Learning Heuristics over Large Graphs via Deep Reinforcement Learning

Akash Mittal[1*], **Anuj Dhawan**[1*], **Sahil Manchanda**[1*], **Sourav Medya**[2], **Sayan Ranu**[1], **Ambuj Singh**[2]
[1]Indian Institute of Technology Delhi
[2]University of California, Santa Barbara
[1]{cs1150208, Anuj.Dhawan.cs115, sayanranu}@cse.iitd.ac.in , [2]{medya, ambuj}@cs.ucsb.edu

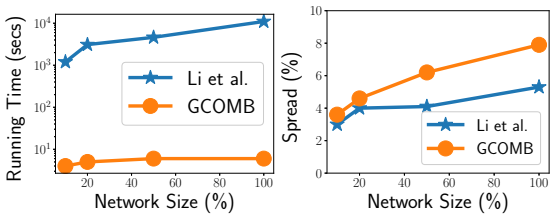
Abstract

Combinatorial optimization problems on graphs are routinely solved in various domains. Recently, it has been shown that heuristics for solving combinatorial problems can be learned using a machine learning-based approach. While existing techniques have primarily focussed on obtaining high-quality solutions, the aspect of scalability to billion-sized graphs has not been adequately addressed. In this paper, we propose a deep reinforcement learning framework called GCOMB to learn algorithms that can solve combinatorial problems over graphs at scale. Besides considering the traditional NP-hard combinatorial problems, we apply our framework to the popular and challenging data mining problem of *Influence Maximization*. GCOMB utilizes Graph Convolutional Network (GCN) to generate node embeddings that predict potential solution nodes. These embeddings are next fed to a Q-learning framework, which learns the combinatorial nature of the problem and predicts the final solution set. Through extensive evaluation on several synthetic and billion-sized real networks, we establish that GCOMB is more than 100 times faster than the state of the art while retaining the same quality of the solution sets.

Introduction

Combinatorial optimization problems on graphs appear routinely in various applications such as viral marketing in social networks (Kempe, Kleinberg, and Tardos 2003), computational sustainability (Dilkina, Lai, and Gomes 2011), and health-care (Wilder et al. 2018). These optimization problems are often combinatorial in nature, which results in NP-hardness. Therefore, designing an exact algorithm is infeasible and polynomial-time algorithms, with or without approximation guarantees, are often desired and used in practice (Goyal, Lu, and Lakshmanan 2011a; Jung, Heo, and Chen 2012a; Medya et al. 2018). Furthermore, these graphs are often dynamic in nature and the approximation algorithms need to be run repeatedly at regular intervals. Since real-world graphs may contain millions of nodes and edges, this entire process becomes tedious and time-consuming.

To provide a concrete example, consider the problem of viral marketing on social networks through *Influence Maximization*(Arora, Galhotra, and Ranu 2017). Given a graph G and a budget b, the goal is to select b nodes (users) from the graph such that their endorsement of a certain product (ex: through a tweet) is expected to initiate a cascade that reaches the largest number of nodes in the graph. It has been shown that this problem is NP-hard by reducing it to the max-coverage problem (Kempe, Kleinberg, and Tardos 2003). Advertising through social networks is a common practice today and needs to solved repeatedly due to the networks being dynamic in nature. Furthermore, even the greedy approximation algorithm has been shown to not scale on large networks (Arora, Galhotra, and Ranu 2017) resulting in a large body of research work(Tang, Shi, and Xiao 2015) (Tang, Xiao, and Shi 2014), (Goyal, Lu, and Lakshmanan 2011b), (Leskovec et al. 2007), (Chen, Yuan, and Zhang 2010), (Jung, Heo, and Chen 2012b), (Ohsaka et al. 2014), (Kempe, Kleinberg, and Tardos 2003).



