Learning Combinatorial Optimization Algorithms over Graphs

Hanjun Dai*, Elias B. Khalil*, Yuyu Zhang, Bistra Dilkina, Le Song
College of Computing, Georgia Institute of Technology
hdai,elias.khalil,yzhang,bdilkina,lsong@cc.gatech.edu

Abstract

Many combinatorial optimization problems over graphs are NP-hard, and require significant specialized knowledge and trial-and-error to design good heuristics or approximation algorithms. Can we automate this challenging and tedious process, and learn the algorithms instead? In many real world applications, it is typically the case that the same type of optimization problem is solved again and again on a regular basis, maintaining the same problem structure but differing in the data. This provides an opportunity for learning heuristic algorithms which can exploit the structure of such recurring problems.

In this paper, we propose a unique combination of reinforcement learning and graph embedding to address this challenge. The learned greedy policy behaves like a meta-algorithm which incrementally constructs a solution, and the action is determined by the output of a graph embedding network capturing the current state of the solution. We show that our framework can be applied to a diverse range of optimization problems over graphs, and provide evidence that our learning approach can compete with or outperform specialized heuristics or approximation algorithms for the Minimum Vertex Cover, Maximum Cut and Traveling Salesman Problems.

Contents

*Both authors contributed equally to the paper.

1 Introduction

Combinatorial optimization problems over graphs arising from numerous application domains, such as transportation, communications and scheduling, are NP-hard, and have thus attracted considerable interest from the theory and algorithm design communities over the years. In fact, of Karp's 21 problems in the seminal paper on reducibility (Karp, 1972), 10 are decision versions of graph optimization problems, while most of the other 11 problems can be naturally formulated on graphs. The traditional approaches to tackling an NP-Hard graph optimization problem have three main flavors: exact algorithms, approximation algorithms and heuristics. Exact algorithms are based on enumeration or branch-and-bound with an integer programming formulation, but are generally prohibitive for large-scale instances. On the other hand, polynomial-time approximation algorithms are desirable, but may suffer from weak worst-case guarantees or empirical performance, or may not even exist for inapproximable problems. Heuristics are often fast, effective algorithms that lack theoretical guarantees, and also require substantial, problem-specific research and trial-and-error on the part of algorithm designers.

All three paradigms seldom exploit a common trait of real-world optimization problems: instances of the same type of problem are solved again and again on a regular basis, maintaining the same combinatorial structure, but differing mainly in their data. That is, in many applications, values of the coefficients in the objective function or constraints can be thought of as being sampled from the same underlying distribution. For instance, consider the following settings:

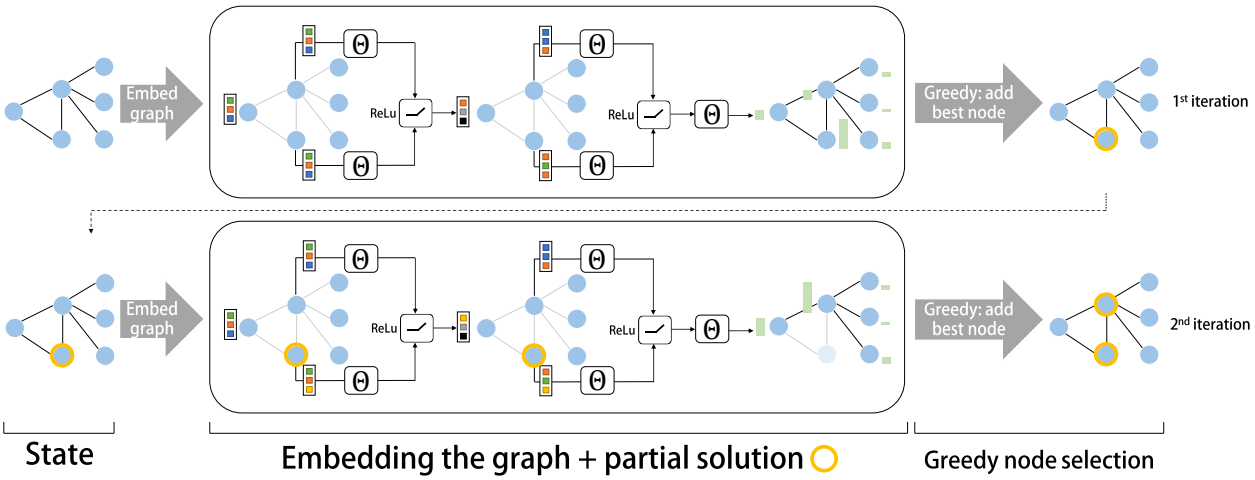


Figure 1: Illustration of the proposed framework as applied to a toy instance of the Minimum Vertex Cover problem. The middle part illustrates two iterations of the graph embedding framework, which results in node scores (the green bars).

– Minimum Vertex Cover: an advertiser on a social network targets a limited set of users with ads hoping they can spread ads to their neighbors; such covering instances needs to be solved repeatedly, since the influence pattern between neighbors may be different each time.

– Maximum Cut: in user clustering analysis, users can be connected as a graph with edge weights representing the dissimilarity between them. A partition of the nodes with maximum cut weight corresponds to two clusters that are maximally dissimilar; such maximum cut instances needs to be solved repeatedly, since the weights may have changed due to new user behavior.

– Traveling Salesman Problem: a package delivery company routes trucks on a daily basis in a given city; thousands of similar optimizations need to be solved, since the underlying demand locations can differ.

Despite the inherent similarity between problem instances arising in the same domain, classical algorithms have not systematically exploited this fact, which motivates the main problem addressed in this paper:

> **Problem Statement:** Given a graph optimization problem G and a distribution $\mathbb{D}$ of problem instances, can we learn better heuristics that generalize to unseen instances from $\mathbb{D}$?

Recently, there has been some seminal work on using deep architectures to learn heuristics for combinatorial problems, including the Traveling Salesman Problem (TSP) (Vinyals et al., 2015; Bello et al., 2016; Graves et al., 2016). However, the architectures used in these works are generic, not yet effectively reflecting the combinatorial structure of graph problems. As we show later, these architectures often require a huge number of instances in order to learn to generalize to new ones. Furthermore, existing works typically use the policy gradient for training (Bello et al., 2016), a method that is not particularly sample-efficient. Additionally, while both of the methods (Vinyals et al., 2015; Bello et al., 2016) can handle variable-size instances, they require manual, ad-hoc input/output engineering (e.g. padding with zeros) to do so.

In this paper, we address the challenge of learning algorithms for graph optimization problems by designing a unique combination of reinforcement learning and graph embedding. The learned policy behaves like a meta-algorithm which incrementally constructs a solution, and the action is determined by a graph embedding network over the current state of the solution. More specifically, our proposed solution framework is different from previous work in the following aspects:

1. **Algorithm design pattern.** We will adopt a *greedy* meta-algorithm design, whereby a feasible solution is constructed by successive addition of nodes based on the graph structure, and maintained to satisfy the problem's graph constraints. As such, the same high-level design can be seamlessly used for different graph optimization problems, a desirable trait as many combinatorial problems are indeed on graphs. This contrasts with recent approaches (Vinyals et al., 2015; Bello et al., 2016; Graves et al., 2016) that adopt a more generic sequence-to-sequence mapping perspective that does not fully exploit graph structure.

2. **Algorithm representation.** We will use a *graph embedding* network of Dai et al. (2016), called structure2vec (S2V), to represent the policy in the greedy algorithm. This novel deep learning architecture over the instance graph "featurizes" the nodes in the graph, which allows the policy to discriminate the usefulness of each node, and generalizes to problem instances of different sizes. Additionally, the graph embedding features capture the properties of a node in the context of its neighborhood, leading to the addition of better nodes to the partial solutions.

3. **Algorithm training.** We will use fitted Q-learning to learn the policy together with the graph embedding network. The framework is set up in such a way that the policy will aim to optimize the objective function of the original problem instance *directly*. The main advantage of this approach is that it can deal with delayed rewards, which here represent the remaining increase in objective function value obtained by the greedy algorithm, in a data-efficient way; In each step of the greedy algorithm, the graph embeddings are updated according to the partial solution to reflect new knowledge of the benefit of *each node* to the final objective value. In contrast, the policy gradient approach of (Bello et al., 2016) updates the model parameters only once w.r.t. the whole solution (e.g. tour in TSP).

The application of a greedy heuristic learned with our framework is illustrated in Figure 1. To demonstrate the effectiveness of the proposed framework, we apply it to three extensively studied graph optimization problems: Minimum Vertex Cover, Maximum Cut and Traveling Salesman Problem. Experimental results show that our framework, a single meta learning algorithm, efficiently learns effective heuristics for all three problems, competing with or outperforming approximation or heuristic algorithms that are tailor-made for the problems. Furthermore, we show that our learned heuristics preserve their effective performance even when used on graphs much larger than the graphs they were trained on.

2 Optimization Problems over Graphs

In this paper, we will illustrate our framework using four types of optimization problems over weighted graphs, namely, minimum vertex cover, maximum cut and two variants of the traveling salesman problem. More specifically, we will denote a weighted graph by $G(V, E, w)$ where V is the set of nodes, E is the set of edges and $w : E \to \mathbb{R}_{\geq 0}$ is the edge weight function, such that $w(u, v)$ denotes the weight of a particular edge $(u, v) \in E$.

