Reinforcement Learning for Combinatorial Optimization over Graphs

Alperen Tercan
Colorado State University Computer Science Department, US
alperen.tercan@colostate.edu

1 Summary

Combinatorial optimization problems over graphs arise in many real-world problems from different domains. Most of these problems are NP-hard and real world applications generate million-sized or larger graphs. Traditional approaches to deal with these can be examined in three main headings: Exact algorithms, approximation algorithms and heuristics. As the names suggest, exact algorithms may suffer from very high running times, approximation algorithms may have weak optimality guarantees and performance, and heuristics may require substantial problem-specific research.[6] Using data-driven methods to exploit the repeating nature of underlying structure of the problems is an active research area.

In this project, the use of reinforcement learning for heuristics in graph algorithms is investigated. This line of research can be seen as an extension of general trend in computer science research to replace rule-based, hand-crafted heuristics with data-driven methods. The project focus was identifying possible future research directions rather than a comprehensive survey of the field.

Although the project was done primarily for educational purposes, it contributes to the field with a more mathematically rigorous and RL-oriented analysis. Mainly, we formulate the underlying MDP that papers implicitly assume. This is important for fruitful and mathematically-sound discussions and research for further use of RL in combinatorial optimization problems over graphs. Also, the report provides a unified analysis of two papers using the formulated framework; which proves to be useful for juxtaposition of two papers.

Therefore, this project report is written to provide a formulation of the problem as an RL task, an analysis of the architectures that offered by [6][7], and discuss some potential problem and improvements identified from these papers.

2 Motivation

Combinatorial optimization problems over graphs arise from numerous application domains, such as social networks, transportation, telecommunications and scheduling.Moreover, most of these problems are NP-hard;hence, they do not have efficient exact solutions. Because of their importance and challenging nature, they have always been of interest to many researchers. Efforts to tackle these problems can be summarized under three groups: exact algorithms, approximation algorithms, and heuristics. Exact algorithms are based on enumeration or branch-and-bound with an integer programming formulation. These methods guarantee exact optimal solutions; however, they can be prohibitive for large graphs. Polynomial time algorithms may suffer from weak optimality guarantees, empirical performance, or they can still be too slow for very large graphs. Therefore, fast and empirically effective heuristics are often go-to approach in many problems. However, they may require substantial problem/domain specific research and feature engineering. More importantly, calculation of heuristics like marginal gain can be computationally too expensive.

Inspired from other fields like Computer Vision and NLP, where data-driven methods replace hand-crafted features and heuristics of past; using data-driven methods for combinatorial optimization problems also becomes popular. Since a problem of this kind usually

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

shares the underlying structure across different instances and differ mainly in data; it is likely that data-driven methods that can exploit this trait will be successful. Morevover, complexity of a decision can be reduced to well below the heuristics'; resulting in a faster algorithm. See Section 9.1 for empirical speed-up results.

Due to such potential, following problem will be addressed in this project is:

Problem Statement: Given a combinatorial optimization problem P over graphs drawn from distribution $\mathbb{D}$, learn a heuristic to solve problem P on an unseen graph G generated from $\mathbb{D}$?

To tackle this issue, we will start with a framework that can formulate a diverse set of combinatorial problems.

3 Problem Formulation

[7] provides a general framework for formulating combinatorial optimization problem over graphs in three components:

- Objective function $f(G, h(\mathcal{S}))$
- Termination condition $t(G, h(\mathcal{S}))$ with boolean output
- Helper function $h(G, \mathcal{S})$

Note that each of these are functions of a graph G and solution set $\mathcal{S} \subseteq \mathcal{V}$. Objective function determines the objective value associated with G and $\mathcal{S}$. Termination condition defines when the problem instance is solved. Helper function reduces problems to an iterative node selection problem. For example, helper function for TSP maintains a tour consisting of nodes of $\mathcal{S}$. Also, helper function can be generalized to include some heuristics that designer wants. In [6], helper function for TSP does not simply appends the node to the end of the tour but it inserts it to the optimal position from $|\mathcal{S}|$ positions in the tour.

This formulation is more general than the one in [7].In fact, [7] can be obtained by setting $h(\mathcal{S}) = \mathcal{S}$ and $t(G, h(\mathcal{S})) = (|\mathcal{S}| = b)$, i.e. it is a special case of this formulation.

Having such a general framework to describe problems makes it possible to describe generic solutions that problem agnostic. For instance, one of the most common heuristics for combinatorial optimization problems is Greedy Approach (Algorithm 1). Moreover, for a large family of problems called *submodular maximization*, it is proven to have $1 - \frac{1}{e} \approx 63\%$ approximation ratio.[5] In other words, the solution that is obtained using Greedy Approach is guaranteed to be no worse than 63% of the optimal solution. Below its formulation using the framework above can be seen.