(a) Scalability (b) Quality

Figure 1: **Comparison of (a) scalability and (b) quality on subgraphs extracted from the YouTube social network (Table 1) at budget=20. The formal definition of quality, i.e., influence spread, is defined in the Problem Formulation section.**

At this juncture, we highlight two key observations. First, although the graph is changing, the underlying model generating the graph is likely to remain the same. Second, the nodes that get selected in the answer set of the approximation algorithm may have certain properties common in them. Motivated by these observations, we ask the following question: *Given an optimization problem P on graph G from a distribution D of graph instances, can we learn an approximation algorithm and solve the problem on an unseen graph generated from distribution D?*

The above observation was first highlighted by (Dai et al. 2017), where they show that it is indeed possible to *learn* combinatorial algorithms on graphs. Subsequently, this algorithm is further improved by (Li, Chen, and Koltun 2018).

*The authors with * have equal contribution

In this work, we extend this line of work further by addressing the following weaknesses of existing works.

• **Scalability**: The primary focus of both (Li, Chen, and Koltun 2018) and (Dai et al. 2017) have been on obtaining quality that is as close to the optimal as possible. Efficiency studies, however, are limited to networks containing only hundreds of thousands nodes. Real-life networks contain millions of nodes and billions of edges (See. Table 1). Can combinatorial algorithms be learned on billion-scale networks? Our study shows that there is scope to improve. Specifically, we apply (Li, Chen, and Koltun 2018) for the Influence Maximization problem on subgraphs extracted from the YouTube social network. The subgraphs are constructed by randomly selecting $X\%$ of the edges from the network. As can be seen in Fig. 1, (Li, Chen, and Koltun 2018) consumes more than 1 hour on a 30% subgraph, and more than 3 hours on the whole YouTube network. In contrast the proposed algorithm, GCOMB, finishes in only 6 seconds, while being also better in quality. We elaborate further on the quality aspect in the Experiments section.

The non-scalability of (Li, Chen, and Koltun 2018) arises due to the design of the framework. Specifically, there are two components: **(1)** a graph convolutional network (GCN) to learn and predict the individual *value* of each node in the graph, and **(2)** a *tree-search* component to analyze the dependence among the node values and identify the set of nodes that collectively work well as a group. Following the tree-search, GCN is repeated on a reduced graph and this process continues iteratively. In this process, there are two scalability bottlenecks. First, tree-search is time consuming on large graphs since this is an actual search procedure and not a prediction. Second, GCN is called repeatedly on iteratively reduced graphs, which does not scale well for large real-life networks.

• **Generalizability to real-life combinatorial problems:** The non-scalability of (Li, Chen, and Koltun 2018) arises from its non-generalizability to a larger class of combinatorial optimization problems. Specifically, (Li, Chen, and Koltun 2018) propose a learning-based heuristic for the Maximal Independent Set problem (MIS). When the combinatorial problem to be solved is not MIS, (Li, Chen, and Koltun 2018) suggest that we map that problem to MIS and then apply their algorithm. While this approach works well for some problems such as Vertex Cover, for problems that are not easily mappable to MIS, the efficacy is compromised as visible in Fig. 1b.

Contributions: In this work, we bridge the above mentioned gaps. Our contributions are summarized as follows:

- **Supervised and Problem-agnostic Framework:** We propose an end-to-end prediction framework called GCOMB to learn algorithms for combinatorial problems on graphs at scale. GCOMB first generates node embeddings through *Graph Convolutional Networks (GCN)*. These embeddings encode the effect of a node on the budget-constrained solution set. Next, these embeddings are fed to a neural network to learn a Q-function and predict the solution set. Unlike (Dai et al. 2017), GCOMB is supervised and thus able to make higher quality predictions. Compared to (Li, Chen, and Koltun 2018), GCOMB uses deep reinforcement learning instead of tree-search to learn and predict the combinatorial nature of the problem.
- **Application:** *Influence Maximization (IM)* on social networks has been a popular research topic in the data mining community (Arora, Galhotra, and Ranu 2017). We apply GCOMB for IM and show that we not only perform better than (Li, Chen, and Koltun 2018), but also improve upon the state-of-the-art algorithm built specifically for IM(Tang, Shi, and Xiao 2015). GCOMB provides similar quality answer sets, while being up to 100 times faster.
- **Massive Scalability:** We benchmark GCOMB on billion-sized networks, where GCOMB finishes in less than a minute. GCOMB produces quality on par with GCN-TreeSearch, while being orders of magnitudes faster. On the whole, GCOMB promises to be the first learning-based approach that can be applied on massive real networks without compromising on quality.

