Stanford MS&E 236 / CS 225: Lecture 8
Graph neural networks as greedy heuristics

Ellen Vitercik[*]

April 30, 2024

1 Connecting the dots

We will now connect the dots between Q-learning and the greedy heuristics for NP-hard problems that we learned about in Lecture 5. First, we review several of the NP-hard problem formulations that we saw in that lecture.

Minimum vertex cover (MVC). A vertex cover of a graph $G = (V, E)$ is a set $S \subseteq V$ such that every edge $(i, j) \in E$ is incident to a vertex in S, i.e., $i \in S$, $j \in S$, or both. In the MVC problem, the goal is to find a vertex cover S with minimum cardinality $|S|$.

Maximum cut. A *cut* in a graph is a subset of its vertices $S \subseteq V$. The *weight* of a cut $w(S)$ is the number of edges that cross S to $V \setminus S$ (assuming for simplicity that G is unweighted). In the max-cut problem, the goal is to find a cut with maximum weight.

Traveling salesman problem (TSP). In TSP, the goal is to find a tour—which visits every node exactly once before returning to the starting point—of shortest length.

In that lecture, we saw several greedy algorithms for these problems, which choose a node to add or remove from the solution which maximizes some *score.* For example, the MVC approximation algorithm adds the nodes incident to the edge (i, j) that maximize the *degree sum* $|N(i)| + |N(j)|$. The max-cut greedy algorithm chooses a node to switch to the opposite side of the cut, selecting the node for which that action will lead to the largest improvement of the cut weight. Finally, the TSP heuristic *fathest insertion*, which we covered in Lecture 2, iteratively expands a subtour. At each iteration, it adds the city that is farthest from any city in the subtour. We summarize these scoring rules in Table 1. As we saw in Lecture 5, these hand-designed scoring rules lead to decent approximation algorithms. In the remainder of this lecture, we will see that with GNNs and Q-learning, we can use ML to design more intricate scoring rules which lead to better empirical performance.

[*]These notes are course material and have not undergone formal peer review. Please feel free to send me any typos or comments.

Problem	**Scoring rule guiding the problem's greedy algorithm**
MVC	Degree sum $\lvert N(i)\rvert + \lvert N(j)\rvert$
Max cut	Improvement in cut weight
TSP	Maximum distance from any city in the subtour

Table 1: Scoring rules guiding the greedy algorithms for several NP-hard problems.

Greedy algorithms	**RL**
Partial solution	State
Scoring function	Q-function
Greedily add the node to the partial solution with the highest score	Greedily take the action with the highest Q-function value

Table 2: Correspondence between greedy algorithms and RL.

2 Q-functions as greedy heuristics

As we saw last lecture, the optimal policy π^* of an MDP is greedy with respect to its Q-function Q^*: in state s, it chooses the action $\pi^*(s) = \text{argmax}_{a\in A} Q^*(s,a)$. As we will see, we can draw a close connection between an RL agent acting greedily according to its Q-function and a greedy algorithm building a solution to an NP-hard problem. As Table 2 lays out, we will view the greedy algorithm's partial solution as a *state* of an MDP. The greedy algorithm's scoring function is analogous to the RL agent's Q-function. Just as the greedy algorithm will add the node to its partial solution with the highest score, the RL agent will act greedily according to its Q-function in the vanilla Q-learning algorithm. We illustrate the connection more formally by defining an MDP with the following states, actions, transition probabilities, and rewards:

Input. The input is a graph $G = (V, E)$, together with a computational problem we aim to solve (e.g., MVC, max-cut, TSP, ...).

States. A *state* is a partial solution to the problem. We will denote the partial solution as a set $S_t = (v_1, v_2, \dots, v_{|S_t|})$ with each $v_i \in V$. For example, S_t may be a partial vertex cover, one side of a cut, a partial tour, etc. The initial state is $S_0 = \emptyset$. Later on, it will be useful to represent the state as a vector $\boldsymbol{x}(S_t) \in \{0,1\}^{|V|}$ where the i^{th} component, $\boldsymbol{x}(S_t)_i$, is defined as

$$\boldsymbol{x}(S_t)_i = \begin{cases} 1 & \text{if } i \in S_t \\ 0 & \text{else.} \end{cases}$$

Actions. An action will be to select a vertex $i_t \notin S_t$ to add to the partial solution S_t.

Transitions. The transition to the next state S_{t+1} is deterministic: $S_{t+1} = (v_1, v_2, \dots, v_{|S_t|}, i_t)$.

Reward. We define the reward $r(S_{t+1})$ to be the change in the NP-hard problem's objective when i_t was added to S_t. For example, in max-cut, the optimization objective is the weight of the cut $w(S_{t+1})$, so $r(S_{t+1}) = w(S_t \cup \{i_t\}) - w(S_t) = w(S_{t+1}) - w(S_t)$. By defining the reward to be the change in problem's objective, we have that the cumulative reward over T timesteps is simply the optimization objective evaluated on the final solution S_T. For example, for max-cut, the cumulative reward (without discounting) is

$$\sum_{t=1}^{T} r(S_t) = \sum_{t=1}^{T} w(S_t) - w(S_{t-1}) = w(S_T) - w(S_0) = w(S_T).$$

Thus, by maximizing the reward at each timestep, we maximize the weight of the final cut.

