Reversible Action Design for Combinatorial Optimization with Reinforcement Learning

Fan Yao[1], **Renqin Cai**[1], **Hongning Wang**[1]

[1] Department of Computer Science, University of Virginia, USA
fy4bc@virginia.edu, rc7ne@virginia.edu, hw5x@virginia.edu,

Abstract

Combinatorial optimization problem (COP) over graphs is a fundamental challenge in optimization. Reinforcement learning (RL) has recently emerged as a new framework to tackle these problems and has demonstrated promising results. However, most RL solutions employ a greedy manner to construct the solution incrementally, thus inevitably pose unnecessary dependency on action sequences and need a lot of problem-specific designs. We propose a general RL framework that not only exhibits state-of-the-art empirical performance but also generalizes to a variety class of COPs. Specifically, we define state as a solution to a problem instance and action as a perturbation to this solution. We utilize graph neural networks (GNN) to extract latent representations for given problem instances for state-action encoding, and then apply deep Q-learning to obtain a policy that gradually refines the solution by flipping or swapping vertex labels. Experiments are conducted on Maximum k-Cut and Traveling Salesman Problem and performance improvement is achieved against a set of learning-based and heuristic baselines.

1. Introduction

Combinatorial optimization problems (COP) have attracted extensive interest from both machine learning and operation research communities in decades because of its pervasive application scenarios (Grötschel, Jünger, and Reinelt 1991; Plante, Lowe, and Chandrasekaran 1987; Waldspurger, d'Aspremont, and Mallat 2015; Candes et al. 2015). Most of these problems are known to be NP-hard (Karp 1972) and extremely challenging for their combinatorial nature: the optimization is performed over a discrete structure (e.g. a weighted graph) and is often associated with exponentially sized feasible solution space. To give a concrete example, consider Maximum k-Cut: given a weighted graph $G = (V, E)$, find a partition of V that divides G into k disjoint sets which maximize the sum of edge weights among each pair of partitions. Another popular example is Traveling Salesman Problem (TSP), in which the goal is to search for the shortest possible route that visits each node on a graph once and only once and returns to the origin node.

Since the solution domain of COP is prohibitively large, exact methods, such as enumeration based approaches are simply intractable. Therefore, classical solutions for COPs mainly focused on heuristics for decades; such methods include simulated annealing (Patel, Mah, and Karimi 1991) and genetic algorithms (Dedieu et al. 2003). However, these heuristics often suffer from arduous case-specific design and redundant computation because common combinatorial structures have to be addressed repeatedly across similar problem instances. Recently, learning-based methods have emerged as an effective tool to provide generalizability across similar problems by exploiting the structure of the target problem. In particular, reinforcement learning (RL) has been identified as a powerful end-to-end framework (Bello et al. 2016; Ma et al. 2019; Kool, Van Hoof, and Welling 2018; Abe et al. 2019; Barrett et al. 2019), because it does not rely on ground-truth solutions compared to supervised methods (Vinyals, Fortunato, and Jaitly 2015; Li, Chen, and Koltun 2018; Mittal et al. 2019; Nowak et al. 2017; Joshi, Laurent, and Bresson 2019). However, most RL-based attempts are tailored for a single class of COPs and require a lot of domain-specific designs. For instance, Bello et al. (2016) adopt the Pointer Network structure (Vinyals, Fortunato, and Jaitly 2015) to encode a permutation of vertex set for TSP, but it is specifically designed to tackle COPs for which the output depends on the length of the input. Ma et al. (2019) resort to a hierarchical RL structure to address TSP with time-window constraints, but such design is only tailored for TSP and can hardly generalize to other COPs. To alleviate the issue, Khalil et al. (2017) proposed a general framework, S2V-DQN, to cope with a wide range of COPs by combining graph neural network (GNN) and deep Q-learning. However, S2V-DQN's node selection strategy is greedy in nature, which prevents it from yielding high-quality solutions since the learned policy cannot revoke its previous decisions based on the observations afterward. Moreover, this greedy node selection strategy suffers when the target COPs have non-sequential structured solutions, e.g., Maximum k-Cut with $k > 2$, because a sequence of irreversible node selections is not an effective way to form graph partitions.

We propose a new general RL framework for COPs with reversible action design. Similar to S2V-DQN, our method also consists of a GNN encoder for state-action encoding and a deep Q-network for policy learning. However, unlike S2V-DQN and other similar approaches in which the agent

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

builds solutions with irreversible actions, our formulation allows RL agents to explore the whole solution space starting from an arbitrary solution and keep improving it. Another strength of our framework is the ability to handle a wider range of COPs with different solution structures. Our method precludes any ad-hoc designs in the RL formulation and leaves problem-specific properties to the state-action encoding networks. As a result, it can easily transfer to different COPs by applying different state-action encoders while maintaining all other essential designs in the RL module. We tested our framework on two extensively studied COPs, Maximum k-Cut and TSP, on both synthetic and real-world datasets and obtained competitive performance compared to both learning-based methods and heuristic baselines. In addition to the encouraging performance, we also observed that the trained agent has learned to approach the optimal solution with the ability to jump out of the local minima along the way, which is the key bottleneck of greedy or heuristic based solutions.

2. Related Works

The recent success of applying machine learning to solve COPs can be traced back to Vinyals, Fortunato, and Jaitly (2015), where pointer networks are used to solve TSP via supervised learning. Then Bello et al. (2016) further developed an RL solution for TSP based on pointer networks using policy gradient. After that, a number of follow-up works emerged to focus on various COPs with different challenges. For example, Kool, Van Hoof, and Welling (2018) applied the Transformer architecture as the state encoder and reported state-of-the-art results on vehicle routing problems. Ma et al. (2019) resorted to a hierarchical RL structure to address TSP with time-window constraints. And Abe et al. (2019) trained a GCN with Monte-Carlo tree search to enhance its generalizability on five NP-hard COPs. Besides these attempts, S2V-DQN (Khalil et al. 2017), to the best of our knowledge, is the first general RL-based solution for COPs. It combines a graph neural network (GNN) and deep Q-network and is demonstrated to be effective across three classes of COPs (i.e. Minimum Vertex Cover, TSP, and Maximum 2-Cut).

