Graph Neural Networks for Combinatorial Optimization

Mike Van Ness, ORIE 7391

April 27, 2022

Quiz!

Q1: What RL method is used to train the model in the assigned reading paper?

(a) REINFORCE

(b) Deep Q-Learning

(c) Policy Gradient

Q2 True/False: Combinatorial Optimization solvers based on Graph Neural Networks find exact solutions:

(a) True

(b) False

Combinatorial Optimization

Two closely related fields:

- **Integer Linear Programming (ILP)**: Linear programming with additional constraint that variables must be integers.
- **Combinatorial Optimization (CO)**: any optimization problem where the variables are discrete.

Combinatorial Optimization

Two closely related fields:

- **Integer Linear Programming (ILP)**: Linear programming with additional constraint that variables must be integers.
- **Combinatorial Optimization (CO)**: any optimization problem where the variables are discrete.

Examples of CO problems:

- **Traveling Salesman Problem (TSP)**: Find a "tour" of vertices with minimum distance.
- **Minimum Vertex Cover (MVC)**: Find a minimum-size vertex set which "covers" every edge.
- **Max-Cut Problem**: Find the vertex "cut" which has the maximum number of crossing edges.

Solving CO Problems

Ways to solve CO problems:

- Formulate CO problem as (M)ILP, then solve exactly using MILP solver.
  - Branch-and-bound, Branch-and-cut.
- Use exact solver specific to the given CO problem.
  - Concorde TSP Solver.
- Use approximate solver (either general or problem specific) with error guarantees.
  - Christofides' Algorithm for TSP, with 3/2 guaranteed error ratio.
- Use heuristic solver, which may be more effecient or even effective than approximate solvers, but without guarantees.
  - LKH+IPT or EAX for TSP.
  - GNN solvers.
- Likely many many more approaches.

Solving CO Problems

The following are true:

- CO problems often involve (or can be reformulated to involve) graphs.
- CO problems are often large in nature.
- Neural networks are suitable for large-data problems.
- Neural networks are very flexible and can be adapted to a wide range of problems.

Thus, it is natural to use neural networks adapted to graphs, i.e. Graph Neural Networks (GNNs), to work with CO problems.

Solving CO Problems

The following are true:

- CO problems often involve (or can be reformulated to involve) graphs.
- CO problems are often large in nature.
- Neural networks are suitable for large-data problems.
- Neural networks are very flexible and can be adapted to a wide range of problems.

Thus, it is natural to use neural networks adapted to graphs, i.e. Graph Neural Networks (GNNs), to work with CO problems.

This warrants a brief detour to introduce GNNs in general before focusing on their applications in CO.

Graphs

A graph is defined as $G = (V, E, w)$ where

- V is a set of vertices/nodes v.
- E is a set of edges between two vertices (u, v).
- $w : E \to \mathbb{R}^+$ is an edge weight function $w(u, v)$.

Ways to represent a graph:

- Adjacency matrix: $|V| \times |V|$ matrix with entries corresponding to edge weights (or 0).
- Adjacency list: set of $|V|$ lists, one for each vertex, each with list of all neighbors of given vertex.

Deep Learning on Graphs

Deep learning models typically work on vectors/matrices/tensors, so how can get apply these models to graph data?

Deep Learning on Graphs

Deep learning models typically work on vectors/matrices/tensors, so how can get apply these models to graph data?

First thought: use adjacency matrix as input to deep learning model.

Problems:

- Adjacency matrices are often *very* sparse.
- Many adjacency matrices can encode the same graph via permutations, so not representative of true structure.

Deep Learning on Graphs

Core idea behind GNN is *message passing*. The idea is to maintain embeddings for each node that are learned using the graph structure and the given task:

1. For every node u, initialize an embedding for u as $h_u^{(0)} \in \mathbb{R}^d$.
2. Iteratively update embedding using embeddings of graph neighbors:

$$h_u^{(k+1)} = F\left(h_u^{(k)}, \{h_v^{(k)} : v \in \mathcal{N}(u)\}; \theta\right)$$

3. Run 2) for T total iterations/layers, generating embeddings $h_u^{(T)}$.
4. Use $h_u^{(T)}, u \in V$ for specific task (e.g. node classification).
5. Update parameters θ using gradient descent from task (e.g. cross-entropy loss for each node).
6. Repeat steps 2-5.

Message Passing Rules

Here are some common message passing functions:

- Simple GNN:

$$h_u^{(k+1)} = \varphi\left(W_1^{(k)} h_u^{(k)} + W_2^{(k)} \sum_{v \in \mathcal{N}(u)} h_v^{(k)}\right)$$

- Graph Convolutional Network (GCN):

$$h_u^{(k+1)} = \varphi\left(W^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{h_v^{(k)}}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right)$$

- Graph Attention Network (GAT):

$$h_u^{(k+1)} = \varphi\left(W^{(k)} \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} h_v^{(k)}\right)$$

GNNs for CO

How can we utilize GNNs to solve (graph-based) CO problems?