A subtlety in MVC and TSP is that the goal is to *minimize* the size of the vertex cover or length of the tour. In this case, we define $r(S_{t+1})$ to be the *negative* change in the optimization objective when i_t is added to S_t. For example, in MVC, the goal is to minimize the size of the vertex cover, so we define

$$r(S_{t+1}) = -(|S_t \cup \{i_t\}| - |S_t|) = -1. \tag{1}$$

Suppose T is the number of steps until S_T is a vertex cover. Then the cumulative reward (without discounting) is

$$\sum_{t=1}^{T} r(S_t) = \sum_{t=1}^{T} -1 = -|S_T|,$$

so maximizing the cumulative reward amounts to minimizing $|S_T|$, as desired.

Finally, as in approximate Q-learning, the policy will be greedy according to a parameterized estimate of the Q-function $f_{\boldsymbol{\theta}}$. In particular, in each timestep t, we will choose the vertex

$$i_t = \text{argmax}_{i \notin S_t} f_{\boldsymbol{\theta}}(\boldsymbol{x}(S_t), i).$$

To define the neural approximation $f_{\boldsymbol{\theta}}$, we will use a GNN. In particular, we compute node embeddings $\boldsymbol{h}_{1,t}, \dots, \boldsymbol{h}_{|V|,t}$ using a standard GNN with input node features defined by $\boldsymbol{x}(S_t)$ (see the paper by Dai et al. [3] for implementation details). Then, Dai et al. [3] define

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}(S_t), i) = \boldsymbol{w}^\top \text{relu}\left(\left[W_1 \sum_{j \in V} \boldsymbol{h}_{j,t}, W_2 \boldsymbol{h}_{i,t}\right]\right),$$

where $\boldsymbol{\theta}$ represents all trainable parameters in the GNN together with $\boldsymbol{w}$, W_1, and W_2. In other words, given the GNN's node embeddings $f_{\boldsymbol{\theta}}(\boldsymbol{x}(S_t), i)$ is the output of a simple, 1-layer neural network.

This is the basic setup by Dai et al. [3]. For simplicity, we've elided some details. For example, the single-step reward signals can be fairly uninformative, as exemplified by Equation (1). To handle this, Dai et al. [3] use *experience replay* to update the parameters $\boldsymbol{\theta}$ using batches of state-action-reward samples rather than single samples.

3 Results

We now highlight a handful of the results from the paper by Dai et al. [3].

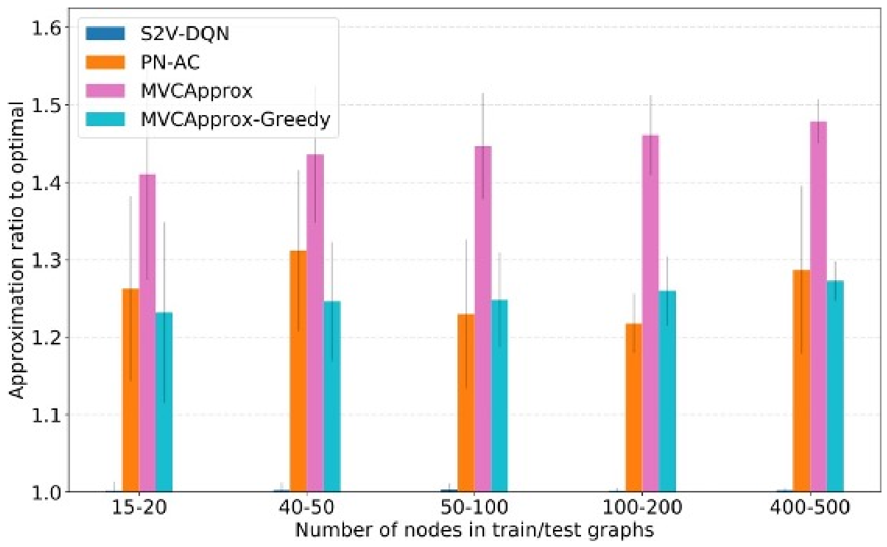

Figure 1: Results for MVC.

3.1 MVC

Figure 1 illustrates the *approximation ratios* of four different methods. If ALG is the size of the vertex cover returned by one of these methods and OPT is the size of the smallest vertex cover, then the approximation ratio is

$$\frac{\text{ALG}}{\text{OPT}}.$$

The training and test graphs are sampled from the Barabasi-Albert random graph family. S2V-DQN (short for "`structure2vec` [2] Deep Q-learning") is the method described in Section 2. PN-AC (short for "Pointer Networks with Actor-Critic") is a deep learning approach based on pointer networks, adapted from the paper by Bello et al. [1], which we covered in Lecture 3 and Lecture 4. MVCApprox-Greedy is the greedy algorithm we covered in Lecture 5, and MVCApprox is a simpler (worse) version of that algorithm. As we can see in Figure 1, S2V-DQN finds vertex covers that are very close to optimal.