Problem Formulation and Preliminaries

Formally, we define our learning task as follows.

Problem 1. *Given a combinatorial optimization problem P over graphs drawn from distribution D,* learn *a heuristic to solve problem P on an unseen graph G generated from D.*

The input to our problem is therefore a set of training graphs $\{G_1, \ldots, G_n\}$ from distribution D and the set of solution sets $\{S_1, \cdots, S_n\}$. Given an unseen graph G from D, we need to predict its solution set S for problem P.

Instances of the proposed problem

Several classical graph combinatorial problems, such as vertex cover, set cover, etc., are listed in (Li, Chen, and Koltun 2018) and (Dai et al. 2017). In this work, we also focus on the real-life combinatorial problem of Influence Maximization since it is of interest to both the academia (Arora, Galhotra, and Ranu 2017) as well as industry (Kumar et al. 2013).

Definition 1 (Social Network.)**.** *A social network is denoted as an edge-weighted graph $G(V, E, W)$, where V is the set of nodes (users), E is the set of directed edges (relationships), and W is the set of edge-weights corresponding to each edge in E.*

We use $W(u, v)$ to denote the weight of edge $e = (u, v)$.

The objective in *influence maximization (IM)* is to maximize the *spread* of influence in a network through activation of an initial set of b *seed* nodes.

Definition 2 (Seed Node)**.** *A node $v \in V$ that acts as the source of information diffusion in the graph $G(V, E, W)$ is called a seed node. The set of seed nodes is denoted by S.*

Definition 3 (Active Node)**.** *A node $v \in V$ is deemed active if either (1) It is a seed node ($v \in S$) or (2) It is influenced by a previously active node $u \in V_a$. Once activated, the node v is added to the set of active nodes V_a.*

Initially, the set of active nodes V_a is the seed nodes S. The spread of influence is guided by the *Independent Cascade (IC)* model.

Algorithm 1 The greedy approach

Require: $G = (V, E)$, optimization function $f(.)$, budget b
Ensure: solution set S, $|S| = b$
1: $S \leftarrow \emptyset$
2: $i \leftarrow 0$
3: **while** $(i < b)$ **do**
4: $\quad v^* \leftarrow \arg\max_{\forall v \in V \setminus S}\{f(S \cup \{v\}) - f(S)\}$
5: $\quad S \leftarrow S \cup \{v^*\}, i \leftarrow i + 1$
6: **Return** S

Definition 4 (Independent Cascade)**.** *Under the IC model, time unfolds in discrete steps. At any time-step i, each newly activated node $u \in V_a$ gets one independent attempt to activate each of its outgoing neighbors v with a probability $p_{(u,v)} = W(u, v)$. The spreading process terminates when in two consecutive time steps the set of active nodes remain unchanged.*

Definition 5 (Spread)**.** *The spread $\Gamma(S)$ of a set of seed nodes S is defined as the total proportion of nodes that are active at the end of the information diffusion process. Mathematically, $\Gamma(S) = \frac{|V_a|}{|V|} \times 100$.*

Since the information diffusion is a stochastic process the measure of interest is the *expected* value of spread. The *expected* value of spread $\sigma(\cdot) = \mathbb{E}[\Gamma(\cdot)]$ is computed by simulating the spread function a large number of times. The goal in IM, is therefore to solve the following problem.

Problem 2 (Influence Maximization (IM))**.** *Given a budget b and a social network G, select a set S of b seeds such that the expected value of spread $\sigma(S) = \mathbb{E}[\Gamma(S)]$ is maximized.*

The greedy approach

The greedy approach is one of the most popular and well-performing strategies to solve combinatorial problems on graphs. Alg. 1 presents the pseudocode. The input to the algorithm is a graph $G = (V, E)$, an optimization function $f(S)$ on a set of nodes S, and budget b. Starting from an empty solution set S, the solution is built iteratively by adding the "best" node to S in each iteration (lines 3-5). The best node $v^* \in V \setminus S$ is the one that provides the highest *marginal gain* on the optimization function (line 4). The process ends after b iterations where b is the budget.