All the aforementioned RL-based methods employ a greedy node selection strategy to construct solutions, i.e., classifying one unlabeled node at a time until all nodes are labeled. Instead of constructing the solution in an incremental manner, Barrett et al. (2019) proposed ECO-DQN for Max Cut, an exploratory method that allows the RL agent to continuously improve any given solution via local search (Johnson, Papadimitriou, and Yannakakis 1988), by modeling the actions as vertex-flipping. This distinctive reversible action design enables the agent to access any valid solution during planning. The reported empirical improvement from this reversible action design suggests its advantage over its irreversible counterpart. However, ECO-DQN involves several ad-hoc designs in its modeling pipeline (such as reward shaping and input feature engineering), which makes it less applicable to other COPs.

3. Method

Problem Formulation

In this work, we focus on the COPs where the problem instances impose a weighted graph structure. With a slight abuse of terminology, we use COP to denote combinatorial optimization problems on graphs. In this case, we can specify the essential components of a COP as follows:

1. A problem instance, i.e. a weighted graph $G = (V,E,w)$ sampled from some underlying distribution $\mathcal{D}$, where $V = (v_1, \cdots, v_n)$, $E = (e_1, \cdots, e_m)$ and $w = \{w(v_i, v_j)\}$ represent the vertex set, the edge set and the weights associated with each edge, respectively. For simplicity, we denote $w_{ij} = w(v_i, v_j)$ and simply assign $w_{ij} = 0$, if v_i and v_j are not directly connected. In this case, G can be fully expressed by the weighted adjacency matrix w.
2. A solution to a problem instance G given by a mapping l from V to $\mathbb{N}$, which assigns an integer-valued label to each vertex. For simplicity, we denote the solution as $L = (l(v_1), \cdots, l(v_n))$.
3. An objective function $O(G, L) \in \mathbb{R}$ that takes a problem instance G and a solution L as input and outputs the evaluation on (G, L). The goal of a COP is to find a global optimal solution $L^*_G = \text{argmin}_L O(G, L)$ under certain constraints to L.

Since the solution space on a finite graph G is also finite, its global optimum L^*_G always exists if the problem is feasible. But in most cases, as the solution space of l is formidably large, trivial enumeration-based solutions are not feasible.

Given the abstract formulation above, we can instantiate some popular COPs by specifying different G, L and O. For example, consider Maximum k-Cut, where G is an arbitrary weighted graph, L is an element from $\mathcal{L} = \{(l_1, \cdots, l_n) | l_i \in \{1, 2, \cdots, k\}\}$, and $O = \sum_{c=1}^{k} \sum_{(i,j) \in \{i,j | l_i = l_j = c\}} w_{ij}$. Similarly we can formulate TSP as: G is a complete graph characterized by a weighted adjacency matrix w, L is an element in the set of all permutations of $\{1, 2, \cdots, n\}$, and $O = w_{l_1 l_n} + \sum_{k=1}^{n-1} w_{l_k l_{k+1}}$. In this paper, we illustrate our solution framework using Maximum k-Cut and TSP, because they represent two typical classes of COPs in a broad sense. TSP represents COPs with sequentially structured solutions, such as vehicle routing problem (Dantzig and Ramser 1959); and Maximum k-Cut represents COPs that fall into the category of node classification, such as Minimum Vertex Cover and Maximal Independent Set. By designing meta-algorithms for solving Maximum k-Cut and TSP, we demonstrate how our methodology can shed light on finding solutions for a wide range of related COPs.

Reinforcement Learning Solution Design

We apply reinforcement learning to design a meta-algorithm that minimizes the discrete objective function $O(G, L)$. Unlike the greedy strategy employed by S2V-DQN (Khalil et al. 2017), in which the agent is trained to construct the solution sequentially by adding nodes to a partial solution, our design allows the agent to gradually improve a given solution L by

adding small perturbations to it at each step. Specifically, we define the primitive components in our RL formulation as follows:

1. State space. The state space $\mathcal{S} = \{s = (G, L) | G \in \mathcal{D}, L \in \mathcal{L}_G\}$ (L may depend on G) is defined as the set of all the valid graph-solution pairs. Note that the distribution $\mathcal{D}$ might contain graphs with different sizes.
2. Action space. The action space $\mathcal{A}(s)$ is designed as all valid perturbations on L given a state $s = (G, L)$, plus a dummy action $\emptyset$ which terminates further actions. $\mathcal{A}(s)$ includes those perturbations such that the perturbed L is still a valid solution in $\mathcal{L}$ for G. In particular, we define two types of perturbations on L:
   (a) Flipping $\mathcal{A}^{flip}$, in which an action a flips the label l_i for node v_i into a different one.
   (b) Swapping $\mathcal{A}^{swap}$, in which an action a swaps the label l_i and l_j for node pair (v_i, v_j).
3. Reward. The reward function $r(s, a)$ is defined as the negative value change for the objective function O after taking action a at state $s = (G, L)$, i.e., $r(s, a) = O(G, L) - O(G, a \circ L)$.
4. Policy. We adopt the Q-learning framework to fit a Q-function $Q(s, a)$ that evaluates the accumulated reward given any state-action pair. According to the learned q-value, we apply a deterministic policy $\pi(s) = \arg\max_{a \in \mathcal{A}(s)} Q(s, a)$.
5. Termination. An episode is terminated when the agent decides to take the dummy action $\emptyset$, or the maximum number of actions M have been executed, where M is a hyper-parameter and grows with the graph size n.

Apart from the difference in action design, another distinction between our work and S2V-DQN lies in the construction of states: in S2V-DQN, the state is a graph with a partial solution; while in our framework the state is a graph with a complete solution. As a result, our solution can always exploit the complete structure of a solution at each step, while S2V-DQN struggles with incomplete information to take the next action, especially in the first few steps when the partial solution only consists of a few scattered nodes. Another advantage of our framework is its flexibility when dealing with inherent or additional constraints, e.g., the permutation constraint in TSP (i.e. no duplicated nodes on a route), or the cut-size constraints in Maximum k-Cut (i.e. given size of each subgraph induced by the cut). To address these constraints, we only need to pose restrictions on the action space to guarantee that each perturbed solution also satisfies the constraints. But in S2V-DQN and other works, such as (Bello et al. 2016) which also adopts S2V-DQN's greedy design pattern, ad-hoc changes to the architecture of the Q-network have to be made to handle the constraints.