3.2 Max-cut

Figure 2 illustrates the approximation ratios of four different methods. The training and test graphs are again sampled from the Barabasi-Albert random graph family. MaxcutApprox is the greedy algorithm we covered in Lecture 5 and SDP is the famous Goemans-Williamson algorithm [4], which provides the best possible worst-case approximation ratio for max-cut, assuming the unique games conjecture is true. The greedy algorithm is surprisingly competitive on these graphs, but S2V-DQN still finds better cuts (with the margin shrinking as the graphs grow).

3.3 TSP

Finally, Figure 3 illustrates the approximation ratios of S2V-DQN, PN-AC, and a wide variety of TSP heuristics. These include, for example, 2-opt, which is the 2-approximation

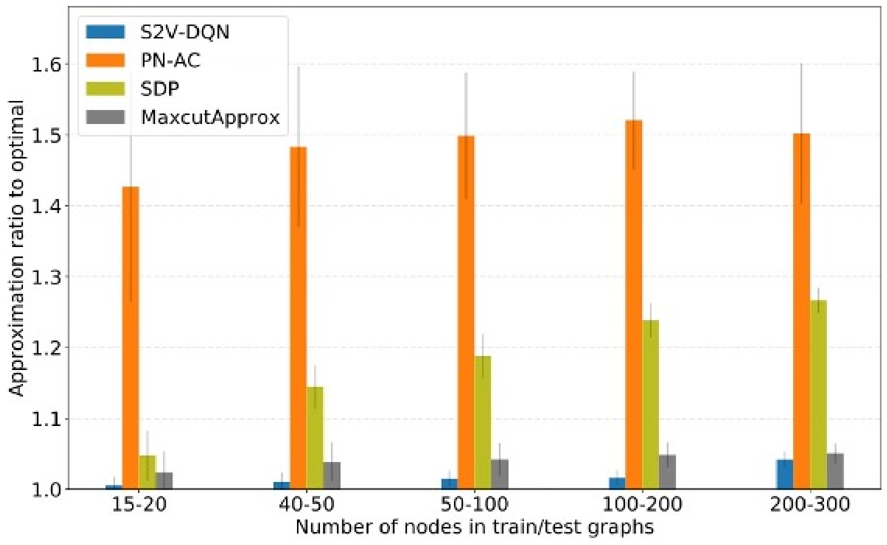

Figure 2: Results for max-cut.

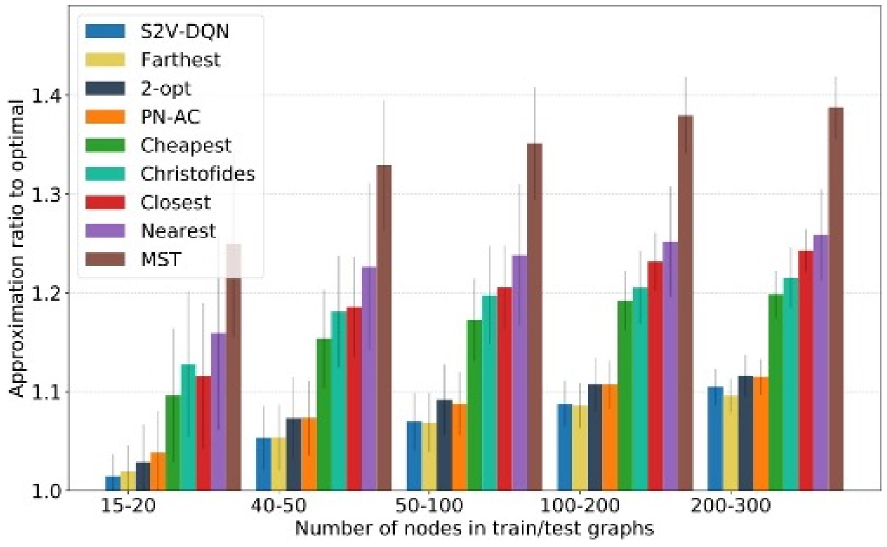

Figure 3: Results for TSP.

algorithm for Euclidean TSP that we covered in Lecture 4. Farthest and Nearest are the the farthest and nearest insertion heuristics that we covered in Lecture 2. It is interesting to note that Farthest is competitive with S2V-DQN, and begins to outperform S2V-DQN as the number of nodes increases.

3.4 Interpretability of the machine-learned heuristics

Dai et al. [3] provide several execution traces of their machine-learned heuristics, comparing the nodes that S2V-DQN adds versus those that existing greedy algorithm add. For example, under MVC, S2V-DQN seems to prioritize nodes such that (1) the node has high degree, but (2) when the node is deleted from the graph, the graph will remain connected, as illustrated by this example. One can make an intuitive argument for why this might make sense: if the complement of a partial vertex cover consists of many disconnected subgraphs, any vertex cover which contains the partial vertex cover will have to include at least one node from each subgraph.

References

[1] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *Workshop track of the International Conference on Learning Representations (ICLR)*, 2017.

[2] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning (ICML)*, 2016.

[3] Hanjun Dai, Elias Boutros Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

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