– **Minimum Vertex Cover (MVC):** Given a graph G, find a subset of nodes $S \subseteq V$ such that every edge is covered, i.e. $(u, v) \in E \Leftrightarrow u \in S$ or $v \in S$, and $|S|$ is minimized. For this problem, a solution

with at most twice as many nodes as the optimal value (i.e. a 2-approximation) can be obtained by constructing a solution incrementally: while there are uncovered edges, add both ends u and v of an uncovered edge (u, v) to the partial solution.

– **Maximum Cut (MAXCUT):** Given a graph G, find a subset of nodes $S \subseteq V$ such that the weight of the cut-set, $\sum_{(u,v)\in C} w(u, v)$, is maximized, where cut-set $C \subseteq E$ is the set of edges with one end in S and the other end in $V \setminus S$.

  For Maximum Cut, a solution with cut weight at least half of the optimal value (i.e. a 2-approximation) can be obtained in pseudo-polynomial time by the following algorithm: starting with $S = \emptyset$, add to S or remove from S any node as long as this step increases the cut weight.

– **2D Traveling Salesman Problem (TSP2D) (Applegate et al., 2011):** Given a set of points in 2D space, find a tour of minimum total weight, where the corresponding graph G has the points as nodes and is fully connected with edge weights corresponding to distances between points; a tour is a cycle that visits each node of the graph *exactly* once.

– **Graph Traveling Salesman Problem (GTSP) (Cornuéjols et al., 1985):** Given a graph G, find a cycle of minimum total weight that visits each node of the graph G *at least* once.

  Greedy insertion heuristics are amongst the most common algorithms for the TSP. For the 2-dimensional TSP, such heuristics maintain a partial tour in each iteration (initially on 3 nodes), and greedily insert a node that is not in current partial tour based on some evaluation function (e.g. closest node to current partial tour). Insertion heuristics can also be extended to the GTSP.

3 Overview of Proposed Framework

Our framework will leverage the fact that instances of the same type of problem are generated according to a distribution $\mathbb{D}$, and will learn an algorithm A which generalizes and performs well in new instances from the same distribution. As an example of a distribution over problem instances, consider the Erdos-Renyi model of graph generation. Under that model, any edge may exist with probability p, and so all graphs on n nodes and M edges exist with equal probability $p^M(1-p)^{\binom{n}{2}-M}$. These graphs share some common statistics, such as expected degree and cluster distribution. As such, if we solve the same type of problem on different instances G generated from the Erdos-Renyi model with similar n and p, we should expect that algorithms working well in a collection of training instances may perform similarly in other test instances, owing to the structural properties of the graphs: similar degree distributions, size of the largest connected component, number of triangles or other motifs, etc. Thus the algorithm to be learned needs to be able to capitalize on these statistics over graphs, and learn the similarity between two problem instances, and between two potential moves in the algorithm steps to support its multi-stage decision processes.

In industry, it is also common practice to solve the same operational optimization problem repeatedly on slightly different data. One prominent example of such a process is the "Maritime Inventory Routing Problems" library (Papageorgiou et al., 2014) of large-scale transportation problems solved regularly by major corporations that ship goods in bulk: the cost minimization problem remains the same, but the ports, amounts of goods and final destinations may differ. That is, the graph G in every problem instance is a subgraph sampled from the global marine transportation network, and furthermore, there are variations in other parameters across instances (e.g. amount of goods). Thus similar intuition as for the Erdos-Renyi graph example should also apply here.

To leverage such a distribution of training instances to learn a good algorithm A, our framework will break down the task into three aspects:

1. **Algorithm design pattern:** What is a suitable meta-algorithm that can be naturally adapted to various combinatorial problems of different sizes with little effort?

2. **Algorithm representation:** How do we succinctly capture the combinatorial structure of the problem and intermediate solutions, and parameterize the actions of the meta algorithm?

3. **Algorithm training:** How do we train the parametrized meta-algorithm in a data-efficient manner?

At a high level, our framework is a unique combination of reinforcement learning and deep learning over graph structures. Our specific answers and rationale to the three aspects of learning a good algorithm A are:

1. We will choose greedy algorithm as our meta-algorithm, since this same high-level design pattern can be seamlessly used for different graph optimization problems, a desirable trait as most combinatorial problems are indeed on graphs.

2. We will use a *graph embedding* network of Dai et al. (2016), called structure2vec, to represent the actual algorithm steps. Structure2vec is a novel deep learning architecture over the instance graph which "featurizes" the nodes over the graph in terms of the usefulness of each node. Additionally, the graph embedding features capture the properties of a node in the context of its neighborhood (e.g. statistics such as total degree or total triangle count of a neighborhood, the number of nodes already in a partial solution), and generalizes to problem instances of different sizes.

3. We will use fitted Q-learning to train the parameters in the greedy policy together with those in the graph embedding network. The main advantage of this approach is that it can deal with delayed rewards, which here represent the remaining increase in objective function value obtained by the greedy algorithm, in a data-efficient way.

In the following sections, we will provide more details on our framework, though other specifications are also possible and left for future investigation.

4 Design Pattern: Greedy Algorithm

We will focus on a popular pattern for designing approximation and heuristic algorithms, namely a greedy algorithm. A greedy algorithm will construct a solution by sequentially adding nodes to a partial solution S, based on maximizing some *evaluation function* Q measuring the quality of a node. We will show that despite the diversity of various combinatorial problems, they can be expressed using a common formulation, and then solved via a single greedy meta-algorithm.

4.1 Common formulation

Algorithms which construct a solution by sequentially adding nodes to a partial solution share the following aspects:

- A problem instance G of a particular type of optimization problem is sampled from a distribution $\mathbb{D}$, meaning that the V, E and w of the instance graph G are generated according to a model or real world data.

- A partial solution is represented as an *ordered list* S, which is also a subset of V. Furthermore, we will introduce a vector of binary decision variables x, with each dimension x_v corresponding to a node $v \in V$.[1] $x_v = 1$ if $v \in S$ and 0 otherwise. One can also view x_v as a tag or additional feature on node v.

- A maintenance (or helper) procedure $h(S)$ will be needed, which maps an ordered list S to a combinatorial structure satisfying the specific constraints of a problem.

- The quality of a partial solution S needs to be scored using a potentially nonlinear function $c(h(S), G)$ based on the combinatorial structure h of S.

- The length of the final solution S may vary depending on the instance and the type of the problem. We will need a binary function $t(h(S))$ indicating whether partial solution S is a complete feasible solution for the problem instance.

- An algorithm $A(G, h(\cdot), t(\cdot), c(\cdot))$ will take a problem instance G, the helper function h, the termination criterion t, and the cost function c, and then output a final solution $\widehat{S}$. Thus the process of applying algorithm A to a combinatorial optimization problem can be viewed as

$$\widehat{S} := A(G, h(\cdot), t(\cdot), c(\cdot)), \text{ where } G \overset{iid.}{\sim} \mathbb{D}, \tag{1}$$

  where the *evaluation function* Q will be used repeatedly inside the algorithm A.

Given the above abstract model, various optimization problems can be expressed by using different helper functions, cost functions and termination criteria:

– **MVC:** The helper function does not need to do any work, and $c(h(S), G) = -|S|$. The termination criterion checks whether all edges have been covered.

– **MAXCUT:** The helper function divides V into two sets, S and its complement $\overline{S} = V \setminus S$, and maintains a cut-set $C = \{(u, v) \,|\, (u, v) \in E, u \in S, v \in \overline{S}\}$. Then, the cost $c(h(S), G) = \sum_{(u,v)\in C} w(u, v)$, and the termination criterion does nothing.

– **TSP2D & GTSP:** The helper function will maintain a tour according to the order of the nodes in S. The simplest way is to append nodes to the end of partial tour in the same order as S. Then the cost $c(h(S), G) = -\sum_{i=1}^{|S|-1} w(S(i), S(i+1)) - w(S(|S|), S(1))$, and the termination criterion is activated when all nodes have been included. Empirically, inserting a node u at the position i^* which increases the tour length the least would be a better choice, i.e.

$$i^* = \underset{i=1,\ldots,|S|}{\operatorname{argmin}} w(S(i-1), u) + w(u, S(i)) - w(S(i-1), S(i)),$$

  assuming that $S(0) = S(|S|)$. We will adopt this latter helper function in our formulations of TSP2D and GTSP.

In our formulation, we assume that the distribution $\mathbb{D}$, the helper function h, the termination criterion t and the cost function c are all given beforehand. The quality of the solution $\widehat{S}$ produce by algorithm A will be determined by the *evaluation function* Q, which will be learned later on using a sample of problem instances. This is in contrast to traditional greedy algorithm design, where the *evaluation function* Q is typically hand-crafted, and require substantial, problem-specific research, and much trial-and-error.

[1] our framework can also handle edge decision variables but it will not be the focus of this paper.