State/Action Representation and the Q-function

To apply Q-learning for solving COPs, we need to obtain a continuous representation of any state-action pair (s, a). This representation should incorporate both the combinatorial nature of graph G and the structure of a solution L. Considering these factors, we adopt the message passing neural networks (MPNN) architecture (Gilmer et al. 2017), which is a general graph neural network framework to obtain node representations by collecting information iteratively from local graph structures. In detail, we first initialize an embedding vector $\mu_v^0 = \mathbf{0} \in \mathbb{R}^d$ for each node v; then at round k, $\{\mu_v^k\}$ is updated by leveraging information from its neighbors by:

$$\mu_v^{k+1} = \text{relu}\Big(\theta_0 x_v + \theta_1 \frac{1}{|N(v)|} \sum_{u \in N(v)} w_{uv} \mu_u^k + \theta_2 \frac{1}{|N(v)|} \sum_{u \in N(v)} \text{relu}(\theta_3 w_{uv})\Big), \tag{1}$$

where $\theta_0, \theta_2 \in \mathbb{R}^{d \times p}, \theta_1 \in \mathbb{R}^{d \times d}$ and $\theta_3 \in \mathbb{R}^p$ are model parameters, $N(v)$ is the set of node v's neighbors and $|N(v)|$ is its cardinality. In the first term, $x_v \in \mathbb{R}^p$ is the static feature vector for vertex v that incorporates additional node information from a problem instance. For instance, in Maximum k-Cut, x_v can take the form of a one-hot label vector of v to encode the current cut information; and in TSP, x_v can be the initial coordinates of v if given. The second term in Eq (1) aims to aggregate information from v's neighbors by taking a weighted average over the neighbors' embeddings proportional to the edge-weights. This encodes neighborhood information together with their edge weight defined relatedness. And the third term is served to emphasize if v is closely connected to its neighbors based on the given edge weights.

The final node embeddings $\{\mu_v^T\}$ are obtained by iterating this procedure for T rounds, such that they are expected to carry T-hop information among the nodes based on the graph topology. The state and action representations can thus be constructed based on the computed node embeddings $\{\mu_v^T\}$. For different COPs, we need different readout functions to compute the graph embedding from node embeddings. Take Maximum k-Cut and TSP as two examples to illustrate this process.

• **Maximum k-Cut**, suppose the current graph cut given by state $s = (G, L)$ is $(V_1, \cdots, V_k)$, where each V_i corresponds to a cluster of nodes. We first define the cluster representation by averaging all the node vectors within the cluster as $H_c(i) = \frac{1}{|V_i|} \sum_{v \in V_i} \mu_v^T$. For the action embedding, we represent a flipping action (i.e., to flip u's label from i to j) as the concatenation $H_a^{flip} = [\mu_u^T; H_c(j)]$, where μ_u^T is the representation of vertex u, which is expected to carry information about vertex u and cluster i, and $H_c(j)$ is the representation of the target cluster j. Similarly, we can also define the representation of the swapping action (i.e., to swap the labels of u and v) as $H_a = [\mu_u^T; \mu_v^T; H_c(l(u)); H_c(l(v))]$, where $l(u)$ and $l(v)$ correspond to u and v's current labels. Finally, we use attention to construct the state representation over cluster embeddings. The attention weight is computed by taking cluster embeddings as the reference and the action embedding as the query: $w(i) = \text{softmax}_i(H_c^\top(i) W_a H_a)$, where $H_c(i) \in \mathbb{R}^d, H_a \in \mathbb{R}^{4d}$, and $W_a \in \mathbb{R}^{d \times 4d}$ is a trainable parameter matrix. Then the state representation is formed by $H_s = \sum_{i=1}^k w(i) H_c(i)$.

• **TSP**, its solution exhibits a sequential structure instead of a clustered one. Therefore, we adopt an RNN encoder to

get the state embedding: $H_s = RNN(\{\mu_i^T\}_{i=1}^n; \Theta^{RNN})$, where $\{\mu_i^T\}_{i=1}^n$ is the sequential input of RNN defined by the node permutation in solution L, Θ^{RNN} is the weights of RNN, and the output H_s is taken as the RNN hidden state at step n. Because of the constraint in TSP, i.e., no repeated nodes on a tour, flipping action does not apply. To specify the swapping action, we define the sequential-swap $(i, j), i < j$, for TSP as to swap the sub-sequence $(L_i, L_{i+1}, \cdots, L_j)$ in L to $(L_j, L_{j-1}, \cdots, L_i)$. Because a sequential-swap action (i, j) only changes edge-weights associated with $v_{i-1}, v_i, v_j, v_{j+1}$ in the TSP tour (i.e., change edges from $v_{i-1,i}, v_{j,j+1}$ to $v_{i-1,j}, v_{i,j+1}$), we represent it by $H_a = [\mu_i^T; \mu_j^T; \mu_{i-1}^T; \mu_{j+1}^T]$, where $1 \leq i < j \leq n, \mu_0^T = \mu_n^T, \mu_{n+1}^T = \mu_1^T$.

Based on the state and action representations, we establish the parameterized Q-function as $Q(s, a) = W_0 \cdot \text{relu}([W_1 H_s; W_2 H_a])$, where $W_0 \in \mathbb{R}^{1\times 2d}, W_1 \in \mathbb{R}^{d\times d_s}, W_2 \in \mathbb{R}^{d\times d_a}$ are trainable parameters, d_s and d_a are the dimensions of state and action embeddings respectively. We use swap action design in Maximum k-Cut as an example in Algorithm 1 to illustrate the procedure of applying our proposed RL solution framework for COPs.

Algorithm 1: Q-learning for swap search

Input: Training epochs E, episode maximum length M, exploration constant ε.
Initialization: Set experience replay buffer $\mathcal{M} = \emptyset$.
for $epoch \leftarrow 1$ to E **do**
  Sample a graph $G(V, w)$ from $\mathcal{D}$.
  Initialize a random solution L_0.
  for $t \leftarrow 0$ to T **do**
    with probability ε, select a random node pair $a_t = (v_i, v_j)$; othereise, $a_t = \text{argmax}_{(i,j)} Q(s_t, (v_i, v_j); \Theta)$.
    Swap the i-th and j-th elements in L_t to give L_{t+1}, assign $s_{t+1} = (G, L_{t+1})$.
    Execute the evaluation function $O(G, L_{t+1})$ to observe reward r_t.
    if $t \geq N$ **then**
      Add tuple $(s_{t-N}, a_{t-N}, r_{t-N,t}, s_t)$ to $\mathcal{M}$.
      Sample a batch $\mathcal{B}$ from $\mathcal{M}$.
      Update Θ for $\mathcal{B}$ via gradient descent.
Output: Model parameter Θ.