4 MDP Formulation

Markov Decision Processes(MDP) provides a mathematical framework to model decision making problems. Since the formulation in Section 3 reduces combinatorial problems to an iterative node selection problem, we can use MDPs to model them, This would allow us to utilize decades of research to solve MDPs using dynamic programming and reinforcement learning. An MDP for Section 3 will be as follows:

- State s is in the form $(G, \mathcal{S})$ where $G \in \mathbb{D}$, $\mathcal{S} \subseteq \mathbb{D}.\mathcal{S}$, $\mathbb{D}$ is the set of graphs we define our problem over, and $\mathbb{D}.\mathcal{S} = \bigcup_{G' \in \mathbb{D}} \bigcup_{\mathcal{S}' \subseteq G'.\mathcal{V}'} \mathcal{S}'$ and $G'.\mathcal{V}$ is the set of nodes of graph G'. Hence, set of states for MDP $\mathfrak{S} = \mathbb{D} \times \mathbb{D}.\mathcal{S}$ Although the notation gets complicated for the sake of mathematical rigor, intuitively states of MDP are a function of G and $\mathcal{S}$.

Algorithm 1: The Greedy Approach

Input: $G = (V, E)$, an optimization problem P with optimization function $f(.)$,termination condition $t(.)$, helper function $h(.)$
Output: solution set $\mathcal{S}$

```
S ← ∅
i ← 0
while (¬t(G, h(S))) do
    v* ← arg max_{∀v∈V\S} f(S ∪ {v}) − f(S)    /* Choose the v with the largest
     marginal gain */
    S ← S ∪ {v*}, i ← i + 1
end
```

Return: $\mathcal{S}$

- Set of actions $\mathfrak{A} = \bigcup_{G' \in \mathbb{D}} G.\mathcal{V}$
- Transition probability function $\boldsymbol{P}(s_{t+1}|s_t, a_t) = \begin{cases} 1, & \text{if } s_{t+1}.\mathcal{S} = s_t.\mathcal{S} \cup \{a_t\} \wedge \neg t(s_t.G, h(s_t.\mathcal{S})) \\ 0, & \text{otherwise} \end{cases}$
  where $s.G$ is the graph at state s and $s.\mathcal{S}$ is the solution set at state s.
- Reward function $\boldsymbol{R} : \mathfrak{S} \to \mathbb{R}$, $\boldsymbol{R}(s_t, a_t) = f(s_{t+1}.G, h(s_{t+1}.\mathcal{S})) - f(s_t.G, h(s_t.\mathcal{S})) = f(s_t.G, h(s_t.\mathcal{S} \cup \{a_t\})) - f(s_t.G, h(s_t.\mathcal{S}))$

Note that the defined MDP is a function of $\mathbb{D}$, not G. This is because we want to learn to solve any graph in $\mathbb{D}$, not only a fixed graph G. From now on, G_t and $\mathcal{S}_t$ will be used as shorthand for $s_t.G$ and $s_t.\mathcal{S}$.

Now, finding an optimal solution to the original problem on graph G is equivalent to finding the policy $\pi(a_t|s_t)$ that maximizes the reward starting from $s_0 = (G, \emptyset)$. Therefore, we can consider any method to solve the original problem as a policy π.

For example, the Greedy Algorithm in Algorithm 1 can be expressed using following π:

$$\pi(a_t|s_t) = \begin{cases} 1, & \text{if } a_t = \underset{a \in \mathcal{G}_t.\mathcal{V}}{\arg\max}(f(G_t, h(\mathcal{S}_t \cup \{a\}))) - f(G_t, h(\mathcal{S}_t)) \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Now we will talk about how we can learn an optimal policy π instead of hand-crafting approximation algorithms or heuristics like the Greedy Approach.

5 Reinforcement Learning to Find the Optimal Policy

Reinforcement learning provides us with the appropriate framework to find an optimal policy for an MDP. Although several approaches exist, the most common way is computing the function $Q(s, a)$, the value function associated with state-action pair (s, a). When the exact $Q(s, a)$ is available, just choosing actions greedily in each state will result in an optimal policy.However, except for really small MDPs, computing $Q(s, a)$ in closed-form is not practical.

Therefore, we use an iterative method to estimate. Consider creating a table with rows and columns represents states and actions, respectively. Then, we can use the following update rule to update our table. When the table converges, i.e. there are no changes in the table anymore, this will be a Q-function that yields an optimal policy with greedy actions.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r(s_t, a_t) + \gamma \arg\max_{a' \in \mathfrak{A}} Q(s_{t+1}, a') - Q(s_t, a_t)] \tag{2}$$

where γ is the discount factor for future rewards. What we've described here is called tabular Q-learning[9], $[r(s_t, a_t) + \gamma \arg\max_{a' \in \mathfrak{A}} Q(s_{t+1}, a')]$ is TD-target, and $[r(s_t, a_t) + \gamma \arg\max_{a' \in \mathfrak{A}} Q(s_{t+1}, a') - Q(s_t, a_t)]$ is TD-error.

However, we cannot actually use tabular Q-function in our case as the number of state-action pairs is virtually infinite. Therefore, we will use function approximation and find $\hat{Q}(s_t, a_t)$ that approximates true Q-function. A common approach is using Neural Networks as the function approximator. Then, we can use TD-error as the cost function learn the parameters of the function approximator such that it minimizes this error. To use them as inputs to NN, we need a fixed-size representation of s and a. Since G and $\mathcal{S}$ varies in size; we need an fixed-size encoding of these. Moreover, a Neural Network has limited expressive power.Thus, similar graphs G and G' or solution sets S and S' should have similar encodings for better approximation. With some abuse of notation this can be defined as: An embedding ψ is said to preserve the distances, hence satisfy similarity requirement, if $\frac{Q(s_2,a_2)}{Q(s_1,a_1)} \propto \frac{Q(\psi(s_2),\psi(a_2))}{Q(\psi(s_1),\psi(a_1))}$. We will continue to denote any representation function as ψ for the sake of simplicity and emphasize the interwoven nature of state embeddings and node embeddings. So, consider ψ as an overloaded method that calls different function depending on the input type, state or action.