Algorithm 1 Greedy Algorithm $A(G, h(\cdot), t(\cdot), c(\cdot))$

Input: An instance $G = (V, E, w)$ from distribution $\mathbb{D}$, helper procedure $h(S)$, termination criterion $t(S)$ and cost function $c(h(S), G)$.
Initialize: partial solution $S \coloneqq ()$, and $\overline{S} \coloneqq V \setminus S$.
repeat
  Set $v^* \coloneqq \text{argmax}_{v \in \overline{S}} Q(h(S), v)$;
  Update partial solution $S \coloneqq (S, v^*)$, and $\overline{S} \coloneqq V \setminus S$;
until termination criterion $t(h(S))$ is true.
Return: Solution S.

4.2 Details of meta-algorithm

Given the common formulation in the last section, we can then use the same generic greedy meta-algorithm to facilitate our learning, and testing on a new problem instance later. More specially, let $S = (v_1, v_2, \ldots, v_{|S|})$, $v_i \in V$ denote the list of nodes constituting the current partial solution, and $\overline{S} = V \setminus S$ the set of candidate nodes for addition, conditional on S. A generic greedy algorithm selects the node v to add next that maximizes an evaluation function, $Q(h(S), v) \in \mathbb{R}$, which depends on the combinatorial structure $h(S)$ of the current partial solution. Then, the partial solution S will be extended as

$$S \coloneqq (S, v^*), \text{ where } v^* \coloneqq \underset{v \in \overline{S}}{\text{argmax}} \; Q(h(S), v), \tag{2}$$

and (S, v^*) denotes the appending of v^* to the end of a list S. The generic steps of this greedy algorithm are illustrated in Algorithm 1.

Overall, the greedy algorithm is very simple, and relies heavily on the evaluation function $Q(h(S), v)$. An evaluation function Q^* is optimal if and only if it results in an optimal solution for any instance drawn from distribution $\mathbb{D}$, i.e. following Q^* as in equation (2) will only add nodes that are in an optimal solution. However, it is unrealistic to assume that such a perfect Q^* is computable in polynomial time, in general, unless $P = NP$. Classical greedy algorithms will use a *hand-crafted* evaluation function $\widehat{Q}(h(S), a)$ that represents the algorithm designer's intuition about what makes for a "good" node given the current partial solution. When the problem becomes complicated, it will be very challenging and tedious to hand-craft such a $\widehat{Q}$. Instead, we will aim to learn a potentially superior evaluation function, $\widehat{Q}(h(S), v; \Theta) \approx Q(h(S), v)$, with parameters Θ, that can approach the performance of a perfect evaluation function.

5 Representation: Graph Embedding

Since we are optimizing over a graph G, we expect that the evaluation function $\widehat{Q}$ should take into account the current partial solution S as it maps to the graph. That is for all the $v \in S$, the corresponding decision variables $x_v = 1$, and they are connected according to the graph structure. Intuitively, $\widehat{Q}$ should summarize the state of such a "tagged: graph G, and figure out the value of a new node to be added in the context of such a graph. Here, both the state of the graph and the context of a node v can be very complex, hard to describe in closed form, and may depend on complicated statistics such as global/local degree distribution, triangle counts, distance to tagged nodes, etc. In order to represent such complex phenomena over combinatorial structures, we will leverage a deep learning architecture over graphs, in particular the structure2vec of Dai et al. (2016), to parameterize $\widehat{Q}(h(S), v; \Theta)$.

5.1 Structure2Vec

In the following, we first provide an introduction to structure2vec of Dai et al. (2016). This graph embedding network will compute a p dimensional feature embedding μ_v for each node $v \in V$ given the current partial solution S. More specifically, structure2vec defines the network architecture recursively according to an input graph structure G, and the computation graph of structure2vec is inspired by graphical model inference algorithms, where node specific tags or features x_v are aggregated recursively according to graph topology G. After a few step of recursion, the network will produce a new embedding for each node which takes into account both graph characteristics and long-range interactions between these node features. One variant of the structure2vec architecture will initialize the embedding $\mu_v^{(0)}$ at each node as 0, and for all $v \in V$ update the embeddings synchronously at each iteration as

$$\mu_v^{(t+1)} := F\left(x_v, \{\mu_u^{(t)}\}_{u\in\mathcal{N}(v)}, \{w(v,u)\}_{u\in\mathcal{N}(v)}; \Theta\right), \tag{3}$$

where $\mathcal{N}(v)$ is the set of neighbors of node v in graph G, and F is a generic nonlinear mapping such as a neural network and kernel function.

Based on the update formula, one can see that the embedding update process is carried out based on the graph topology. A new round of embedding sweeping across the nodes will start only after the embedding update for all nodes from the previous round has finished. It is easy to see that the update also defines a process where the node features x_v are propagated to other nodes via the nonlinear propagation function F. Furthermore, the more iterations one carries out the update, the farther away the node features will propagate to distant nodes and get aggregated nonlinearly at distant nodes. In the end, if one terminates the iteration after T steps, each node embedding $\mu_v^{(T)}$ will contain information about its T-hop neighborhood determined by both graph topology and the involved node features. An illustration of 2 iterations of graph embedding can be found in Figure 1.

5.2 Parameterizing $\widehat{Q}(h(S), v; \Theta)$

We now discuss the parameterization of $\widehat{Q}(h(S), v; \Theta)$ using the embeddings from structure2vec. In particular, we design F to update a p-dimensional embedding μ_v as

$$\text{relu}\big(\theta_1 x_v + \theta_2 \sum_{u\in\mathcal{N}(v)} \mu_u + \theta_3 \sum_{u\in\mathcal{N}(v)} \text{relu}(\theta_4 w(v,u))\big) \tag{4}$$

where $\theta_1 \in \mathbb{R}^p$, $\theta_2, \theta_3 \in \mathbb{R}^{p\times p}$ and $\theta_4 \in \mathbb{R}^p$ are the model parameters, and relu is the rectified linear unit[2] applied elementwise to its input. For simplicity of explanation, x_v here is a binary scalar which is described in previous section. But it is straightforward to extend it to a vector representation, by incorporating useful node information. To make the nonlinear transformations more powerful, we can add some more layers of relu before we pool over the neighboring embeddings μ_u.

Once the embedding for each node is computed, we will use these embeddings to define the $\widehat{Q}(h(S), v; \Theta)$ function. More specifically, we will use the embedding for node v and the pooled embedding over the entire graph as the surrogates for the v and $h(S)$, respectively. That is

$$\widehat{Q}(h(S), v; \Theta) = \theta_5^\top \text{relu}[\theta_6 \textstyle\sum_{u\in V} \mu_u, \theta_7 \mu_v], \tag{5}$$

[2] $\text{relu}(z) = z$ if $z > 0$ and 0 otherwise

Algorithm 2 Computing $\widehat{Q}(h(S), v; \Theta)$

Input: $G = (V, E, w)$, S and v.
Initialize: $x_v = 1$ if $v \in S$ and otherwise $x_v = 0$; $\mu_v^{(0)} = \mathbf{0}$, for all $v \in V$.
for $t = 1$ **to** T **do**
  for $v \in V$ **do**
    Update $\mu_v^{(t)} := \mathrm{relu}\big(\theta_1 x_v + \theta_2 \sum_{u \in \mathcal{N}(v)} \mu_u^{(t-1)} + \theta_3 \sum_{u \in \mathcal{N}(v)} \mathrm{relu}(\theta_4 w(v,u))\big)$.
  end for
end for{fixed point update.}
return $\widehat{Q}(h(S), v; \Theta) := \theta_5^\top \mathrm{relu}[\theta_6 \sum_{u \in V} \mu_u^{(T)}, \theta_7 \mu_v^{(T)}]$.

where $\theta_5 \in \mathbb{R}^{2p}$, $\theta_6, \theta_7 \in \mathbb{R}^{p \times p}$. Since the embedding μ_u is computed based on the parameters from the graph embedding network, $\widehat{Q}(h(S), v)$ will depend on a collection of 7 parameters $\Theta = \{\theta_i\}_{i=1}^7$.

The overall computation for $\widehat{Q}(h(S), v; \Theta)$ is summarized in Algorithm 2. We note that the collection of graph embeddings $\{\mu_v^{(T)}\}_{v \in V}$ can be computed once for each new partial solution S. To query the function value for node v, we need only apply formula (5) using the corresponding $\mu_v^{(T)}$. As such, the number of iterations T for the graph embedding computation is usually small, such as $T = 4$.

The parameters Θ will be learned. Previously, Dai et al. (2016) required a ground truth label for every input graph G in order to train the structure2vec architecture. There, the output of the embedding is linked with a softmax-layer, so that the parameters can by trained end-to-end by minimizing the cross-entropy loss. This approach is not applicable to our case due to the lack of training labels. Instead, we train these parameters together *end-to-end* using reinforcement learning.

6 Training: Q-learning

We show how reinforcement learning is a natural framework for learning the evaluation function $\widehat{Q}$. The definition of the evaluation function $\widehat{Q}$ naturally lends itself to a *reinforcement learning* (RL) formulation (Sutton & Barto, 1998), and we will use $\widehat{Q}$ as a model for the state-value function in RL. We note that we would like to learn a function $\widehat{Q}$ *across a set of* m *graphs from distribution* $\mathbb{D}$, $\mathcal{D} = \{G_i\}_{i=1}^m$, with potentially different sizes. The advantage of the graph embedding parameterization in our previous section is that we can deal with different graph instances and sizes in a unified way.