GCOMB

GCOMB consists of two phases: the *training phase* and the *testing* phase. The input to the training phase is a set of graphs and the optimization function $f(.)$ corresponding to the combinatorial problem being solved. The output of the training phase is a sequence of two different neural networks with their corresponding learned parameters. In the testing phase, the inputs are identical as in the greedy algorithm,

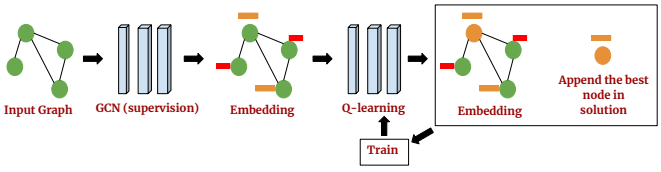


Figure 2: **The flowchart of the training phase of GCOMB.**

which are the graph $G = (V, E)$, the optimization function $f(.)$ and the budget b. The output of the testing phase is the solution set, which is constructed using the learned neural networks from the training phase.

Fig. 2 presents the pipeline of the training phase. The training phase can be divided into two parts: a network embedding phase through *Graph Convolutional Network (GCN)* and a Q-learning phase. Given a training graph and its solution set, the GCN learns network embeddings that separates the potential solution nodes from the rest. Next, the embeddings of only the potential solution nodes are fed to a Q-learning framework, which allows us to predict those nodes that collectively form a good solution set. The next sections elaborate further on these two key components of the training phase.

Embedding via GCN

Our goal is to learn node embeddings that can predict the nodes that are likely to be part of the answer set. Towards that, one can set up a classification-based pipeline, where, given a training graph $G = (V, E)$ and its greedy solution set S corresponding to the optimization function $f(S)$, a node v is called *positive* if $v \in S$; otherwise it is negative. This approach, however, has two key weaknesses. **(1)** First, it assumes all nodes that are not a part of S to be equally bad. In reality this may not be the case. To elaborate, consider the case where $f(\{v_1\})$=$f(\{v_2\})$, but the marginal gain of node v_2 given $S = \{v_1\}$, i.e., $f(\{v_1, v_2\}) - f(\{v_1\})$, is 0 and vice versa. In this scenario, only one of v_1 and v_2 would be selected in the answer set although both are of equal quality on their own. **(2)** Second, the budget is provided as an input parameter only at querying/testing time. Thus, it is not even clear what the size of S should be.

To address the first issue, we *sample* from the *solution space* and learn embeddings that reflect the marginal gain provided by a node. To sample from the solution space, we perform a *probabilistic* version of the greedy search in Alg. 1. Specifically, in each iteration, instead of selecting the node with the highest marginal gain, we choose a node with probability proportional to its marginal gain. The probabilistic greedy algorithm runs m times to construct m different solution sets $\mathbb{S} = \{S_1, \cdots, S_m\}$ and the score of node $v \in V$ is set to proportion of marginal gain contributed by v across all m solution sets.

$$score(v) = \frac{\sum_i^m gain_i(v)}{\sum_i^m f(S_i)} \quad (1)$$

Here, $gain_i(v)$ denotes the marginal gain contribution of v to S_i.

Now, to overcome the issue of budget, that is the size of each solution set for training, we run greedy till *convergence* of the marginal gains. More specifically, let v_{t-1} and v_t be the nodes added to S_i in iteration $t-1$ and t respectively. The probabilistic greedy algorithm terminates if $gain_i(v_{t-1}) - gain_i(v_t) \leq \Delta$, where Δ is a small value.

Given $score(v)$ for each node, our next task is to learn embeddings that can predict its score. Towards that, we use a Graph Convolutional Network (GCN) (Hamilton, Ying, and Leskovec 2017). The pseudocode for this component

Algorithm 2 Graph Convolutional Network (GCN)

Require: $G = (V, E)$, $\{score(v)$, input features $\mathbf{x}_v\ \forall v \in V\}$, depth K, weight matrices $\mathbb{W}^k,\ \forall k \in [1, k]$ and weight vector $\mathbf{w}$, dimension size d.