So, now that we look for an encoding with similarity property, this is where node embedding methods come handy.

6 Node Embeddings

In this section we will discuss ways to find representations for s and a, i.e. graph G, solution $\mathcal{S}$, and node v. We will also discuss whether they satisfy both fixed-size and similarity consistency conditions. Actually, both of the methods that we will discuss are instances of a general framework called Graph Convolutional Network(GCN) that calculated node embeddings. Algorithm 2 shows the pseudocode for a generic GCN that can be specified to be any of these methods. In the next subsections, we will discuss how these methods can be obtained from this generic algorithm.

Algorithm 2: Generic GCN - Node Embedding

Input: Graph $G(\mathcal{V}, \mathcal{E}, \mathcal{W})$, node tags(raw features) x_v, hyperparameters K
Output: Parameter set Θ

Initialize $\boldsymbol{\mu}_v^0 \; \forall v \in G.\mathcal{V}$
for $k \in [0, K-1]$ **do**
  for $v \in G.\mathcal{V}$ **do**
    $\mathcal{N}(v) \leftarrow \{u | (v, u) \in \mathcal{E}\}$
    $\boldsymbol{\mu}_{\mathcal{N}}^k(v) \leftarrow AGGREGATE(\{\boldsymbol{\mu}_u, \forall u \in \mathcal{N}(v)\})$
    Compute $\boldsymbol{\mu}_v^{k+1} \leftarrow F(x_v, \boldsymbol{\mu}_v^k, \boldsymbol{\mu}_{\mathcal{N}}^k(v, \mathbf{W})$;
  end
  $scôre(v \leftarrow \boldsymbol{w}^T \cdot \boldsymbol{\mu}_v, \forall v \in \mathcal{V})$
end
Return: $\boldsymbol{\mu}_v \; \forall v \in \mathcal{V}$

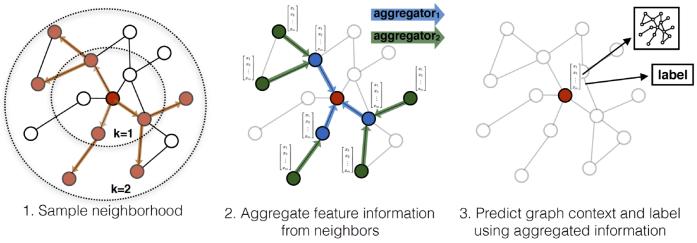

Figure 1 Flowchart for a generic Graph Convolution Network[4]

6.1 Structure2Vec(S2V)

Structure2Vec) In this subsection, a node embedding method, S2V, will be introduced. Then, a representation for $s = (G, \mathcal{S})$ will be obtained using the node embeddings.

Structure2Vec[3] provides a way to iteratively compute node embeddings for graph G and raw node features $\boldsymbol{x}_v$ for $v \in \mathcal{V}$. In one variant, node embeddings are initialized to 0 and updated using the following generic update rule:

$$\boldsymbol{\mu}_v^{k+1} \leftarrow F(x_v, \{\boldsymbol{\mu}_u^k\}_{u \in \mathcal{N}(v)}, \{w(v,u)\}_{u \in \mathcal{N}(v)}; \Theta)$$

In this equation,

- x_v is the raw feature vector for node v
- $\boldsymbol{\mu}_v^k$ is the embedding of node v in iteration t of embedding process.
- $\boldsymbol{\mu}_v^0$ is 0 as mentioned
- $\boldsymbol{\mu}_v^K$ is the desired embedding, where T is *the depth*. Depth T results in using the embeddings of neighbors up to T-hops.
- F is a generic nonlinear mapping, like a NN or kernel function
- $\mathcal{N}(v)$ is the set of neighbors of node v
- $w(v,u)$ weight of edge (v,u)

A particular instance of this generic rule is used in [6]:

$$\boldsymbol{\mu}_v^{k+1} \leftarrow ReLU(\theta_1 x_v + \theta_2 \sum_{u \in \mathcal{N}(v)} \boldsymbol{\mu}_u^k + \theta_3 \sum_{u \in \mathcal{N}(v)} ReLU(\theta_4 w(v,u)))$$

We will call this *S2V-DQN* as it is the name for the overall architecture proposed in the paper. In the paper x_v is a binary scalar, such that $x_v = 1$ if $v \in \mathcal{S}$, 0 otherwise. But, it can be generalized to a vector easily. ReLU is the rectified linear unit, defined as $ReLU(z) = max(0, z)$ and applied elementwise. The parameters are $\boldsymbol{\theta}_1 \in \mathbb{R}^{p \times q}$, $\boldsymbol{\theta}_2, \boldsymbol{\theta}_3 \in \mathbb{R}^{p \times p}$, and $\boldsymbol{\theta}_4 \in \mathbb{R}^p$. Here p is the size of $\boldsymbol{\mu}_v$ and q is the size of $\boldsymbol{x}_v$ if it is generalized a vector.