6.1 Reinforcement learning formulation

More specifically, we will set up the state, action, reward in the reinforcement learning framework as follows:

- *States*: a state S is a sequence of actions (nodes) on a graph G. Since we have already represented nodes in the tagged graph with their embeddings, the state is a vector in p dimensional space, $\sum_{v \in V} \mu_v$. It is easy to see that this embedding representation of the state can be used across different graphs. The terminal state $\widehat{S}$ will depend on the problem at hand;

- *Transition*: Transition is deterministic here, which corresponds to tagging the node $v \in G$ that was selected as the last action with feature $x_v = 1$.

- *Actions*: an action v is a node of G that is not part of the current state S. Similarly, we will represent actions as their corresponding node embedding μ_v, and such definition is applicable across different graphs.
- *Rewards*: the reward function $r(S, v)$ at state S is defined as the change in the cost function after taking action v and transit to a new state $S' := (S, v)$. That is

$$r(S, v) = c(h(S'), G) - c(h(S), G), \tag{6}$$

  and $c(h(()), G) = 0$. As such, the *cumulative reward* R of a terminal state $\widehat{S}$ coincides exactly with the objective function value of the $\widehat{S}$, i.e. $R(\widehat{S}) = \sum_{i=1}^{|\widehat{S}|} r(S_i, v_i)$ is equal to $c(h(\widehat{S}), G)$.
- *Policy*: Based on $\widehat{Q}$, a deterministic greedy policy $\pi(v|S) := \text{argmax}_{v \in \overline{S}} \widehat{Q}(h(S), v)$ will be used. Selecting action v corresponds to adding a node of G to the current partial solution, which results in collecting a reward $r(S, v)$.

Table 1 shows the instantiations of the reinforcement learning framework for the three optimization problems considered herein. We let Q^* denote the *optimal* Q-function for each RL problem. Our graph embedding parameterization $\widehat{Q}(h(S), v; \Theta)$ from last section will then be a function approximation model for it, which will be learned via n-step Q-learning.

Table 1: Definition of reinforcement learning components for each of the three problems considered.

Problem	**State**	**Action**	**Maintenance**	**Reward**	**Termination**
Minimum Vertex Cover	subset of nodes selected so far	add node to subset	None	-1	all edges are covered
Maximum Cut	subset of nodes selected so far	add node to subset	None	change in cut weight	cut weight cannot be improved
Traveling Salesman Problem	partial tour	grow tour by one node	Insertion operation	change in tour cost	tour includes all nodes

6.2 Learning algorithm

In order to perform end-to-end learning of the parameters in $\widehat{Q}(h(S), v; \Theta)$, we use a combination of n-step Q-learning (Sutton & Barto, 1998) and *fitted Q-iteration* (Riedmiller, 2005), as illustrated in Algorithm 3. We use the term *episode* to refer to a complete sequence of node additions starting from an empty solution, and until termination; a *step* within an episode is a single action (node addition).

Standard (1-step) Q-learning updates the function approximator's parameters *at each step* of an episode by performing a gradient step to minimize the squared regression loss:

$$(y - \widehat{Q}(h(S_t), v_t; \Theta))^2, \tag{7}$$

where $y = r(S_t, v_t) + \gamma \max_{v'} \widehat{Q}(h(S_{t+1}), v'; \Theta)$ for a non-terminal state S_t. The n-step Q-learning helps deal with the issue of *delayed rewards*, where the final reward of interest to the agent is only received far in the future during an episode. In our setting, the final objective value of a solution is only revealed after many node additions. As such, the 1-step update may be too myopic. A natural extension of 1-step Q-learning is to wait n steps before updating the approximator's parameters, so as to collect a more accurate estimate of the future rewards. Formally, the update is over the same squared loss (7), but with a different target,

$$y = \sum_{i=0}^{n-1} r(S_{t+i}, v_{t+i}) + \gamma \max_{v'} \widehat{Q}(h(S_{t+n}), v'; \Theta).$$

The fitted Q-iteration approach has been shown to result in faster learning convergence when using a neural network as a function approximator (Riedmiller, 2005; Mnih et al., 2013), a property that also applies in our setting. Instead of updating the Q-function sample-by-sample as in Equation (7), the fitted Q-iteration approach uses *experience replay* to update the function approximator with a batch of samples from a dataset E, rather than the single sample being currently experienced. The dataset E is populated during previous episodes, such that at step $t+n$, the tuple $(S_t, a_t, R_{t,t+n}, S_{t+n})$ is added to E, with $R_{t,t+n} = \sum_{i=0}^{n-1} r(S_{t+i}, a_{t+i})$. Instead of performing a gradient step in the loss of the current sample as in (7), stochastic gradient descent updates are performed on a random sample of tuples drawn from E.

It is known that *off-policy* reinforcement learning algorithms such as Q-learning can be more sample efficient than their policy gradient counterparts (Gu et al., 2016). This is largely due to the fact that policy gradient methods require *on-policy* samples for the new policy obtained after each parameter update of the function approximator.

Algorithm 3 Q-learning for the Greedy Algorithm

Initialize experience replay memory $\mathcal{M}$ to capacity N
for episode $e = 1$ **to** L **do**
  Draw graph G from distribution $\mathbb{D}$
  Initialize the state to the empty sequence $S_1 = ()$
  for step $t = 1$ **to** T **do**
    $v_t = \begin{cases} \text{random node } v \in \overline{S}_t, & \text{w.p. } \epsilon \\ \operatorname{argmax}_{v \in \overline{S}_t} \widehat{Q}(h(S_t), v; \Theta), & \text{w.p. } 1-\epsilon \end{cases}$
    Add v_t to partial solution: $S_{t+1} \coloneqq (S_t, v_t)$
    if $t \geq n$ **then**
      Add tuple $(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t)$ to $\mathcal{M}$
      Sample random batch from $B \overset{iid.}{\sim} \mathcal{M}$
      Update Θ by SGD over (7) for B
    end if
  end for
end for
return Θ

7 Experimental Evaluation

We will compare our method for learning algorithms, Structure2Vec Deep Q-learning (S2V-DQN), to pointer networks trained with actor-critic algorithm (PN-AC), and heuristics designed specially for each types of problem in comprehensive experiments.

7.1 Experiment settings

Instance generation. To evaluate the proposed method against other deep learning approaches and approximation/heuristic algorithms, we generate graph instances for each of the four types of problem. For the Minimum Vertex Cover, Maximum Cut and Graphical TSP problems, we generate two types of graphs: Erdős-Renyi (ER) (Erdos & Rényi, 1960) and Barabasi-Albert (BA) (Albert & Barabási, 2002). For a given range on the number of nodes, e.g. 50-100, we first sample the number of nodes uniformly at random from

that range, then generate a graph according to either ER or BA. For the two-dimensional TSP problem, we use an instance generator from the DIMACS TSP Challenge to generate uniformly random points in the 2-D grid, or clustered points in the 2-D grid. We refer the reader to the Appendix A.1 for complete details on instance generation, including generation of edge weights.

Structure2Vec Deep Q-learning. For our method, we use the graph representations and hyper parameters which are described in Section A.3. Those hyper parameters are selected via preliminary results on small graphs, and then shared for larger ones. Note that for 2D TSP where the graph is essentially fully connected, we simply build the K-nearest neighbor graph to alleviate the computation difficulty for large graph. In the experiment, K is set to be 10. For MVC where we train the model on graphs with up to 500 nodes, we use the model trained on small graphs as initialization for training on larger ones. We refer this trick as 'pre-training', which is denoted in Figure 3.

Pointer Networks With Actor-Critic. We compare our methods against a reinforcement learning based algorithm, as described in Bello et al. (2016). We implement and train their algorithm (named PN-AC thereinafter) for all the tasks. The original model only works on Euclidian 2D TSP problem, where each node is represented by (x, y) coordinates. To handle other graph problems, we describe each node by its adjacency vector instead of a coordinates vector. The original PN-AC is not designed for graphical TSP problem. Thus, to handle different graph sizes, we conduct a singular value decomposition (SVC), and obtain a low-rank approximation for the adjacency matrix. Finally, we use the low-rank embeddings as the inputs of the pointer network.

Greedy Approximation Algorithms. Besides the PN-AC, we also include powerful heuristic methods proposed in literature. Specifically:

- **Minimum Vertex Cover (MVC):** We include *MVCApprox*, which randomly selects an uncovered edge and adds both of its endpoints Papadimitriou & Steiglitz (1982). We also designed a stronger variant, called *MVCApprox-Greedy*, that greedily picks the uncovered edge with maximum sum of degrees of its endpoints. Both of them have a 2-approximation guarantee.

- **Maximum Cut (MAXCUT):** We include *MaxcutApprox*, which maintains the cut set $(S, V \setminus S)$ and randomly selects to move any node from one side to the other side of the cut that results in cut weight improvement Kleinberg & Tardos (2006). To make it stronger, we also greedily move the node that results in the largest incremental gain.