Ensure: d-dimensional vector representations $\boldsymbol{\mu}_v,\ \forall v \in V$

1: $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v,\ \forall v \in V$
2: **for** $k \in [1, K]$ **do**
3: **for** $v \in V$ **do**
4: $N(v) \leftarrow \{u|(v, u) \in E\}$
5: $\mathbf{h}_N^k(v) \leftarrow$ Weighted MEANPOOL $\left(\left\{h_u^{k-1}, \forall u \in N(v)\right\}\right)$
6: $\mathbf{h}_v^k \leftarrow ReLU\left(\mathbb{W}^k \cdot \text{CONCAT}\left(\mathbf{h}_N^k(v), h_v^{k-1}\right)\right)$
7: $\mathbf{h}_v^k \leftarrow \frac{\mathbf{h}_v^k}{\|\mathbf{h}_v^k\|}$
8: $\boldsymbol{\mu}_v \leftarrow \mathbf{h}_v^K,\ \forall v \in V$
9: $\widehat{score}(v) \leftarrow \mathbf{w}^T \cdot \boldsymbol{\mu}_v\ , \forall v \in V$

is provided in Alg. 2. Each iteration in the outer loop represents the *depth* (line 2). In the inner loop, we iterate over all nodes (line 3). While iterating over node v, we fetch the current representations of a sampled set of v's neighbors and *aggregate* them through a weighted MEANPOOL layer (lines 4-5). Specifically, for dimension i, we have $\mathbf{h}_N^k(v)_i = \sum_{\forall u \in N(v)} \frac{W(v,u) \times h_{u_i}^{k-1}}{\sum_{\forall u' \in N(v)} W(v,u')}$. $W(v, u)$ is the edge weight. The aggregated vector is next *concatenated* with the representation of v, which is then fed through a fully connected layer with *ReLU* activation function (line 6), where ReLU is the *rectified linear unit* ($ReLU(z) = max(0, z)$). The output of this layer becomes the input to the next iteration of the outer loop. Intuitively, in each iteration of the outer loop, nodes aggregate information from their local neighbors, and with more iterations, nodes incrementally receive information from neighbors of higher depth (i.e., distance).

At depth 0, the embedding of each node v is $h_v^0 = \mathbf{x}_v$, while the final embedding is $\boldsymbol{\mu}_v = h_v^K$ (line 9). In the fully-connected layers, Alg. 2 requires the parameter set $\mathbb{W} = \{\mathbb{W}^k, k = 1, 2, \cdots, K\}$ to apply the non-linearity (line 6). Intuitively, $\mathbb{W}^k$ is used to propagate information across different depths of the mode.

To train the parameter set $\mathbb{W}$ and obtain predictive representations, the final representations are passed through another fully connected layer to obtain their predicted value $\widehat{score}(v)$ (line 8). The parameters Θ for the proposed framework are therefore the weight matrices $\mathbf{W}$ and the weight vector $\mathbf{w}$. To learn Θ, we apply stochastic gradient descent on the *mean squared error* loss function.Specifically,

$$J(\Theta) = \frac{1}{|V|} \sum_{\forall v \in V} (score(v) - \widehat{score}(v))^2 \tag{2}$$

Defining $\mathbf{x}_v$: The initial feature vector $\mathbf{x}_v$ at depth 0 should have the raw features that are relevant with respect to the combinatorial problem being solved. For example, in Influence Maximization (IM), the summation of the outgoing edge weights of a node is an indicator of its own spread. As we discuss later in Experiments section, we use this as the only feature for not only IM, but also for the vertex cover and max coverage problems.

Algorithm 3 Learning Q-function

Require: $\boldsymbol{\mu}_v, \forall v \in V$, hyper-parameters M, N relayed to fitted Q-learning, number of episodes L and sample size T.

Ensure: Learn parameter set Θ

1: Initialize experience replay memory M to capacity N
2: **for** episode $e \leftarrow 1$ to L **do**
3: **for** step $t \leftarrow 1$ to T **do**
4: $v_t \leftarrow \begin{cases} \text{random node } v \notin S_t \text{ with probability } \epsilon \\ \text{argmax}_{v \notin S_t} Q'(S_t, v, \Theta) \text{ otherwise} \end{cases}$
5: $S_{t+1} \leftarrow S_t \cup \{v_t\}$
6: **if** $t \geq n$ **then**
7: Add tuple $(S_{t-n}, v_{t-n}, \sum_{i=t-n}^{t} r(S_i, v_i), S_t)$ to M
8: Sample random batch B from M
9: Update the parameters Θ by SGD for B
10: **return** Θ

Learning Q-function

While GCN captures the individual importance of a node towards a particular combinatorial problem, through Q-learning (Sutton and Barto 2018), we capture nodes that collectively form a good solution set. More specifically, given some set of nodes S and a node $v \notin S$, we aim to predict $Q(S, v)$ (intuitively long-term reward for adding v to S) through the surrogate function $Q'(S, v; \Theta)$. For any Q-learning task, we need to define the following five aspects: state space, actions, rewards, policy and termination.