There are several issues with this architecture at this stage. Firstly, the update method is agnostic to relation between $w(v,u)$ and $\boldsymbol{\mu}_u$. That limits its expressive power. Moreover, in its update rule it does not use its embedding from the previous iteration. Although that information is theoretically available from embeddings of neighbors, it is quite indirect. Finally, although it does not cause any problems at this stage, making embeddings dependent on solution set $\mathcal{S}$ will be a big bottleneck in the future. After each transition of the MDP, i.e. after each node insertion to $\mathcal{S}$ embeddings needs to computed again. Since the embedding process is inherently recursive, this can be a restraint to solve large graphs.

Now that embeddings for nodes are ready, [6] proposes a way to represent state $s = (G, \mathcal{S})$ by combining embeddings for nodes: $\psi(s) = \sum_{u \in G.\mathcal{V}} \boldsymbol{\mu}_u^K$. Since computation of $\boldsymbol{\mu}_v$ depends on weight matrix $\mathbf{W}$ and x_v, the state will be a function of G and $\mathcal{S}$ as expected.

For actions, we will simply use the node embedding for that node. It can be seen that action representations depend on

Notice that the representations for the state s and action a have fixed size of p which is the size of embedding vector $\boldsymbol{\mu}$. Hence, it satisfies the fixed-size requirement that we had. However, since the parameters $\{\theta_i\}_{i=1}^4$ are random;it does not necessarily satisfies the similarity requirement. We will discuss how these parameters are adjusted,learned, to satisfy similarity requirement after introducing another embedding method. The short answer is Q-learning.

6.2 GraphSAGE

In this section, we will introduce another architecture for representation.

This method uses the generic architecture of GraphSAGE[4] with some modification[7]. Similar to S2V, GraphSAGE proposes a iterative way to calculate node embeddings with a different update rule. The generic update rule for GraphSAGE is:

$$\boldsymbol{\mu}_v^{k+1} \leftarrow F(\boldsymbol{\theta}_k \, Concat(\boldsymbol{\mu}_v^k, \, AGGREGATE_k(\{\boldsymbol{\mu}_u^k, \, \forall u \in \mathcal{N}(v)\})))$$

$$\boldsymbol{\mu}_v^{k+1} \leftarrow \frac{\boldsymbol{\mu}_v^{k+1}}{\|\boldsymbol{\mu}_v^{k+1}\|}$$

where $AGGREGATE$ is an aggregator like $MEANPOOL$.$Concat$ concatenates to vectors from $\mathbb{R}^a$ and $\mathbb{R}^b$ to be $\mathbb{R}^{a+b}$. $\Theta = \{\theta_k\}_{k=1}^K$ is the set of parameters that we use for different iterations.

[7] uses this architecture with following choices:

$$AGGREGATE_k(\{\boldsymbol{\mu}_u^k, \, \forall u \in \mathcal{N}(v)\}) = WeightedMEANPOOL(\{\boldsymbol{\mu}_u^k, \, \forall u \in \mathcal{N}(v)\})$$

$$= \sum_{u \in N(v)} \frac{w(v,u) \times \boldsymbol{\mu}_u^{(t)}}{\sum_{u' \in N(v)} w(v,u')}$$

$$and$$

$$F(.) = ReLU(.)$$

And it initializes $\boldsymbol{\mu}_v^0 = x_v$ where x_v is the weighted out degree of node v, i.e. $\sum_{u \in N(v)} w(v,u)$. We will call this architecture $GCOMB$ after the name of the overall architecture in [7]. GCOMB has some important differences in comparison to S2V-DQN. Firstly, by aggregating the neighbor embeddings with *Weighted MEANPOOL*, they preserve the connection between $w(v,u)$ and $\boldsymbol{\mu}_u$. Moreover, it directly uses $\boldsymbol{\mu}_v^k$ when computing $\boldsymbol{\mu}_v^{k+1}$. Finally, the embeddings are not dependent on solution set $\mathcal{S}$, importance of which will be discussed later.

After obtaining the node embeddings, [7] proposes following representations for state s and action a:

$$\psi(s_t) = Concat(MAXPOOL(\{\boldsymbol{\mu}_{v'}, v' \in \mathcal{S}_t\}), MAXPOOL(\{\boldsymbol{\mu}_{v'}, v' \in G_t.\mathcal{V} \setminus \mathcal{S}_t\}))$$

$$\psi(a_t) = \boldsymbol{\mu}_v, v = a_t.v$$

So, $s_t \in \mathbb{R}^{2p}$ and $a_t \in \mathbb{R}^p$.

Notice that, similar to S2V, fixed-size requirement is satisfied but parameters needs to be adjusted, learned, for similarity requirement. The learning process will be discussed in the next section; but short answer is supervised learning.

7 Learning Embedding Parameters

In this section we will discuss how to learn parameters for the embeddings discussed in the previous section so they satisfies the similarity requirement. Remember that obtained representations will be used by Q-approximator and the similarity requirement is needed for a good Q-approximation, since it has limited expressive power.

Remarkably, for S2V-DQN[6] uses Q-learning to tune its parameters. In other words, instead of considering the representations as fixed inputs to the Q-approximator, it pipelines embedding component in front of Q-approximator and learn their parameters collectively. We will talk about this learning process in Q-approximators and Q-learning section.