- **2D Traveling Salesman Problem (TSP2D) and Graphical TSP:** For both problems, we include these 2-approximation heuristics (Applegate et al., 2006): Minimum Spanning Tree (MST), Cheapest insertion (Cheapest), Closest insertion (Closest). We also add the Nearest Neighbor (Nearest) heuristic.

Details on Validation and Testing. For PN-AC and our proposed S2V-DQN, we use a CUDA K80-enabled cluster for training and testing. These two algorithms use 100 held-out graphs for validation purposes, and we report the test results on another 1000 graphs. We use CPLEX to get optimal solutions for MVC and MAXCUT, and use Concorde to get optimal solution for the two TSP problems (more details in the appendix). All the numbers reported in the paper are the approximation ratio with respect to this optimal solution or the best solution found by the solvers within 1 hour.

For the MVC, we vary the training and test graph sizes in the ranges {15-20, 40-50, 50-100, 100-200, 400-500}, and further generalize the test graphs of size up to *1200*. For all other three tasks which involve edge weights, due to the limited computation resource, we train up to 200-300 nodes, and generalize to 300-400.

During testing, instead of using Active Search as in Bello et al. (2016), we simply use the greedy policy. This gives us much faster inference, while still being powerful enough.

We modify existing open source code to implement both our algorithm [3] and PN-AC [4]. Our code will also be made available online once published.

7.2 Comparison of solution quality

To evaluate the performance of our learned greedy heuristics against PN-AC and problem-specific heuristic algorithms, for each test instance, we compute the *approximation ratio* of each method relative to the optimal solution, then average the ratio over the set of test instances. The approximation ratio of a solution S to a problem instance G is defined as $\mathcal{R}(S, G) = \max(\frac{OPT(G)}{c(h(S))}, \frac{c(h(S))}{OPT(G)})$, where $c(h(S))$ is the objective value of solution S, and $OPT(G)$ is the optimal value of instance G.

Figure 2 shows the average approximation ratio on different sizes/types of graphs, across different tasks. In all of these figures, lower is better, implying a better approximation ratio. Overall, our proposed method, S2V-DQN, performs significantly better than other methods. In the MVC task, the performance of S2V-DQN is particularly good, as the approximation ratio is roughly 1 and the bar is barely visible.

The PN-AC algorithm performs well in TSP tasks. Since in both TSP settings, the graph is essentially fully connected, graph structure information is not so important. On problems such as MVC, where graph information is more crucial, our algorithm is doing significantly better.

7.3 Convergence of S2V-DQN

In Figure 3, we plot our algorithm's convergence with respect to the held-out validation performance. We first obtain the convergence curve for each type of problem under every graph distribution. To visualize the convergence at the same scale, we plot the approximate ratio.

Figure 3 shows that our algorithm converges nicely on the MVC and MAXCUT problems. For the MVC, we use the model trained on small graphs to initialize the model for training on larger ones. Since our model also generalizes well to problem with different sizes, the curve looks almost flat. Regarding the two TSP problems, where the graph is essentially fully connected, it is harder to learn a good model based on graph structure. Nevertheless, as is shown in previous section, the graph embedding can still learn good feature representations with multiple embedding iterations.

7.4 Generalization

The graph embedding framework enables us to train and test on arbitrary sized graphs. As such, the model itself can be simply applied to larger graphs. How does the performance of the learned algorithm scale with the size of the test graphs?

For MVC, we test the generalization ability of the trained model on graphs with up to 1200 nodes. For MaxCut and the two TSP tasks, we use the graphs with size up to 400 nodes. Table 2 shows the partial results. For the full results, please refer to Appendix A.2. We can see the S2V-DQN achieves a very low approximation ratio. Note that the "optimal" value using which the approximation ratios were computed may not be truly optimal (due to the solver time cutoff at 1 hour), and so the solver's solution does typically get worse as problem size grows. This is why sometimes we can even get better approximation ratio on larger graphs.

[3] https://github.com/Hanjun-Dai/graphnn

[4] https://github.com/devsisters/pointer-network-tensorflow

Table 2: S2V-DQN's generalization results. These test results are produced by S2V-DQN algorithms trained on graphs with 50-100 nodes.

	50-100	100-200	200-300	300-400	400-500	500-600	1k-1.2k
MVC	1.0033	1.0041	1.0045	1.0040	1.0045	1.0048	1.0062
MAXCUT	1.0150	1.0181	1.0202	1.0188	1.0123	1.0177	1.0038
TSP2D	1.0730	1.0895	1.0869	1.0918	1.0944	1.0975	1.1065
GTSP	1.0776	1.0915	1.1016	1.1075	1.1113	1.1158	1.1252

7.5 Scalability

In obtaining a solution given a test graph, our algorithm has polynomial runtime which is $O(VE)$, where V is number of steps (typically the number of nodes) and E is number of edges. For instance, on graphs with 1200 nodes, we can find the solution of MVC within 11 seconds, using a single graphics card, while getting an approximation ratio of 1.0062. For dense graphs, we can also sample the edges for graph embedding computation to save more time, which we will investigate more extensively in future.

Figure 4 shows the scalability of our algorithm, as well as other competitors. The dot size in the scatter plots is linear to the time cost. For simplicity, we use the model trained on graphs with 50 to 100 nodes, and generalize to graphs with different sizes. We can see our algorithm only takes a few seconds to solve the problem size up to 1200. Though being slower than those heuristics, we obtain much better approximation ratio.

7.6 Visualization of solutions

In Figure 5, 6 and 7, we visualize solutions found by our algorithm for MVC, MAXCUT and TSP2D problems, respectively. For the ease of presentation, we only visualize small-size graphs. For MVC and MAXCUT, the graph is of the ER type and has 18 nodes. For TSP2D, we show solutions for a "random" instance (18 points) and a "clustered one" (15 points).

For MVC and MAXCUT, we show two step by step examples where S2V-DQN finds the optimal solution. For MVC, it seems we are picking the node which covers the most edges in the current state. However, we've experimentally shown that our algorithm is doing significantly better than such greedy heuristics. This might imply that S2V-DQN learns a smarter greedy or dynamic programming like strategy. In the example of MAXCUT, it is even more interesting to see that the algorithm did not pick the node which gives the largest intermediate reward at the beginning. Also in the intermediate steps, the agent seldom chooses a node which would cancel out the edges that are already in the cut set. This also shows the effectiveness of graph state representation, which provides useful information to support the agent's node selection decisions. For TSP2D, we visualize an optimal tour and one found by S2V-DQN for two instances. While the tours found by S2V-DQN differ slightly from the optimal solutions visualized, they are of comparable cost and look qualitatively acceptable. The cost of the tour found by S2V-DQN is within 0.07% and 0.5% of optimum, respectively.

8 Related Work

Machine learning for combinatorial optimization. Reinforcement learning is used to solve a job-shop flow scheduling problem in (Zhang & Dietterich, 2000). Boyan and Moore (Boyan & Moore, 2000) use

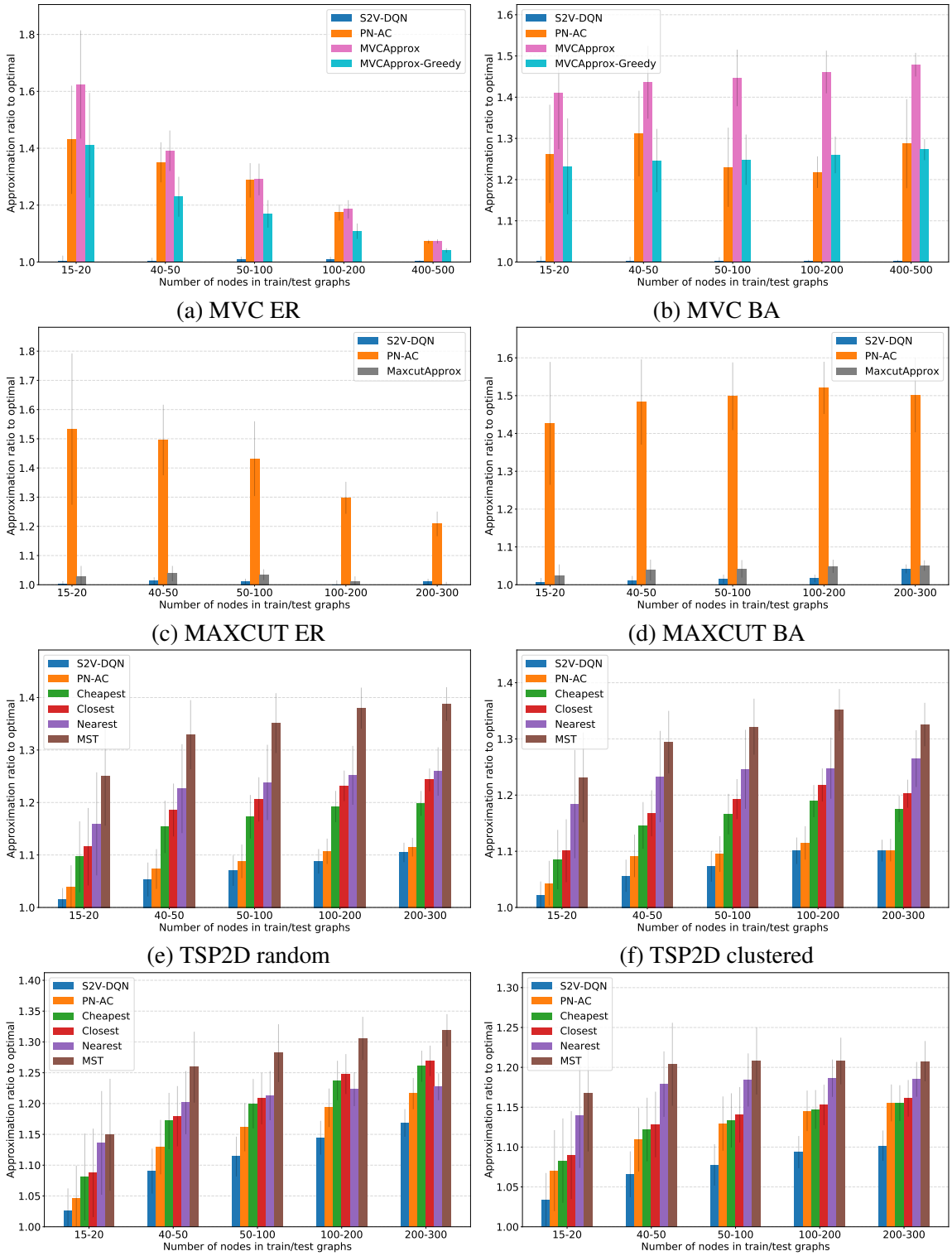
regression to learn good restart rules for local search algorithms. Both of these methods require hand-designed, problem-specific features, a limitation with the learned graph embedding.