End-to-End Training

We apply N-step off-policy TD method (Sutton and Barto 2018), i.e., N-step Q-learning to train $Q(s, a)$, which has been demonstrated to be effective when dealing with delayed rewards. By evaluating the value function N-step ahead, we encourage the agent to be less myopic by avoiding eagerly punishing an action which induces a negative immediate reward. At each training step, we first sample a random batch of state-action-reward tuples $\mathcal{B} = \{(s_1^0, \{a_1^j\}_{j=1}^N, \{r_1^j\}_{j=1}^N, s_1^N), \cdots, (s_b^0, \{a_b^j\}_{j=1}^N, \{r_b^j\}_{j=1}^N, s_b^N)\}$ from the replay buffer $\mathcal{M}$, where the index $j = 1, \cdots, N$ denotes the observations at the j-th step from the corresponding state s^0. The q-loss is given by $\sum_{i=1}^b \left(y_i - Q(s_i, a_i; \Theta)\right)^2$, where the target y_i is computed as the accumulated reward in N steps starting from s_i^0 and plus the estimated long-term rewards starting from s_i^N:

$$y_i = \sum_{j=1}^{N} \gamma^{j-1} r_i^j + \gamma^N \arg\max_a Q(s_i^N, a; \hat{\Theta}), \qquad (2)$$

where $\gamma \in (0, 1)$ is the discounting factor and $\hat{\Theta}$ is the currently estimated parameter in the target Q-net.

The main challenge for Algorithm 1 lies in the evaluation of optimal action in planning: computing the maximum of $Q(s, a)$ over all actions in Eq (2) could be extremely expensive, especially when we adopt the swapping action space $\mathcal{A}^{swap}$ design, which is of size $O(n^2)$ and n is the number of nodes. In this case, the time and space complexity for a single forward operation is $O(n^2 d^2)$ and $O(n^2 d)$ respectively, where d is the hidden dimension of state representation. Inspired by (Van de Wiele et al. 2020; Dulac-Arnold et al. 2015), we introduce an auxiliary action-proposal network to perform action elimination. The idea is that after the state embedding H_s is obtained for the current state s, we directly propose a pseudo action $\tilde{a}(s)$ in $\mathbb{R}^d$ from the auxiliary network: $\tilde{a}(s) = AP(H_s; \Theta_{ap})$.

Although $\tilde{a}(s)$ might not directly map to a real action, $\tilde{a}(s)$ is expected to lie around the optimal action in state s. Therefore, we can generate a distribution $\pi^{prop}(a)$ over the whole action space by leveraging the similarity between $\tilde{a}(s)$ and each node's embedding μ_v^T. For example, for swapping actions, we set $\pi^{prop}(v_i, v_j) \propto \exp\left(s(v_i) + s(v_j) + \theta_0 s(v_i) s(v_j)\right)$, where $s(v) = \tilde{a}(s)^\top \mu_v^T$ and $\theta_0 \in \mathbb{R}$ is a trainable parameter. To perform action elimination, we first choose an action reserve ratio $\epsilon \in (0, 1]$, then draw a subset $\mathcal{A}_{prop}^{swap}$ of size $\epsilon|\mathcal{A}^{swap}| \propto O(n)$ from $\mathcal{A}^{swap}$ according to $\pi^{prop}(a)$, and restrict the search space of actions in Eq (2) to $\mathcal{A}_{prop}^{swap}$. By deploying the auxiliary action proposal network, we reduce the time and space complexity for a single forward operation to $O(n^2 + nd^2)$ and $O(n^2 + nd)$. In practice, we observe that the hidden dimension d should always increase proportionally to n in order to guarantee good empirical performance. And this action elimination technique is essential to make the RL framework applicable to large-scale COPs.

To train the action proposal network, we introduce a regularized loss function:

$$\mathcal{L}(\Theta_{ap}; s) = -\log \pi^{prop}(a^*(s)|\Theta_{ap}) - \lambda H(\pi^{prop}(a|\Theta_{ap})), \qquad (3)$$

where $a^*(s) = \arg\max_{a \in \mathcal{A}_{prop}^{swap}} Q(s, a; \hat{\Theta})$ is the action selected by the Q-net among the proposed action set $\mathcal{A}_{prop}^{swap}$. The first term in Eq (3) is to minimize the negative log-likelihood for the selected action, which makes the action with the highest Q-value more likely to be proposed. The second term is the negative entropy of the proposal distribution, and minimizing it would encourage uncertainty in π^{prop} throughout training and prevents it from collapsing to a deterministic distribution.

4. Experiments

Problem Setups We evaluate our proposed solution framework on Maximum k-Cut and TSP. For Maximum k-Cut, we consider two settings: Maximum Cut (i.e., k=2) and Maximum k-Cut with size constraint (i.e., $k > 2$ and the sizes of the graph cut are given). For the training, validation and test set of Maximum Cut, we use complete graphs whose weighted adjacency matrices are given by the pairwise Euclidean distances among n nodes uniformly sampled from $[0, 1]^h$. For Maximum k-Cut, we use a synthetic dataset generated by k-clustered graph, where each problem instance $G = (V, E, w)$ is a complete graph generated by first sampling k centroids $\{c_1, \cdots, c_k\}$ uniformly from $[0, 1]^h$, and then sample m nodes $\{x_{ij}\}_{j=1}^m$ for each centroid c_i from Gaussian distribution $N(c_i, \sigma_i^2 I_h)$, where m, h, σ_i are hyper-parameters and I_h is the identity matrix of size h. The weighted adjacency matrix w is then computed as the Euclidean distances between each pair of nodes, i.e., $w_{ij} = ||x_i - x_j||_2$. For TSP, we draw training graphs from a distribution where each node x_i is uniformly sampled from a 2-D square $\{(x, y)|0 \leq x \leq 1, 0 \leq y \leq 1\}$ and the weighted adjacency matrix w is computed by Euclidean distances accordingly, i.e., $w_{ij} = ||x_i - x_j||_2$. For the testing set of TSP, we resort to a public benchmark TSPLIB derived from real-world instances (Reinelt 1991).