- Approximate solution to CO problem directly using GNNs.
  - Non-autoregressive: solve in one-shot [1].
  - Autoregressive: solve iteratively [2].
- Use GNN to decompose the problem, solve subparts [3].
- Use GNN to edit CO problem, then solve easier problem [4].

We'll focus on [2] for this talk.

S2V-DQN

S2V-DQN [2]: greedy approach to solve TSP, MVC, and Max-Cut using GNNs and Deep RL:

1. Current partial solution $S = (v_1, \ldots, v_{|S|}), \overline{S} = V \setminus S$.
2. Message passing with $x_v = \text{int}(v \in S)$:

$$\mu_v^{(t+1)} \leftarrow \text{relu}\big(\theta_1 x_v + \theta_2 \sum_{u \in \mathcal{N}(v)} \mu_u^{(t)} + \theta_3 \sum_{u \in \mathcal{N}(v)} \text{relu}(\theta_4\, w(v,u))\big)$$

3. Update Q-function:

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

4. Update partial solution:

$$S \leftarrow S \cup \arg\max_{v \in \overline{S}} \widehat{Q}(h(S), v; \Theta)$$

S2V-DQN

Parameters $\Theta = \{\theta_1, \ldots, \theta_7\}$ need to be updated without labels.

Solution: use RL across several problem instances.

- **State:** current partial solution S
- **Action:** adding node $v \in \overline{S}$ to current solution S.
- **Rewards:** difference in cost between S and updated solution $S' = S \cup \{v\}$:

$$r(S, v) = c(h(S'), G) - c(h(S), G)$$

- **Policy:** based on estimated evaluation function $\hat{Q}$, policy

$$\pi(v|S) := \arg\max_{v' \in \overline{S}} \widehat{Q}(h(S), v')$$

S2V-DQN

How does $\hat{Q}$ get updated (Q1)?

- Standard 1-step Q-learning minimizes

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

where $y = \gamma \max_{v'} \widehat{Q}(h(S_{t+1}), v'; \Theta) + r(S_t, v_t)$.

- n-step Q-learning does the same, but with

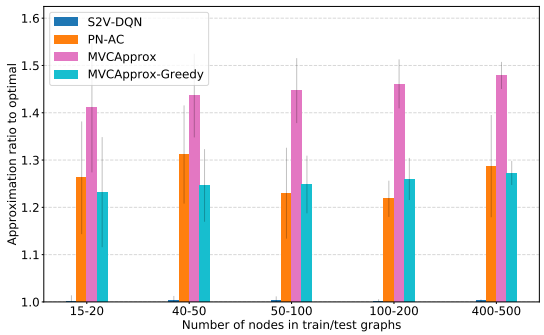
$$y = \sum_{i=0}^{n-1} r(S_{t+i}, v_{t+i}) + \gamma \max_{v'} \widehat{Q}(h(S_{t+n}), v'; \Theta).$$

- Fitted Q-iteration: maintain a dataset E of tuples $(S_t, v_t, R_{t,t+n}, S_{t+n})$ where $R_{t,t+n} = \sum_{i=0}^{n-1} r(S_{t+i}, v_{t+i})$ to sample from when updating Θ

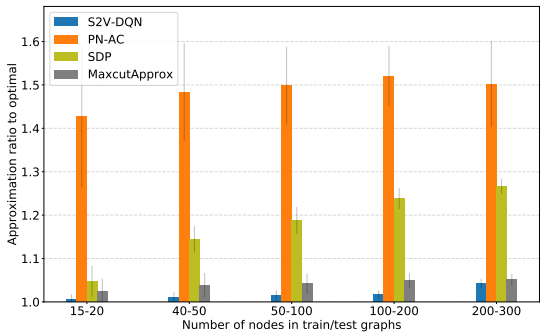
Algorithm Q-learning for the Greedy Algorithm

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 empty $S_1 = ()$
  for step $t = 1$ **to** T **do**
    $$v_t = \begin{cases} \text{random node } v \in \overline{S}_t, & \text{w.p. } \epsilon \\ \arg\max_{v \in \overline{S}_t} \widehat{Q}(h(S_t), v; \Theta), & \text{otherwise} \end{cases}$$
    Add v_t to partial solution: $S_{t+1} := (S_t, v_t)$
    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 for B
    end if
  end for
end for
return Θ