Machine learning for branch-and-bound. *Learning to search* in branch-and-bound is another related research thread. This thread includes machine learning methods for branching (Lagoudakis & Littman, 2001; Khalil et al., 2016), tree node selection (He et al., 2014; Sabharwal et al., 2012), and heuristic selection (Samulowitz & Memisevic, 2007). In comparison, our work promotes an even tighter integration of learning and optimization.

Deep learning for continuous optimization. In continuous optimization, methods have been proposed for learning an update rule for gradient descent (Andrychowicz et al., 2016; Li & Malik, 2016) and solving black-box optimization problems (Chen et al., 2016); these are very interesting ideas that highlight the possibilities for better algorithm design through learning.

9 Conclusions

We presented an end-to-end machine learning framework for automatically designing greedy heuristics for hard combinatorial optimization problems on graphs. Central to our approach is the combination of a deep graph embedding approach with a reinforcement learning formulation. Through extensive experimental evaluation, we demonstrate the effectiveness of the proposed framework in learning greedy heuristics that can outperform existing manually-designed greedy approximation algorithms. The excellent performance of the learned heuristics is consistent across multiple different tasks, graph types, and graph sizes.



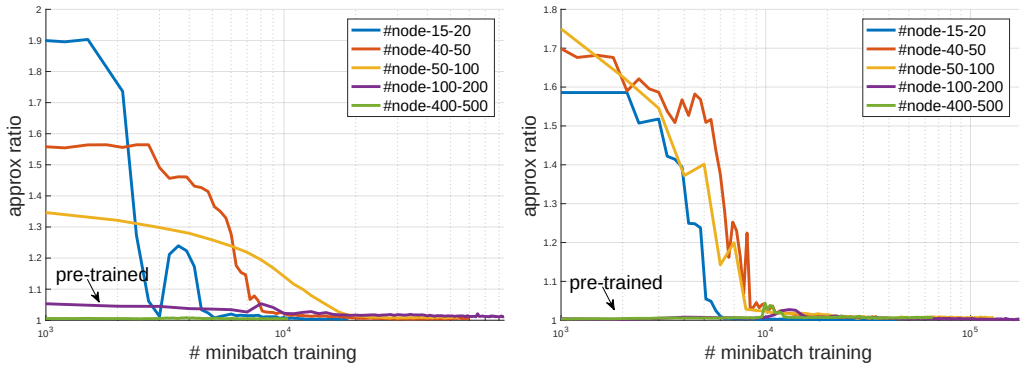
(a) MVC ER (b) MVC BA

(c) MAXCUT ER (d) MAXCUT BA

(e) TSP2D random (f) TSP2D clustered

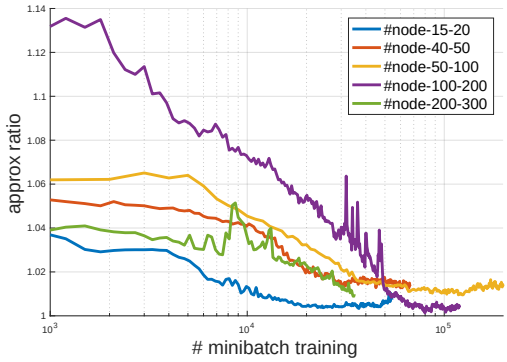
(g) GTSP ER (h) GTSP BA

Figure 2: Approximation ratio on 1000 test graphs. Note that on MVC, our performance is pretty close to optimal. In this figure, training and testing graphs are generated according to the same distribution.

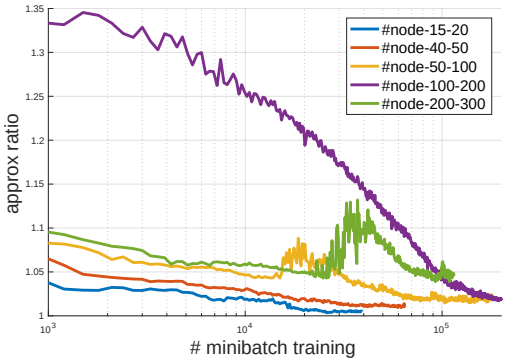


(a) MVC ER

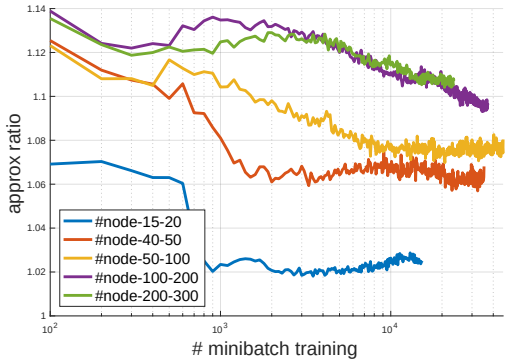
(b) MVC BA



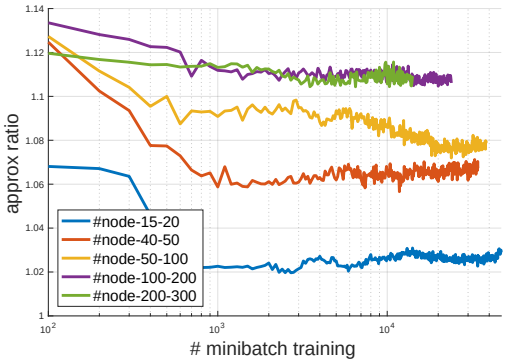
(c) MAXCUT ER



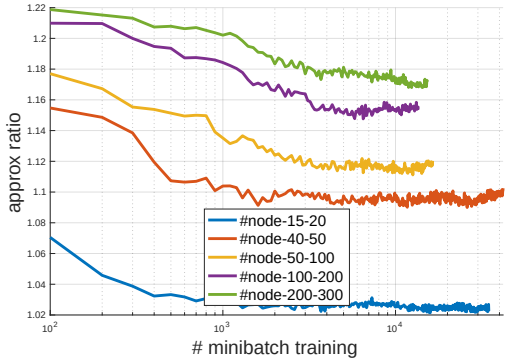
(d) MAXCUT BA



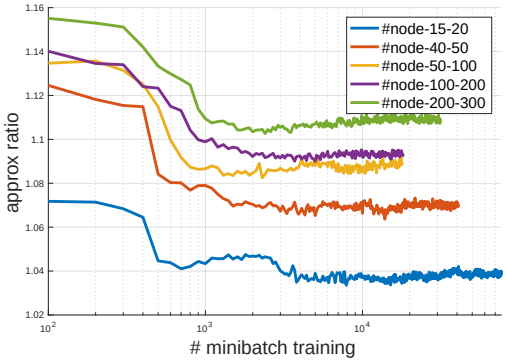
(e) TSP2D random



(f) TSP2D clustered

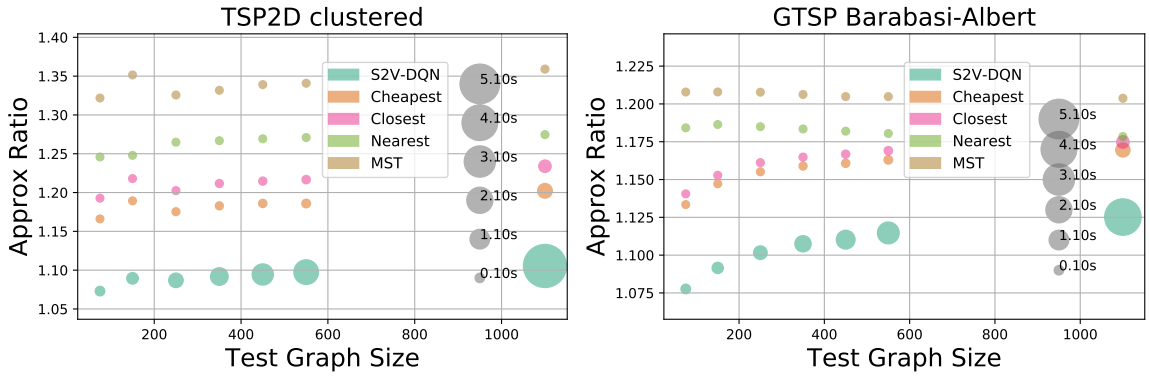


(g) GTSP ER



(h) GTSP BA

Figure 3: S2V-DQN convergence measured by the held-out validation performance.