7.1 Learning Parameters for GCOMB

Contrary to S2V-DQN, GCOMB uses an intermediate supervision signal to learn the parameters for embeddings. So, it treats embeddings as fixed inputs. The design of the supervision signal uses some engineering choices. Firstly, we shall remember the aim of the embeddings: the states and nodes that are similar in the sense that they would give similar results in the actual Q-functions should have similar encodings. The problem is that we don't know what the actual Q-function looks like. However, if the optimal solutions to the problems were available, they could be reverse-engineered to find the optimal policy, which uses the actual Q-function greedily. For instance, if the optimal policy chooses a_t in state s_t, it implies that $Q(s_t, a_t) >= Q(s_t, a)\ \forall a$.

However, the optimal solutions are not available either.So, GCOMB uses the greedy algorithm to find near-optimal solutions.Now that solutions are available, they can be used to assess how good each node is. Then, embeddings can be learned to get encodings for good ones close to each other, and encodings for bad ones close each other. At this point, a metric for quality should be defined. A straight-forward idea is to assign value 1 to each node $v \in \mathcal{S}$ where $\mathcal{S}$ is the solution set; and value 0 to others.

However, this metric can be misleading as it does not distinguish among good nodes or among bad nodes, by definition of being binary. This would cause some efficiency issues in the learning time. Moreover, this is worsened by the fact depending on the nature of the problem; only one of the two very similar nodes can make it to $\mathcal{S}$. Consider the case $f(G, \{v_1\}) = f(G, \{v_2\})$ and $f(G, \{v_1, v_2\}) - f(G, v_1) = f(G, \{v_1, v_2\}) - f(G, \{v_2\}) =$, then only one v_1, v_2 can be chosen to $\mathcal{S}$.

To tackle this issue, the greedy algorithm is modified to be probabilistic such that instead of $v \leftarrow \arg\max_{\forall v \in V \setminus S} f(S \cup \{v\}) - f(S)$, we use $v \sim \boldsymbol{p}(v') \propto f(\mathcal{S} \cup \{v'\}) - f(\mathcal{S})$, i.e. nodes are chosen with probabilities proportional to their marginal gain. Then this algorithm is run m times and $gain_i(v)$ defined as follows:

$$gain_i(v_i) = \begin{cases} f(\mathcal{S}_i^{t+1}) - f(\mathcal{S}_i^t), \text{if node } v \text{ is added to } \mathcal{S} \text{ in } t^{th} \text{ iteration of experiment } i \\ 0, \text{otherwise} \end{cases}$$

Moreover, to make the resulting embedding generalizable over the same problem with different termination conditions, a budget b in [7], they modify the termination condition to be: Terminate experiment i when $gain_i(v_{t+1}) - gain_i(v_t) < \delta$ such that v_t, v_{t+1} are chosen in iterations t and $t+1$ of experiment i, respectively.

Finally, $score(v)$ is defined as follows:

$$score(V) = \frac{\sum_i^m gain_i(v)}{\sum_i^m f(G, \mathcal{S}_i)}$$

Since this supervision is not fit to node embeddings directly, as it does not say what should our embedding actually be, we will pipeline an extra component after the embedding. Define $\hat{score}(v) = \boldsymbol{w}^T \boldsymbol{\mu}_v$ and minimize the mean squared error

$$J(\{w\} \cup \{\theta_k\}_{k=1}^K) = \frac{1}{|\mathcal{V}|} \sum_{\forall v \in \mathcal{V}} (score(v) - \hat{score}(v))^2$$

So, we can run SGD on parameter set $\{w\} \cup \{\theta_k\}_{k=1}^K$ with cost function J. Algorithm 3 summarizes the embedding learning process in a simple manner. Note that for the sake of simplicity we didn't include batch training.

Notice that w is not actually useful for embedding purposes, it is only used as a part of parameter sharing in the pipelined architecture. To make the most out of the supervision signal, we use a minimal output stage of unshared parameter w just an inner product.This will help require the model to do most of the learning using shared parameters $\{\theta_k\}_{k=1}^K$.

Since a simple function of $\boldsymbol{\mu}_v$ is able to predict a metric of the quality of a node, we can infer that node embeddings $\mu(.)$ preserves the distances, hence have the similarity property.

As the embeddings are learned, inputs to Q-approximator are obtained. Now, parameters of Q-approximator needs to be learned in order to find the optimal policy.

Algorithm 3: GCOMB - Learning the Embeddings

Input: A set of training graphs $\mathbb{D}$, hyperparameters N, K
Output: Parameter set Θ