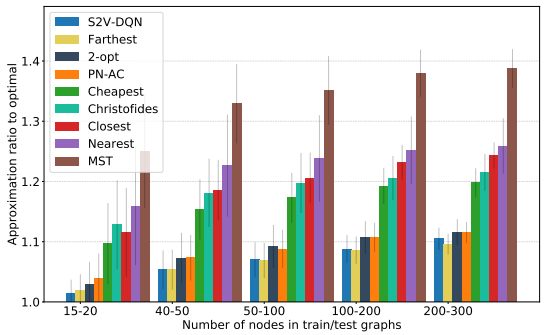
S2V-DQN Results



(a) MVC BA



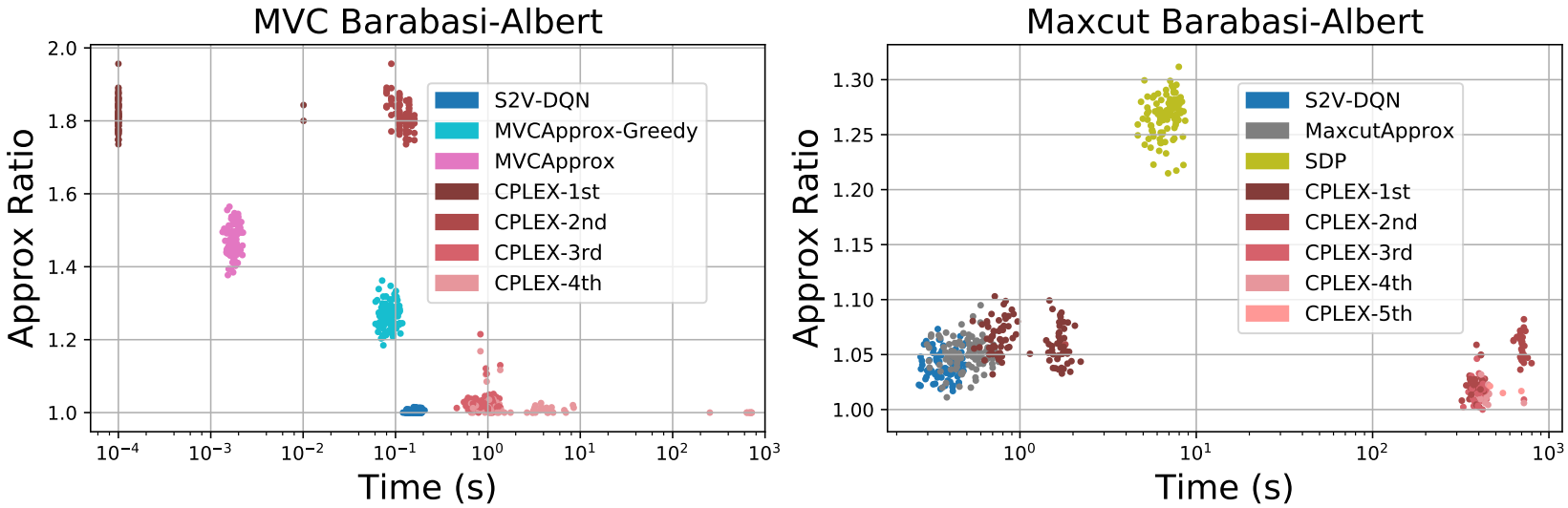
(b) MAXCUT BA



(c) TSP random

(Q2) 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.

S2V-DQN Results



(a) MVC BA 200-300 (b) MAXCUT BA 200-300

Time-approximation trade-off for MVC and MAXCUT. In this figure, each dot represents a solution found for a single problem instance, for 100 instances. For CPLEX, we also record the time and quality of each solution it finds, e.g. CPLEX-1st means the first feasible solution found by CPLEX.

S2V-DQN Results

Table: Realistic data experiments, results summary. Values are average approximation ratios.

Problem	**Dataset**	**S2V-DQN**	**Best Competitor**	**2nd Best Competitor**
MVC	MemeTracker	**1.0021**	1.2220 (MVCApprox-Greedy)	1.4080 (MVCApprox)
MAXCUT	Physics	**1.0223**	1.2825 (MaxcutApprox)	1.8996 (SDP)
TSP	TSPLIB	**1.0475**	1.0800 (Farthest)	1.0947 (2-opt)

Questions?

Thank you!

Sources

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

[2] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song.
Learning combinatorial optimization algorithms over graphs.
Advances in neural information processing systems, 30, 2017.

[3] N. Sonnerat, P. Wang, I. Ktena, S. Bartunov, and V. Nair.
Learning a large neighborhood search algorithm for mixed integer programs.
arXiv preprint arXiv:2107.10201, 2021.

[4] R. Wang, Z. Hua, G. Liu, J. Zhang, J. Yan, F. Qi, S. Yang, J. Zhou, and X. Yang.
A bi-level framework for learning to solve combinatorial optimization on graphs.
Advances in Neural Information Processing Systems, 34, 2021.