(a) TSP2D clustered (b) GTSP BA

Figure 4: Scalability test on different tasks. In this figure, the size of a circle is proportional to the average running time of a method over the test instances.

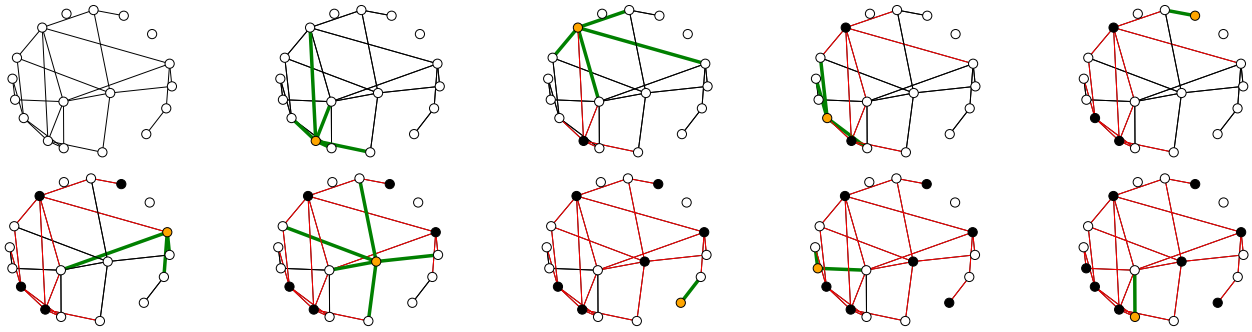


Figure 5: Minimum Vertex Cover: a solution to an instance as it is constructed by our greedy heuristic for an ER graph with 18 nodes. Our solution has cost 9, which is also the optimal value. At each iteration, the selected node is colored in orange, and nodes in the partial solution up to that iteration are colored in black. Newly covered edges are in thick green, previously covered edges are in red, and uncovered edges in black (Best viewed in color).

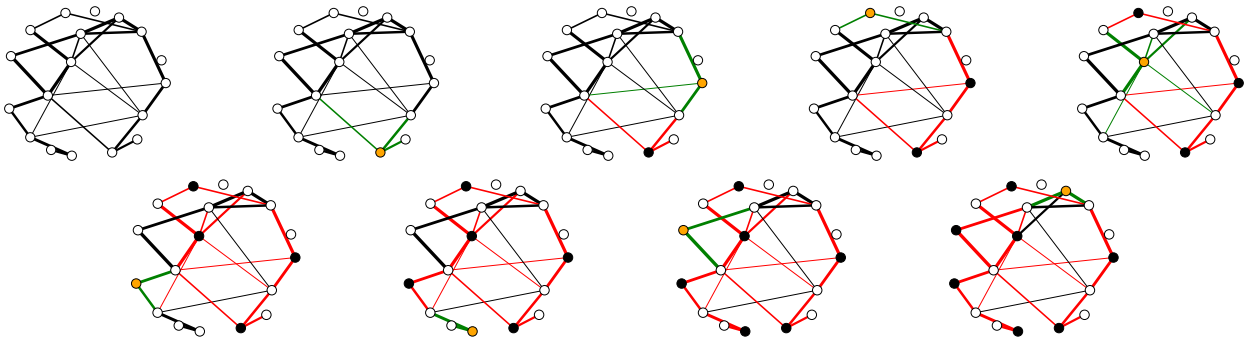


Figure 6: Maximum Cut: a solution to an instance as it is constructed by our greedy heuristic for an ER graph with 18 nodes. Our solution has cost 13.6552, which is also the optimal value. Nodes are partitioned into two sets: white or black nodes. At each iteration, the node selected to join the set of black nodes is highlighted in orange, and the new cut edges it produces are in green. Cut edges from previous iteration are in red (Best viewed in color).

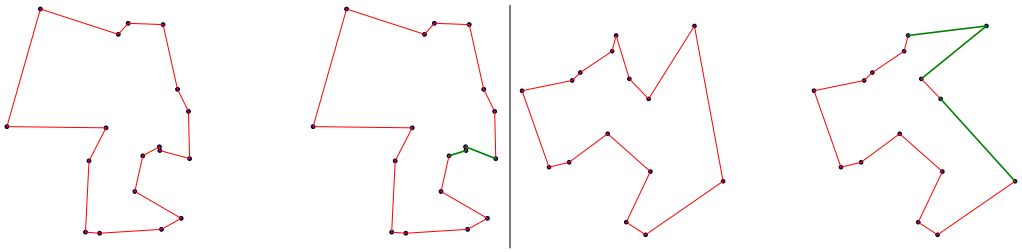


Figure 7: Traveling Salesman Problem (2D). Left: optimal tour to a "random" instance with 18 points (all edges are red), compared to a tour found by our method next to it. For our tour, edges that are not in the optimal tour are shown in green. Our tour is 0.07% longer than an optimal tour. Right: a "clustered" instance with 15 points; same color coding as left figure. Our tour is 0.5% longer than an optimal tour. (Best viewed in color).

References

Albert, Réka and Barabási, Albert-László. Statistical mechanics of complex networks. *Reviews of modern physics*, 74 (1):47, 2002.

Andrychowicz, Marcin, Denil, Misha, Gomez, Sergio, Hoffman, Matthew W, Pfau, David, Schaul, Tom, and de Freitas, Nando. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pp. 3981–3989, 2016.

Applegate, David, Bixby, Robert, Chvatal, Vasek, and Cook, William. Concorde TSP solver, 2006.

Applegate, David L, Bixby, Robert E, Chvatal, Vasek, and Cook, William J. *The traveling salesman problem: a computational study*. Princeton university press, 2011.

Bello, Irwan, Pham, Hieu, Le, Quoc V, Norouzi, Mohammad, and Bengio, Samy. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Boyan, Justin and Moore, Andrew W. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1(Nov):77–112, 2000.

Chen, Yutian, Hoffman, Matthew W, Colmenarejo, Sergio Gomez, Denil, Misha, Lillicrap, Timothy P, and de Freitas, Nando. Learning to learn for global optimization of black box functions. *arXiv preprint arXiv:1611.03824*, 2016.

Cornuéjols, Gérard, Fonlupt, Jean, and Naddef, Denis. The traveling salesman problem on a graph and some related integer polyhedra. *Mathematical programming*, 33(1):1–27, 1985.

Dai, Hanjun, Dai, Bo, and Song, Le. Discriminative embeddings of latent variable models for structured data. In *ICML*, 2016.

Erdos, Paul and Rényi, A. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci*, 5:17–61, 1960.

Graves, Alex, Wayne, Greg, Reynolds, Malcolm, Harley, Tim, Danihelka, Ivo, Grabska-Barwińska, Agnieszka, Colmenarejo, Sergio Gómez, Grefenstette, Edward, Ramalho, Tiago, Agapiou, John, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

Gu, Shixiang, Lillicrap, Timothy, Ghahramani, Zoubin, Turner, Richard E, and Levine, Sergey. Q-prop: Sample-efficient policy gradient with an off-policy critic. *arXiv preprint arXiv:1611.02247*, 2016.

He, He, Daume III, Hal, and Eisner, Jason M. Learning to search in branch and bound algorithms. In *Advances in Neural Information Processing Systems*, pp. 3293–3301, 2014.

Johnson, David S and McGeoch, Lyle A. Experimental analysis of heuristics for the stsp. In *The traveling salesman problem and its variations*, pp. 369–443. Springer, 2007.

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

Khalil, Elias Boutros, Le Bodic, Pierre, Song, Le, Nemhauser, George L, and Dilkina, Bistra N. Learning to branch in mixed integer programming. In *AAAI*, pp. 724–731, 2016.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kleinberg, Jon and Tardos, Eva. *Algorithm design*. Pearson Education India, 2006.

Lagoudakis, Michail G and Littman, Michael L. Learning to select branching rules in the dpll procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344–359, 2001.

Li, Ke and Malik, Jitendra. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin A. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Papadimitriou, C. H. and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, New Jersey, 1982.

Papageorgiou, Dimitri J, Nemhauser, George L, Sokol, Joel, Cheon, Myun-Seok, and Keha, Ahmet B. MIRPLib–a library of maritime inventory routing problem instances: Survey, core model, and benchmark results. *European Journal of Operational Research*, 235(2):350–366, 2014.

