, 103612.

Rosin, C.D., 2011. Multi-armed bandits with episode context. Ann. Math. Artif. Intell. 61, 203–230.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., Silver, D., 2020. Mastering Atari, Go, chess and shogi by planning with a learned model. Nature 588, 604–609.

Selsam, D., Lamm, M., Bünz, B., Liang, P., Dill, D.L., De Moura, L., 2019. Learning a SAT solver from single-bit supervision. In: 7th Int. Conf. Learn. Represent. ICLR 2019. pp. 1–11.

Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D., 2016a. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D., 2017a. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. pp. 1–19, ArXiv.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D., 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science 362 (80-.), 1140–1144.

Silver, D., Schrittwieser, J., Simonyan, K., 2016b. I.A.- nature, U. 2017, mastering the game of go without human knowledge. Nature 550, 354.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Van Den Driessche, G., Graepel, T., Hassabis, D., 2017b. Mastering the game of Go without human knowledge. Nature 550, 354–359.

Silver, D., Sutton, R.S., Müller, M., 2012. Temporal-difference search in computer Go. Mach. Learn. 87, 183–219.

Sobieczky, H., 2020. A learning-based iterative method for solving vehicle routing problems. In: Iclr, Vol. 3. pp. 3–5.

Tian, Y., Ma, J., Gong, Q., Sengupta, S., Chen, Z., Pinkerton, J., Lawrence Zitnick, C., 2019. Elf OpenGo: An analysis and open reimplementation of Alphazero. In: 36th Int. Conf. Mach. Learn. ICML 2019, pp. 10885–10894.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 5999–6009.

Veličković, P., Casanova, A., Liò, P., Cucurull, G., Romero, A., Bengio, Y., 2018. Graph attention networks. In: 6th Int. Conf. Learn. Represent. ICLR 2018 - Conf. Track Proc. pp. 1–12.

Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J.P., Jaderberg, M., Vezhnevets, A.S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T.L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., Silver, D., 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature 575, 350–354.

Vinyals, O., Bengio, S., Kudlur, M., 2016. Order matters: Sequence to sequence for sets. In: 4th Int. Conf. Learn. Represent. ICLR 2016 - Conf. Track Proc. pp. 1–11.

Vinyals, O., Fortunato, M., Jaitly, N., 2015. Pointer networks. Adv. Neural Inf. Process. Syst. 2015-Janua, 2692–2700.

Wang, G., Ying, R., Huang, J., Leskovec, J., 2020. Direct multi-hop attention based graph neural network. pp. 1–15.

Wiseman, S., Rush, A.M., 2016. Sequence-to-sequence learning as beam-search optimization. In: Conf. Empir. Methods Nat. Lang. Process. Proc. EMNLP 2016. pp. 1296–1306.

Wu, T.R., Wei, T.H., Wu, I.C., 2020. Accelerating and improving AlphaZero using population based training. ArXiv.

Xiang, X., Qiu, J., Xiao, J., Zhang, X., 2020. Demand coverage diversity based ant colony optimization for dynamic vehicle routing problems. Eng. Appl. Artif. Intell. 91, 103582.

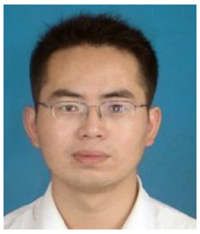
Xing, Z., Tu, S., Xu, L., 2020. Solve traveling salesman problem by monte carlo tree search and deep neural network. ArXiv.

Xu, K., Jegelka, S., Hu, W., Leskovec, J., 2019. How powerful are graph neural networks? In: 7th Int. Conf. Learn. Represent. ICLR 2019. pp. 1–17.

Zhang, J., Cui, L., Gouza, F.B., 2018. SeGEN: Sample-ensemble genetic evolutionary network model. pp. 1–12, ArXiv.

Qi Wang received the B.S. and M.Eng. degree in software engineering from Jilin University (Changchun city) and Central South University (Changsha city) in 2012 and 2016, China, respectively. He is currently pursuing the PH.D. degree in software engineering at the school of computer science, Fudan University, Shanghai, China.

His current research interests include combinatorial optimization, deep learning, and reinforcement learning.



Yongsheng Hao received his MS Degree of Engineering from Qingdao University in 2008. Now, he is a senior engineer of Network Center, Nanjing University of Information Science & Technology. His current research interests include distributed and parallel computing, mobile computing, Grid computing, web Service, particle swarm optimization algorithm and genetic algorithm. He has published more than 30 papers in international conferences and journals.

Jie Cao received the Ph.D. degree from Southeast University, Nanjing, China, in 2005. He was an Associate Professor, from 1999 to 2006. From 2006 to 2009, he was a Postdoctoral Fellow of the Academy of Mathematics and Systems Science, Chinese Academy of Science. From 2009 to 2019, he was a Professor with the School of Management and Economics, Nanjing University of Information Science and Technology. Since May 2019, he has been the Vice President of the Xuzhou University of Technology. His research interests include system engineering, and management science and technology.