Note that in our settings all training and validation graphs are complete graphs, which is computationally cumbersome when n becomes large. To scale up for larger n (e.g., $n > 50$), we use the K-nearest neighbor graph ($K = 50$) to replace the complete graph, i.e., reserve top-K nearest neighbors in the weighted adjacency matrix. We used two types of synthetic graphs and a real-world dataset from TSPLIB (Reinelt 1991) in the experiment. For more detail of our configuration and the hyper-parameter setting, please refer to our full arxiv version https://arxiv.org/pdf/2102.07210.pdf.

Metrics and baselines For Maximum Cut, we consider two RL-based baselines S2V-DQN (Khalil et al. 2017) and ECO-DQN (Barrett et al. 2019). In addition, we consider several popular heuristic-based methods including semidefinite programming (SDP) (Goemans and Williamson 1995), genetic programming (GP) (Kim, Kim, and Moon 2001), and greedy algorithm. SDP is a non-greedy algorithm that gives a solution by relaxing the discrete problem to a continuous one and then applying semidefinite programming to address the resulting optimization problem. GP is an evolutionary computation method that imitates biological evolution by iteratively improving a set of solutions through mutations and selections. The greedy algorithm we use here is an iterative method that starts with a random cut and flips or swaps the label of a vertex at each step with the greatest immediate increase in the cut value until no further improvement can be made, which corresponds to a RL agent in our framework that always follows the one-step reward.

For Maximum k-Cut ($k > 2$) with size constraint, S2V-DQN and ECO-DQN cannot directly apply. As a result, we use three heuristic baselines, i.e., SDP, GP, greedy algorithm for benchmarking. Note that the greedy algorithm we use here takes swapping as an action to guarantee the size constraints.

For the baselines of TSP, we include S2V-DQN and two approximation algorithms Farthest insertion (Farthest) and 2-opt, which are reported as the two best-performing approximation methods on TSPLIB dataset (Khalil et al. 2017). The implementation details for these two approximation algorithms can be found in (Applegate et al. 2006). Note that the 2-opt method is essentially the greedy algorithm that takes the swapping action as we designed for TSP.

We use the approximation ratio $R^{app} = O^*/O^{opt}$ as a metric to evaluate the quality of solutions, where O^* is the best objective value given by an algorithm, and O^{opt} is the true optimal objective value. For Maximum k-Cut problems with large sizes, we do not have access to O^{opt}. Considering that the greedy algorithm demonstrates itself as the strongest baseline for large-scale Maximum k-Cut problems, we apply it multiple times with different initial solutions and take its best result as the alternative for O^{opt}. For the test problem instances of TSP, O^{opt} is provided in the TSPLIB dataset.

Solution quality comparison To compare the quality of solutions, we train our algorithm with problem instances up to 200 nodes and test it on 100 held-out graphs of the same size. The maximum episode length is set to twice the size of the graph; and during each training epoch, the training batch is sampled uniformly from $\mathcal{D}$ with a randomly initialized solution, with the batch sizes ranging from 50 to 500 to meet the memory limitation posed by a single graphic card. The agent starts planning from a randomly initialized solution on each test instance. Table 3 and 4 summarize the results on the average approximation ratio across Maximum Cut, Maximum k-Cut, and TSP with variance over multiple trials (i.e. different trials are the trajectories starting from randomly drawn initial states) shown on the superscripts and subscripts. Our algorithm is denoted by the name LS-DQN (i.e. Local Search DQN).

For Maximum Cut with $k = 2$, the flipping action design is employed in our solution. As shown in Table 3, LS-DQN performed as good as ECO-DQN and outperformed S2V-DQN on larger test graphs. Compared to approximation heuristics such as SDP and GP, LS-DQN showed a better and more stable performance, and the gain became more significant as the test graph size increases. It is not a surprise to observe the greedy baseline has strong performance, because it can always guarantee a local minimum (i.e., there is no flipping that generates immediate improvement); and in Maximum Cut, local minima were close to the global minimum with very high probability based on our observations. The performance gain of LS-DQN mainly comes from its ability to jump out of those local minima during planning. For Maximum k-Cut, we test our model on three different graph scales: $(k, m) = (5, 6), (10, 10), (10, 20)$. As shown in the bottom of Table 3, LS-DQN performed significantly better than SDP and GP, and slightly better than the greedy algorithm. Table 4 shows the result for TSP. The Farthest and 2-opt algorithms are the two strongest baselines for the TSPLIB dataset according to the results reported in (Khalil et al. 2017). As the graph sizes in TSPLIB are not identical to our training size, we train our RL agent on graphs with

Table 1: Evaluation of generalization of LS-DQN (LS) and comparison with S2V-DQN (S2V). Models are trained on graphs with 50 nodes, and tested on larger sizes up to 300. The average approximation ratio over different testsets is reported.

Test Size	51-100		101-150	
Algorithm	LS	S2V	LS	S2V
MAXCUT	0.984	0.988	0.974	0.971
TSP	1.034	1.075	1.054	1.089
Test Size	151-200		200-300	
Algorithm	LS	S2V	LS	S2V
MAXCUT	0.972	0.975	0.978	0.981
TSP	1.088	1.087	1.094	1.095

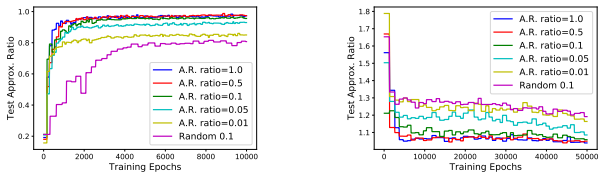
fixed sizes and test it on-the-fly on a batch of graphs whose sizes fall in a certain range, and report the result on the best tour encountered over the training epochs, like it was done in (Khalil et al. 2017). Our LS-DQN reaches the best approximation ratio on graph sizes up to 200 compared to S2V-DQN and the other two heuristics.

Generalization to larger graphs To investigate the generalization ability of our solution on both Maximum Cut and TSP, we train our RL agent on small graphs of a fixed size ($n = 50$) and test it on larger graphs with sizes up to 300. Table 1 summarizes the results with the comparison to S2V-DQN, where the reported values are the approximation ratios averaged over 100 test graphs. As we can see, LS-DQN achieved encouraging and consistent approximation ratios across different test graph sizes on both tasks. LS-DQN generalizes as good as S2V-DQN on Maximum k-Cut, and has shown some advantage on TSP tasks. This confirms the generalization of our proposed RL solution framework and the applicability of learned agents across distinct problem instances.