for *epoch* $\in [1, N]$ **do**
  Draw graph G from set $\mathbb{D}$
  Calculate score(v) for G
  $\boldsymbol{\mu}_v^0 \leftarrow \boldsymbol{x}_v, \forall v \in G.\mathcal{V}$
  for $k \in [1, K]$ **do**
    for $v \in G.\mathcal{V}$ **do**
      $\mathcal{N}(v) \leftarrow \{u | (v, u) \in \mathcal{E}\}$
      $\boldsymbol{\mu}_{\mathcal{N}}^k(v) \leftarrow WeightedMEANPOOL(\{\boldsymbol{\mu}, \forall u \in \mathcal{N}(v)\})$
      $\boldsymbol{\mu}_v^k \leftarrow ReLU(\boldsymbol{\theta}^k Concat(\boldsymbol{\mu}_{\mathcal{N}}^{k-1}(v), \boldsymbol{\mu}_v^{k-1}))$
      $\boldsymbol{\mu}_v^k \leftarrow \frac{\boldsymbol{\mu}_v^k}{\|\boldsymbol{\mu}_v^k\|}$
    end
    $\hat{score}(v \leftarrow \boldsymbol{w}^T \cdot \boldsymbol{\mu}_v, \forall v \in \mathcal{V})$
  end
  $J(\{w\} \cup \{\theta_k\}_{k=1}^K) = \frac{1}{|\mathcal{V}|} \sum_{\forall v \in \mathcal{V}} (score(v) - \hat{score}(v))^2$
  Run SGD on parameter set $\{w\} \cup \{\theta_k\}_{k=1}^K$ to minimize J
end
Return: Θ

8 Q-approximation and Q-learning

Now, we will discuss the architectures these two papers [7][6] proposes to approximate the actual Q-function. These architectures consist of a function approximator and a technique to learn its parameters.

8.1 S2V-DQN

S2V-DQN uses $\hat{Q}(s_t, a_t; \Theta) = \theta_5^T ReLU\ (Concat(\ \theta_6 \sum_{u \in G_t.\mathcal{V}} \boldsymbol{\mu}_u^T,\ \theta_7 \boldsymbol{\mu}_v^T\ ))$ where $\theta_5 \in \mathbb{R}^{2p}$ and $\theta_6, \theta_7 \in \mathbb{R}^{p \times p}$. Remember that $\boldsymbol{\mu}_v$ is a function of $\{\theta_i\}_{i=1}^4$ and these parameters still not learned.

Therefore, Q-learning is used to learn the parameters of both embeddings and Q-approximator. This can be interpreted as allowing Q-approximator to learn the embeddings as it likes. Also, the whole pipeline of embedders and Q-approximator can be considered as one very complex Q-approximator.

Here, a modified version of the TD-target will be used : $\sum_{i=0}^{n-1} r(s_{t+i}, a_{t+i}) + \gamma max_{a'} \hat{(Q)}(s_{t+n}, a)$ where $\hat{Q}$ is the Q-approximator and n is a hyperparameter. The intuition is to update the values using long-term returns rather than 1-step rewards; hence, allowing $\hat{Q}$ to get foresighted faster. The problem here is that this is not an off-policy method anymore which is a fact that can hurt the training if not acknowledged and addressed properly.

Consider this, under a policy π, a path p is taken from state s_t starting with action a_t to s_{t+n} and total reward is $R_{t,t+n}$. Now, since the path p depends on the policy π, total reward $R_{t,t+n}$ depends on π too. So, transitions experienced under a different policy is useless. Notice that when $n = 1$ the policy dependency is broken, as the Q gives the value for given s_t and a_t. Therefore, a_t is already chosen, it does not matter which policy π is used. There are methods to do $n-$step Q-learning but they require storing more information.

Regardless, the authors keep the remaining algorithm with regular Q-learning and use replay buffer. Replay buffer is a memory that stores previously experienced transition usually in the form (s_t, a_t, r_t, s_{t+1}), and it is one of the most common techniques. By allowing the use of previous experiences multiple times, it increases sample efficiency. Moreover, as it stored previous experiences to be used later, it stabilizes the learning by preventing forgetting. Finally, it allows the use of mini-batch training and increase the efficiency of computations. S2V-DQN changes the stored information to $(s_t, a_t, R_{t,t+n}, s_{t+n})$. As discussed before using long-term rewards instead of 1-step rewards without proper adjustments to the algorithm breaks many assumptions behind the analysis of Q-learning. However; it looks like they do not experience any serious issues empirically. In short, the resulting algorithm can be seen in Algorithm 4

8.2 GCOMB

GCOMB uses the following as its Q-approximator:

$$\hat{Q}(s_t, a_t, \boldsymbol{\vartheta}) = \vartheta_1^T ReLU(\ (Concat\ (\ \vartheta_2 \psi(s_t)), \vartheta_3 \psi(a_t)\ ))$$

$\psi(s_t)$ and $\psi(a_t)$ are as defined in 7.1. So $\vartheta_1 \in \mathbb{R}^{3p}$, $\vartheta_2 \in \mathbb{R}^{2p \times 2p}$, and $\vartheta_3 \in \mathbb{R}^{p \times p}$. Learning part of the GCOMB is same S2V-DQN. It uses Q-learning with same modifications. Only difference is GCOMB treats the embeddings as fixed-input and trains only Q-approximator parameters, implied by the change of parameter notation from θ to ϑ.