Riedmiller, Martin. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pp. 317–328. Springer, 2005.

Sabharwal, Ashish, Samulowitz, Horst, and Reddy, Chandra. Guiding combinatorial optimization with uct. In *CPAIOR*, pp. 356–361. Springer, 2012.

Samulowitz, Horst and Memisevic, Roland. Learning to solve QBF. In *AAAI*, 2007.

Sutton, R.S. and Barto, A.G. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

Vinyals, Oriol, Fortunato, Meire, and Jaitly, Navdeep. Pointer networks. In *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015.

Zhang, Wei and Dietterich, Thomas G. Solving combinatorial optimization tasks by reinforcement learning: A general methodology applied to resource-constrained scheduling. *Journal of Artificial Intelligence Reseach*, 1:1–38, 2000.

Appendix

A Experiment Details

A.1 Problem instance generation

All graphs were generated using the NetworkX [5] package in Python.

A.1.1 Minimum Vertex Cover

For the Minimum Vertex Cover (MVC) problem, we generate random Erdős-Renyi (edge probability 0.15) and Barabasi-Albert (average degree 4) graphs of various sizes, and use the integer programming solver CPLEX 12.6.1 with a time cutoff of 1 hour to compute optimal solutions for the generated instances. When CPLEX fails to find an optimal solution, we report the best one found within the time cutoff as "optimal".

A.1.2 Maximum Cut

For the Maximum Cut (MAXCUT) problem, we use the same graph generation process as in MVC, and augment each edge with a weight drawn uniformly at random from $[0, 1]$. We use a quadratic formulation of MAXCUT with CPLEX 12.6.1. and a time cutoff of 1 hour to compute optimal solutions, and report the best solution found as "optimal".

A.1.3 Traveling Salesman Problem

For the Graphical TSP problem (GTSP) (Cornuéjols et al., 1985), we use the same graph generators as in MVC and MAXCUT, but also guarantee that the generated graphs are connected (i.e. there exists a path between any two nodes of a graph). The connectivity is guaranteed by using the largest connected component of the generated graph as the final graph. Integer edge weights are drawn uniformly at random from $[1, 10000]$, as the solver Concorde can only handle integer distances.

For the (symmetric) 2-dimensional TSP, we use the instance generator of the 8th DIMACS Implementation Challenge [6] (Johnson & McGeoch, 2007) to generate two types of Euclidean instances: "random" instances consist of n points scattered uniformly at random in the $[10^6, 10^6]$ square, while "clustered" instances consist of n points that are clustered into $n/100$ clusters; generator details are described in page 373 of (Johnson & McGeoch, 2007).

To compute optimal TSP solutions for both GTSP and TSP, we use the state-of-the-art solver, Concorde [7] (Applegate et al., 2006), with a time cutoff of 1 hour.

A.2 Full results on generalization

The full generalization results can be found in Table 3, 4, 5, 6, 7, 8, 9 and 10.

[5] https://networkx.github.io/

[6] http://dimacs.rutgers.edu/Challenges/TSP/

[7] http://www.math.uwaterloo.ca/tsp/concorde/

Train \ Test	15-20	40-50	50-100	100-200	400-500	500-600	1000-1200
15-20	1.0032	1.0883	1.0941	1.0710	1.0276	1.0246	1.0111
40-50	\	1.0037	1.0076	1.1013	1.0651	1.0573	1.0299
50-100	\	\	1.0079	1.0304	1.0463	1.0427	1.0238
100-200	\	\	\	1.0102	1.0142	1.0125	1.0103
400-500	\	\	\	\	1.0021	1.0027	1.0057

Table 3: S2V-DQN's generalization on MVC problem in ER graphs

Train \ Test	15-20	40-50	50-100	100-200	400-500	500-600	1000-1200
15-20	1.0016	1.0027	1.0039	1.0066	1.0125	1.0150	1.0491
40-50	\	1.0027	1.0051	1.0092	1.0161	1.0170	1.0228
50-100	\	\	1.0033	1.0041	1.0045	1.0048	1.0062
100-200	\	\	\	1.0016	1.0021	1.0026	1.0060
400-500	\	\	\	\	1.0025	1.0026	1.0030

Table 4: S2V-DQN's generalization on MVC problem in BA graphs

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400
15-20	1.0034	1.0167	1.0407	1.0667	1.1067	1.1489
40-50	\	1.0127	1.0154	1.0089	1.0198	1.0383
50-100	\	\	1.0112	1.0124	1.0546	1.0862
100-200	\	\	\	1.0005	1.0021	1.0211
200-300	\	\	\	\	1.0106	1.0272

Table 5: S2V-DQN's generalization on MAXCUT problem in ER graphs

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400
15-20	1.0055	1.0119	1.0176	1.0276	1.0357	1.0386
40-50	\	1.0107	1.0119	1.0139	1.0144	1.0119
50-100	\	\	1.0150	1.0181	1.0202	1.0188
100-200	\	\	\	1.0166	1.0183	1.0166
200-300	\	\	\	\	1.0420	1.0394

Table 6: S2V-DQN's generalization on MAXCUT problem in BA graphs

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400
15-20	1.0147	1.0511	1.0702	1.0913	1.1022	1.1102
40-50	\	1.0533	1.0701	1.0890	1.0978	1.1051
50-100	\	\	1.0701	1.0871	1.0983	1.1034
100-200	\	\	\	1.0879	1.0980	1.1024
200-300	\	\	\	\	1.1049	1.1090

Table 7: S2V-DQN's generalization on TSP2D problem in random graphs

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400
15-20	1.0214	1.0591	1.0761	1.0958	1.0938	1.0966
40-50	\	1.0564	1.0740	1.0939	1.0904	1.0951
50-100	\	\	1.0730	1.0895	1.0869	1.0918
100-200	\	\	\	1.1009	1.0979	1.1013
200-300	\	\	\	\	1.1012	1.1049

Table 8: S2V-DQN's generalization on TSP2D problem in clustered graphs

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400
15-20	1.0258	1.1296	1.1667	1.2053	1.2265	1.2383
40-50	\	1.0905	1.1181	1.1772	1.2175	1.2356
50-100	\	\	1.1140	1.1523	1.1855	1.2067
100-200	\	\	\	1.1445	1.1692	1.1893
200-300	\	\	\	\	1.1686	1.1857

Table 9: S2V-DQN's generalization on Graphical TSP problem in ER graphs

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400
15-20	1.0340	1.0678	1.0824	1.1038	1.1175	1.1247
40-50	\	1.0659	1.0794	1.0971	1.1096	1.1154
50-100	\	\	1.0776	1.0915	1.1016	1.1075
100-200	\	\	\	1.0937	1.1033	1.1087
200-300	\	\	\	\	1.1018	1.1071

Table 10: S2V-DQN's generalization on Graphical TSP problem in BA graphs

A.3 Experiment Configuration of S2V-DQN

The node/edge representations and hyper parameters used in our experiments is shown in Table 11. For our method, we simply tune the hyper parameter on small graphs (*i.e.*, the graphs with less than 50 nodes), and fix it for larger graphs.

Problem	**node tag**	**edge feature**	**# Embedding**	T	**Batch size**	**n-step**
Minimum Vertex Cover	0/1 tag	N/A	64	5	128	5
Maximum Cut	0/1 tag	edge length; end node tag	64	3	64	1
Graphical TSP	0/1 tag; start/end node	edge length; end node tag	64	4	128	2
2D TSP	coordinates; 0/1 tag; start/end node	edge length; end node tag	64	4	64	1

Table 11: S2V-DQN's configuration used in Experiment.

A.4 Stabilizing the training of S2V-DQN

For the learning rate, we use exponential decay after a certain number of steps, where the decay factor is fixed to 0.95. We also anneal the exploration probability ϵ from 1.0 to 0.05 in a linear way.

We also normalize the intermediate reward by the maximum number of nodes. For Q-learning, it is also important to disentangle the actual Q with obsolete $\tilde{Q}$, as mentioned in Mnih et al. (2015).

A.5 Experiment Configuration of PN-AC

For experiments of PN-AC across all tasks, we follow the configurations provided in Bello et al. (2016): 1) For the input data, we use mini-batches of 128 sequences with 0-paddings to the maximal input length (which is the maximal number of nodes) in the training data. 2) For node representation, we use coordinates for TSP2D, so the input dimension is 2. For MVC, MAXCUT and GTSP, we represent nodes based on the adjacency matrix of the graph. To get a fixed dimension representation for each node, we use SVD to get a low-rank approximation of the adjacency matrix. We set the rank as 8, so that each node in the input sequence is represented by a 8-dimension vector. 3) For the network structure, we use standard single-layer LSTM cells with 128 hidden units for both encoder and decoder parts of the pointer networks. 4) For the optimization method, we train the PN-AC model with the Adam optimizer (Kingma & Ba, 2014) and use an initial learning rate of 10^{-3} that decay every 5000 steps by a factor of 0.96. We initialize all the model parameters uniformly random within $[-0.08, 0.08]$ and clip the $L2$ norm of the gradients to 1.0. 5) For the baseline function in the actor-critic algorithm, we use the exponential moving average performance of the sampled solution from the pointer network as the baseline.