Trade-off between efficiency and accuracy As we have discussed in the End-to-End Training subsection, the deployment of the action-proposal network (AP-net) is essential to enable our method to scale to large graphs. In the AP-net, the action reserve ratio $\epsilon \in (0, 1]$ controls the proportion of actions being evaluated at each state: a large ϵ allows the agent to evaluate q-values on more state-action pairs, which enhances the quality of planning but also increases the time and space complexity. We investigate the trade-off between efficiency and approximation ratio in our solution. Figure 1 shows the training curve for Maximum k-Cut and TSP under different action reserve ratios. We choose $\epsilon \in \{1.0, 0.5, 0.01, 0.05, 0.01\}$, and a baseline that randomly samples 10% actions without using the AP-net. As we can find, the performance gap is nearly negligible when $\epsilon \geq 0.1$. When $\epsilon \leq 0.05$, the performance drop becomes sensible, but the approximation ratio when reserving only 1% actions proposed by the AP-net is still better than randomly selecting 10% actions. Table 2 lists the relative running time and memory cost when applying different ϵ. It shows that we can maintain 99% performance in terms of the approximation ratio while only consuming approximately 10% time and space for Maximum k-Cut. For TSP, since we adopt an RNN structure in the Q-net, the time and space savings are acceptable but not as significant as in Maximum k-Cut.

Table 2: Trade-off between time/space complexity and the Approx. ratio (A.R.) with different action reserve ratios ϵ in the AP-net. The value shown in the table is scaled according to the result at $\epsilon = 1.0$.

ϵ	k-Cut			TSP		
	A.R.	Time	Space	A.R.	Time	Space
0.5	1.00	0.51	0.53	1.00	0.68	0.51
0.1	0.99	0.11	0.14	0.98	0.29	0.12
0.05	0.95	0.06	0.10	0.93	0.23	0.09
0.01	0.82	0.02	0.06	0.80	0.19	0.06



(a) Maximum k-Cut (b) TSP

Figure 1: Learning curves for Maximum k-Cut and TSP with different action reserve ratios. The test performance is evaluated every 200 epochs in Maximum k-Cut and every 100 epochs in TSP.

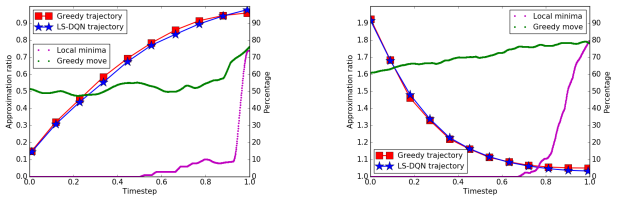
5. Visualization

To better understand the RL agent's behavior in solving COPs, we visualize the results obtained from our model.

Trajectory Statistics We compare the intra-episode behavior of our proposed method, LS-DQN, with the greedy algorithm equipped with the same action space to illustrate how our LS-DQN agent provides better solutions. We chose greedy algorithm for comparison due to its strong performance in our empirical results reported in the main paper.

Figure 2 shows the traces of the improvement in terms of the approximation ratio metric during test episodes for Maximum k-Cut and TSP, respectively. Each episode is terminated at the step where the maximum objective value is reached before hitting the maximum episode length (i.e., 100) or the dummy action was chosen. The traces for LS-DQN (blue star curve) and greedy algorithm (red square curve) are averaged over 200 test instances of size (k, m)=50 for Maximum k-Cut and n=50 for TSP. We normalize episode length to [0, 1] in order to calculate the average approximation ratio across episodes with different lengths. In addition, at each timestep, we use the green dots to illustrate how frequently a greedy move (the action with the maximum immediate reward) is taken by LS-DQN, and the purple dots to show how often the agent encounters a local-minimum state (the state where no action with positive immediate reward is available).

As we can find, in both problems, the trained agent chose greedy actions at a lower frequency at the early stage and it tended to take more greedy actions later in the episode. It indicates that the trained agent learns to explore the solution space at the beginning so that it can benefit more



(a) Maximum k-Cut (b) TSP

Figure 2: Averaged behavior of LS-DQN over 200 instances of graph size (k, m)=50 for Maximum k-Cut and n=50 for TSP. Timesteps are scaled to the range of [0, 1], where 0 and 1 represent the first state (i.e., a randomly initialized one) and the last state (i.e. where the best solution is obtained within the maximum episode length), respectively. The purple and green dots represent the frequency of LS-DQN stepping into a local minimum and the frequency of taking a greedy move at each time step. The red square and blue star track the improvement of the approximation ratio metric from the greedy algorithm and LS-DQN. The result demonstrates the strength of LS-DQN lies in two aspects: 1. it encourages exploration in the initial stage and exploits greedy move more often at the end; 2. it is able to jump out of the local minimum by taking currently less promising actions.

in the future. We should note that in LS-DQN a state encodes a particular solution to the problem instance. When the state (i.e., the current solution) gets closer to the optimal, the LS-DQN agent takes more greedy actions; and that is when the performance of LS-DQN started to surpass the greedy algorithm. The purple dots suggest that although an LS-DQN agent may run into local-minimum states along the way, however, it manages to jump out and land at a better local minimum. For example, in the later stage more and more states have no action with positive immediate reward. A greedy algorithm typically will terminate by then; but the LS-DQN agent managed to take a series of (currently less promising) actions to keep improving the quality of its obtained solution. In our evaluations, the agent secured a good local-minimum state with a high probability of around 80%. For those cases where an episode ends up in a non-local-minimum state, they are caused by the approximation error of Q-function: the absolute value of the Q estimation shrinks and gets close to zero as the agent approaches the optimal state, therefore the approximation error is more likely to obscure the positive Q-value to a negative one, which results in an early stop.

Visualization of Learned Strategy We illustrate two examples of the search trajectory from a trained LS-DQN agent for Maximum k-Cut and TSP, respectively.