Algorithm 4: Q-learning

Input: A set of training graphs $\mathbb{D}$, hyperparameters M, N, n, L, T
Output: Parameter set Θ

Initialize experience replay memory M to capacity N
for *episode* $e = 1$ **to** L **do**
  Draw graph G from dataset $\mathbb{D}$
  Initialize the state to $s_0 = (G, \emptyset)$ /* $s_t = (G, \mathcal{S}_t)$) */
  for *step* $t = 1$ **to** T **do**
    $a_t = \begin{cases} \textit{random node } v \in G.\mathcal{V} \setminus \mathcal{S}_t, & \text{with probability } \epsilon \\ \arg\max_{v \in G.\mathcal{V} \setminus \mathcal{S}_t} \hat{Q}(s_t, v; \Theta), & \text{otherwise} \end{cases}$
    Add a_t to partial solution: $S_{t+1} := S_t \cup \{a_t\}$
    if $t >= n$ **then**
      Add tuple $(s_{t-n}, a_{t-n}, R_{t-n}, s_t)$ to M
      Sample random batch $B \sim M$
      Update Θ by SGD over $(y - \hat{Q}(s_t, a_t; \Theta))^2$ for B
    end
  end
end
Return: Θ

Finally, we obtained $\hat{Q}$ that approximates the true Q-function. Then, if we are solving the problem over graph G, at each step t, we can simply choose the node with the highest Q value. That is, $a_t = \arg\max_{v \in G.\mathcal{V}} \hat{Q}(s_t, v)$. Note that for S2V, we still need to recompute the embedding at each step. Whereas in GCOMB, we can compute the embeddings once at the beginning and use them as long as we are on graph G.

9 Discussions

In this section, a very brief description of experimental results from the papers[6][7] will be followed by some discussions relating to possible issues with the proposed methods and some future work directions.

9.1 Experimental Results

[6] shows that their method can solve graphs of 500-1000 nodes quickly and with a good approximation ratio. They use TSP, Minimum Vertex Cover, and MaxCut problems as benchmarks. Moreover, they show that their method can generalize to ~ 1000 nodes training samples, even when they just train on graphs of ~ 50 nodes. [7] puts a great emphasize on scalability. They show that their method can solve graphs of $\sim 10^6$ nodes $100 - 150$ times faster than greedy algorithm while keeping the quality on par with greedy on benchmark problems Influence Maximization, Maximum Coverage Problem, Maximum Vertex Cover. We leave a more detailed discussion of experimental results to the respective papers.

9.2 Reliability of Solutions

One important problem with data-driven approaches is that there is no theoretical guarantee for performance. Although it is possible to derive some statistical guarantees like confidence

intervals, they are usually based on some assumptions. One of the most important ones is sampling from a known data distribution and using the same data distribution for training and testing. In other words, not having any completely novel data in testing. However, this assumption often is not accurate in real-world data. So, in order to avoid unexpected behavior of the model, developing methods to assess the novelty of a data point is important.

In this next two sections, we will discuss two approaches for novelty detection.

9.3 Traditional Novelty Detection

Novelty detection is a well-explored field as a result of decades long research. There are several methods from signal processing, statistics, and machine learning communities. For example, using auto-encoders is one of these methods[1]. Auto-encoders are usually used to obtain a low-dimensional representation of a high-level input in an unsupervised way utilizing neural networks. It has a bow-tie like architecture, the first half is an encoder and the second part is a decoder. Decoder tries to reconstruct the input, but this requires having an encoding that captures important features of the input. In a statistical sense, it tunes its parameters to capture the most of the variance. If a data point results with a large reconstruction error, it can be concluded that it didn't occur in the training data enough; otherwise the model would adjust its parameters to capture it too. Therefore, reconstruction error can be used as a metric for novelty.

However, most of these methods are not directly applicable to graphs. The problems with graphs that make traditional machine learning methods fail in combinatorial problems, still stands with novelty detection. So, we need a representation that can handle graphs of different sizes and exploits permutation-invariance like features of graphs.

This report explains how representations that satisfy these requirement can be obtained using Graph Convolutional Networks but those methods can fail here. Because the previous task was a node selection problem, embeddings for individual nodes were needed. Hence, local properties of graphs were more emphasized than general ones.This can be seen by comparing how node and state embeddings were obtained. Usually state embeddings is as simple as a *MEANPOOL* or *MAXPOOL* of node embeddings. On the other hand, novelty detection does not require embeddings for individual nodes. Only representation of the graph that captures general properties is needed.Therefore, it is a dilemma between using available low quality representation and learning a new representation.If we choose learning a new representation, we need to find a way to cleverly represent the graphs.

In short, traditional approach offers a results from decades-long research but using their methods over graphs still require substantial research. Moreover, it is very likely to cause considerable increase in training complexity. Also, the method might be slow over very large graphs during testing. Therefore, we will propose a different approach that uses TD-error as novelty metric.

9.4 TD-Error Novelty Detection

In addition to Q-learning, many RL algorithms that use value-learning tries to minimize TD-error. The intuition is as $\hat{Q}$ converges to true Q, prediction errors will be minimized. Also, the more an agent experiences a part of state space, the more its predictions get more accurate. TD-error gets smaller in frequently-visited parts of the state space whereas it can be very large in not-visited parts. Then, TD-error can be used as a metric for novelty.

Since, it uses only $\hat{Q}$, there is no overhead for the training. When testing, $\hat{Q}(s_t, a_t)$ and $\arg\max_a \hat{Q}(s_{t+1}, a)$ will be already calculated to choose appropriate action in timestep t and

$t+1$, respectively. However, $r(s_t, a_t)$ is not always available to the agent, and it may require some computation. For example, Influence Maximization uses expectation of output of a stochastic process, as objective function. Therefore, computing reward may require several simulations; hence, introduce some considerable overhead in testing.