Figure 3 and 4 show the comparison between greedy algorithm and our LS-DQN on a Maximum k-Cut instance of size (k, m)=(5,4). For the ease of presentation, we plot the complementary graph instead of the entire cut and revise the objective value O as the sum of edge-weights within each cluster. The graph cut result is visualized by 5 different colors on vertices and the chosen swapping action at each step is marked by stars. In Figure 3, the greedy algorithm starts from an initial solution with $O = 13.74$ and stops at $O = 8.04$ in five steps, as no further swapping actions can immediately improve the current solution. Clearly this greedy strategy is trapped by the locally stable clusters (e.g., the green and blue cluster in Figure 3 at step 6), which look optimal in the local region but are not necessarily globally optimal. As a contrast to the greedy heuristic, Figure 4 shows how LS-DQN arrives at a better solution from the same initial state. It chose not to take any greedy action in the first 5 steps to avoid the formation of locally stable clusters and then started to exploit greedy actions to refine the cut locally from the 6th step. As a result, it yielded a better solution at the end.

Figure 5 and 6 report the same comparison on TSP of size n=15. The TSP tour is visualized by the dashed lines between vertices, the swapping action is marked by stars on vertices, and the solid lines colored in red and blue suggest the two red lines will be replaced by the two blue lines after the swapping action. Figure 5 shows the greedy algorithm (i.e., 2-opt) that starts from a tour with $O = 8.63$ and stops at $O = 4.16$ in five steps, yielding a locally optimal solution. Interestingly, Figure 6 shows how LS-DQN got a better solution by a different search trajectory. Starting from the same solution as in Figure 5, the LS-DQN agent behaved exactly the same as the greedy heuristic in the first 5 steps and arrives at the same local minimum. However, it managed to take an aggressive move which increases the tour length from $O = 4.16$ to $O = 4.35$. After taking this seemingly bad action, the agent made two consecutive greedy actions to further reduce the tour length from $O = 4.35$ to $O = 3.86$. This example demonstrates LS-DQN's ability to jump out of local minima by making farsighted decisions.

6. Conclusions and Future Work

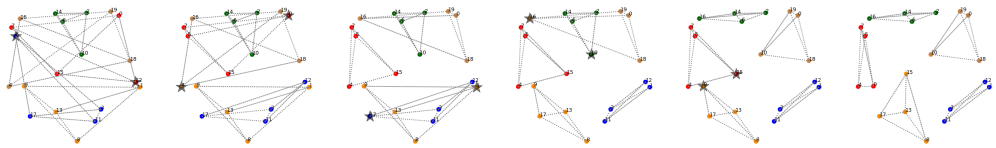
In this paper, we introduced a general end-to-end RL framework for solving combinatorial optimization problems on graphs. The key idea behind our design is to view a solution to a problem instance as state and reversible perturbation to this solution as action. We introduce graph neural networks to extract latent representations of graphs for state-action encoding, and apply deep Q-learning to obtain a policy that gradually improves the solution. We instantiated the meta-algorithm for Maximum k-Cut and TSP, where extensive experiment results demonstrate the solution's competitive performance and generalization across problem instances. The major obstacle that prevents its application to large graphs is the formidable size of swapping action space, which has been successfully addressed by introducing an auxiliary action-proposal network. One important direction of this work is the refinement of the state representation, since our current design does not fully consider the combinatorial structure of the solution: we use one-hot labels on vertices to represent a graph cut and a sequence to represent a tour in TSP. Although they work well empirically, to design a network architecture that encodes various nontrivial combinatorial structures remains a challenging yet promising direction.

Table 3: Approximation ratio comparison for Maximum k-Cut ($k = 2$). Larger is better.

Train/Test Size	LS-DQN	S2V-DQN	ECO-DQN	SDP	GP	Greedy
$n = 20$	$0.98^{+0.02}_{-0.02}$	$0.98^{+0.01}_{-0.01}$	$0.99^{+0.01}_{-0.02}$	0.98	$0.96^{+0.03}_{-0.03}$	$0.98^{+0.02}_{-0.02}$
$n = 50$	$0.98^{+0.02}_{-0.02}$	$0.97^{+0.02}_{-0.02}$	$0.98^{+0.02}_{-0.02}$	0.97	$0.96^{+0.03}_{-0.03}$	$0.97^{+0.02}_{-0.02}$
$n = 100$	$0.97^{+0.01}_{-0.01}$	$0.94^{+0.02}_{-0.02}$	$0.97^{+0.02}_{-0.02}$	0.95	$0.92^{+0.03}_{-0.03}$	$0.94^{+0.02}_{-0.02}$
$n = 200$	$0.95^{+0.01}_{-0.01}$	$0.94^{+0.02}_{-0.02}$	$0.96^{+0.01}_{-0.01}$	0.89	$0.85^{+0.02}_{-0.02}$	$0.93^{+0.02}_{-0.02}$
$(k, m) = (5, 6)$	$0.98^{+0.02}_{-0.02}$	-	-	0.92	$0.67^{+0.05}_{-0.05}$	$0.98^{+0.02}_{-0.02}$
$(k, m) = (10, 10)$	$0.97^{+0.02}_{-0.02}$	-	-	0.89	$0.42^{+0.05}_{-0.05}$	$0.96^{+0.02}_{-0.02}$
$(k, m) = (10, 20)$	$0.95^{+0.02}_{-0.02}$	-	-	0.86	$0.34^{+0.04}_{-0.04}$	$0.94^{+0.02}_{-0.02}$

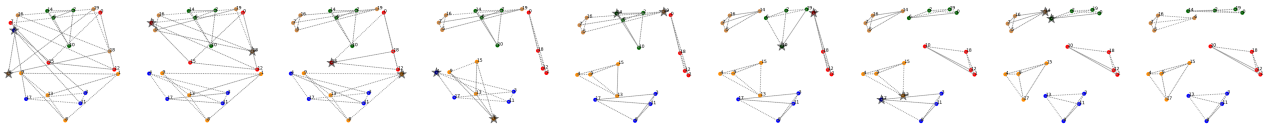
Table 4: Approximation ratio comparison for TSP. Smaller is better.