To alleviate this problem, the prediction error can be modified as

$$\begin{aligned}\hat{Q}(s_t, a_t) + f(G_t, \mathcal{S}_t) - [f(G_{t'}, \mathcal{S}_{t'}) + \hat{Q}(s_{t'}, a_{t'})] \\ &= \hat{Q}(s_t, a_t) - [(f(G_{t'}, \mathcal{S}_{t'}) - f(G_t, \mathcal{S}_t)) + \hat{Q}(s_{t'}, a_{t'})] \\ &= \hat{Q}(s_t, a_t) - [\sum_{i=t}^{t'-1} r(s_i, a_i) + \hat{Q}(s_{t'}, a_{t'})]\end{aligned}$$

Note that last equation follows from the definition of $\boldsymbol{R}$ in the MDP definition. This new error give us a chance to reduce the number of objective value queries by giving up the ability of checking short-term prediction errors. We believe that it wouldn't be difficult to find a good rule of thumb for $t' - t$, such that it balances between prediction error information and computation time.

The main problem with this approach is it lacks any theoretical or empirical analysis for now. The only paper using a similar idea and could be found at the time this report was written, is [8]. It uses TD-error as a novelty metric for exploration, as a novel approach for Intrinsic Motivation. However, it also lacks sufficient analysis.

This concludes our discussion of novelty detection methods for reliability of solutions. It is important notice that novelty detection does not convey much information about performance of the model given a data point. There can be graphs that occur frequently during training but the model cannot solve well because they are inherently not suited to the method.

An alternative task can be predicting the performance of the model on a datapoint, i.e. optimality of produced solution. However, this is a much harder problem and requires substantial research.

9.5 Suboptimality of Supervision

In GCOMB, $score(v)$ was used as supervision signal and it was calculated using a variant of Greedy Approach.(See Section 7.1) The issue is greedy approach produces suboptimal solutions itself. Hence, the supervision signal is usually very noisy. However, as exact algorithms are not tractable over large graphs; there is no way to get the optimal solution for a given graph in an efficient way; unless they come in pairs. If training graphs can be generated with known solutions in an efficient way, it could be a way to tackle this noisy supervision signal.

There actually exist some research on this, particularly for TSP. [2] They formulate the TSP problem on a generic graph as a 0-1 integer LP problem. Then, they create a solvable proxy problem with same objective function and looser constraint, that is $\Omega_{original} \subseteq \Omega_{proxy}$ where Ω is the constraint set. By this design, optimal solutions to the proxy problem that $\in \Omega_{original}$ are optimal solutions to the original problem as well. Using primal-dual methods, analytical expressions for solutions in terms of dual variables are obtained. Then, they reintroduce loosened constraints.Finally, graphs with known solutions can be obtained by varying the dual variables.In this way they can generate symmetric, asymmetric, and triangular-inequality TSP problems.

This paper empirically shows that instances that generated by this method are not easier than the random ones. Although this is an important result, there are important issues.

Since this is a problem-dependent method to generate solutions, it can generate only TSP problems. So, this kind of methods should be developed for each problem unless a general framework is developed. However, such a framework is either not possible, not practical or not easy to find in the foreseeable future. The fact that only relevant paper that could be found is from 1988, reinforces this claims.

10 Conclusion

In this report, our goal was to provide an introduction to the research on Reinforcement Learning for Combinatorial Optimation over Graphs. Instead of merely summarizing the papers, we took a more textbook-like approach. Firstly, a unifying problem definition(See section 3 is provided. Secondly, it is formulated as an MDP and RL approach to solve the problem is explained. Then, two papers [7][6] are reviewed as example methods taking the RL approach. Finally, a discussion of several issues that can point to some future research directions is given. Although this project was done mainly for educational purposes, it contributes to the surveyed paper by providing the necessary technical background and a unified interpretation.

References

1 Tsatsral Amarbayasgalan, Bilguun Jargalsaikhan, and Keun Ho Ryu. Unsupervised novelty detection using deep autoencoders with density based clustering. *Applied Sciences*, 8(9):1468, 2018.

2 Jeffrey L. Arthur and James O. Frendewey. Generating travelling-salesman problems with known optimal tours. *The Journal of the Operational Research Society*, 39(2):153–159, 1988. URL: `http://www.jstor.org/stable/2582378`.

3 Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2702–2711. JMLR.org, 2016. URL: `http://proceedings.mlr.press/v48/daib16.html`.

4 Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

5 David Kempe, Jon M. Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In Lise Getoor, Ted E. Senator, Pedro M. Domingos, and Christos Faloutsos, editors, *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, pages 137–146. ACM, 2003. `doi:10.1145/956750.956769`.

6 Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 6348–6358, 2017. URL: `http://papers.nips.cc/paper/7214-learning-combinatorial-optimization-algorithms-over-graphs`.

7 Akash Mittal, Anuj Dhawan, Sahil Manchanda, Sourav Medya, Sayan Ranu, and Ambuj K. Singh. Learning heuristics over large graphs via deep reinforcement learning. *CoRR*, abs/1903.03332, 2019. URL: `http://arxiv.org/abs/1903.03332`, `arXiv:1903.03332`.

8 Riley Simmons-Edler, Ben Eisner, Daniel Yang, Anthony Bisulco, Eric Mitchell, Sebastian Seung, and Daniel Lee. Reward prediction error as an exploration objective in deep rl, 2019. `arXiv:1906.08189`.

9 Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.