Train Size	Test Size	LS-DQN	S2V-DQN	Farthest	2-opt
$n = 50$	51-100	$1.04^{+0.01}_{-0.01}$	$1.05^{+0.01}_{-0.01}$	$1.07^{+0.01}_{-0.01}$	$1.07^{+0.01}_{-0.01}$
$n = 100$	101-150	$1.05^{+0.01}_{-0.01}$	$1.05^{+0.01}_{-0.01}$	$1.08^{+0.01}_{-0.01}$	$1.09^{+0.01}_{-0.01}$
$n = 150$	151-200	$1.06^{+0.01}_{-0.01}$	$1.07^{+0.01}_{-0.01}$	$1.08^{+0.01}_{-0.01}$	$1.09^{+0.01}_{-0.01}$



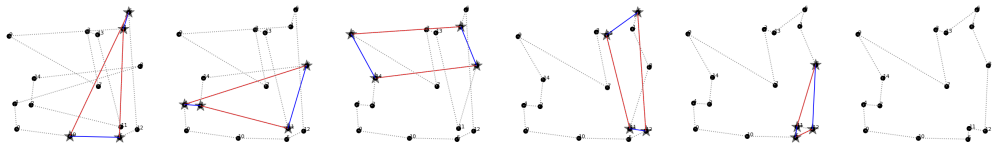
(a) O=13.74 (b) O=12.18 (c) O=10.53 (d) O=8.89 (e) O=8.07 (f) O=8.04

Figure 3: Sample episode of greedy algorithm for Maximum k-Cut



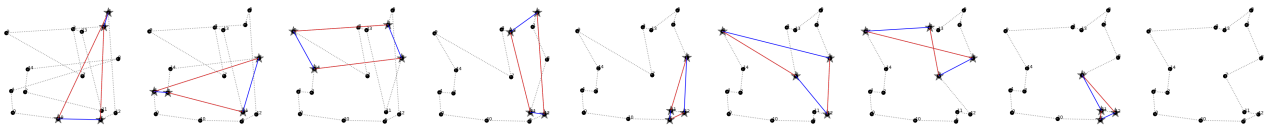
(a) O=13.74 (b) O=12.25 (c) O=10.38 (d) O=9.19 (e) O=8.05 (f) O=7.38 (g) O=6.54 (h) O=6.33 (i) O=6.19

Figure 4: Sample episode of LS-DQN for k-Cut



(a) O=8.63 (b) O=7.34 (c) O=6.31 (d) O=5.36 (e) O=4.22 (f) O=4.16

Figure 5: Sample episode of greedy heuristic for TSP



(a) O=8.63 (b) O=7.34 (c) O=6.31 (d) O=5.36 (e) O=4.22 (f) O=4.16 (g) O=4.35 (h) O=3.88 (i) O=3.86

Figure 6: Sample episode of LS-DQN for TSP

References

Abe, K.; Xu, Z.; Sato, I.; and Sugiyama, M. 2019. Solving NP-Hard Problems on Graphs by Reinforcement Learning without Domain Knowledge. *arXiv preprint arXiv:1905.11623*.

Applegate, D. L.; Bixby, R. E.; Chvatal, V.; and Cook, W. J. 2006. *The traveling salesman problem: a computational study*. Princeton university press.

Barrett, T. D.; Clements, W. R.; Foerster, J. N.; and Lvovsky, A. I. 2019. Exploratory combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1909.04063*.

Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; and Bengio, S. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

Candes, E. J.; Eldar, Y. C.; Strohmer, T.; and Voroninski, V. 2015. Phase retrieval via matrix completion. *SIAM review*, 57(2): 225–251.

Dantzig, G. B.; and Ramser, J. H. 1959. The truck dispatching problem. *Management science*, 6(1): 80–91.

Dedieu, S.; Pibouleau, L.; Azzaro-Pantel, C.; and Domenech, S. 2003. Design and retrofit of multiobjective batch plants via a multicriteria genetic algorithm. *Computers & Chemical Engineering*, 27(12): 1723–1740.

Dulac-Arnold, G.; Evans, R.; van Hasselt, H.; Sunehag, P.; Lillicrap, T.; Hunt, J.; Mann, T.; Weber, T.; Degris, T.; and Coppin, B. 2015. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 1263–1272. JMLR. org.

Goemans, M. X.; and Williamson, D. P. 1995. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6): 1115–1145.

Grötschel, M.; Jünger, M.; and Reinelt, G. 1991. Optimal control of plotting and drilling machines: a case study. *Zeitschrift für Operations Research*, 35(1): 61–84.

Johnson, D. S.; Papadimitriou, C. H.; and Yannakakis, M. 1988. How easy is local search? *Journal of computer and system sciences*, 37(1): 79–100.

Joshi, C. K.; Laurent, T.; and Bresson, X. 2019. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*.

Karp, R. M. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations*, 85–103. Springer.

Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, 6348–6358.

Kim, S.-H.; Kim, Y.-H.; and Moon, B.-R. 2001. A Hybrid Genetic Algorithm for the MAX CUT Problem. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, GECCO'01, 416–423. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 1558607749.

Kool, W.; Van Hoof, H.; and Welling, M. 2018. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*.

Li, Z.; Chen, Q.; and Koltun, V. 2018. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, 539–548.

Ma, Q.; Ge, S.; He, D.; Thaker, D.; and Drori, I. 2019. Combinatorial Optimization by Graph Pointer Networks and Hierarchical Reinforcement Learning. *arXiv preprint arXiv:1911.04936*.

Mittal, A.; Dhawan, A.; Manchanda, S.; Medya, S.; Ranu, S.; and Singh, A. 2019. Learning heuristics over large graphs via deep reinforcement learning. *arXiv preprint arXiv:1903.03332*.

Nowak, A.; Villar, S.; Bandeira, A. S.; and Bruna, J. 2017. A note on learning algorithms for quadratic assignment with graph neural networks. *stat*, 1050: 22.

Patel, A.; Mah, R.; and Karimi, I. 1991. Preliminary design of multiproduct noncontinuous plants using simulated annealing. *Computers & chemical engineering*, 15(7): 451–469.

Plante, R. D.; Lowe, T. J.; and Chandrasekaran, R. 1987. The product matrix traveling salesman problem: an application and solution heuristic. *Operations Research*, 35(5): 772–783.

Reinelt, G. 1991. TSPLIB—A Traveling Salesman Problem Library. *INFORMS Journal on Computing*, 3(4): 376–384.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Van de Wiele, T.; Warde-Farley, D.; Mnih, A.; and Mnih, V. 2020. Q-Learning in enormous action spaces via amortized approximate maximization. *arXiv preprint arXiv:2001.08116*.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer networks. In *Advances in neural information processing systems*, 2692–2700.

Waldspurger, I.; d'Aspremont, A.; and Mallat, S. 2015. Phase recovery, maxcut and complex semidefinite programming. *Mathematical Programming*, 149(1-2): 47–81.