- **State space:** A state is the aggregation of two sets of nodes: nodes selected in the current solution set S and those not selected, i.e., $V \backslash S$. Thus, the state corresponding to solution set S is captured using two vectors: $\boldsymbol{\mu}_S = \text{MAXPOOL}\left(\{\boldsymbol{\mu}_v,\ v \in S\}\right)$ and $\boldsymbol{\mu}_{V \backslash S} = \text{MAXPOOL}\left(\{\boldsymbol{\mu}_v,\ v \notin S\}\right)$. We use MAXPOOL since it better captures the diversity of the feature vectors than mean. Empirically, we also observe better results.
- **Action:** An action corresponds to adding a node $v \notin S$ (represented as $\boldsymbol{\mu}_v$) to the solution set.
- **Rewards:** The reward function at state S is the marginal gain of adding node v to S, i.e. $r(S, v) = f(S \cup \{v\}) - f(S)$.
- **Policy:** The policy $\pi(v|S)$ (given state S and action) is deterministic and, as in the greedy policy, selects the node with the highest *predicted* marginal, i.e.,

$$\pi(v|S) = \arg\max_{v \notin S} Q'(S, v; \Theta) \tag{3}$$

- **Termination:** We terminate when $|S| = b$; b is the budget.

Learning Θ: Alg. 3 presents the pseudocode of learning the parameter set Θ. We partition Θ into four weight vectors $\Theta_1, \Theta_2, \Theta_3, \Theta_4$ such that, $Q'(S, v; \Theta) = \Theta_4 \cdot \boldsymbol{\mu}_{\boldsymbol{S,v}}$, where

$$\boldsymbol{\mu}_{\boldsymbol{S,v}} = ReLU\left\{\text{CONCAT}\left(\Theta_1 \cdot \boldsymbol{\mu}_S, \Theta_2 \cdot \boldsymbol{\mu}_{V \backslash S}, \Theta_3 \cdot \boldsymbol{\mu}_v\right)\right\} \tag{4}$$

If the dimension of the initial node embeddings is d, the dimensions of the weight vectors are as follows: $\Theta_4 \in \mathbb{R}^{3d \times 1}; \Theta_1, \Theta_2, \Theta_3 \in \mathbb{R}^{d \times d}$. In Eq. 4, *ReLU* is applied element-wise to its input vector.

The standard Q-learning updates parameters in a single episode via a SGD step to minimize the squared loss.

$$J(\Theta) = (y - Q'(S_t, v_t\Theta))^2 \quad (5)$$

$$\text{where } y = \gamma \cdot max_v \left(Q'\left(S_{t+1}, v; \Theta\right)\right) + r(S_t, v_t) \quad (6)$$

S_t denotes current solution set, γ is the discount factor, and v_t is the considered node. To better learn the parameters, we perform n-step Q-learning instead of 1-step Q-learning. n-step Q-learning incorporates delayed rewards, where the final reward of interest is received later in the future during an episode (lines 6-9). This avoids the myopic setting of 1-step update. The key idea here is to wait for n steps so that the approximator's parameters are updated and therefore, more accurately estimate future rewards. To incorporate n-step rewards, Eq. 6 is modified as follows.

$$y = \gamma \cdot max_v(Q'(S_{t+n}, v; \Theta)) + \sum_{t=0}^{n-1} r(S_t, v_t) \quad (7)$$

For efficient learning of the parameters, we perform *fitted Q-iteration* (Riedmiller 2005), which results in faster convergence using a neural network as a function approximator (Mnih et al. 2013) (M defined in line 1 of Alg. 3).

Summary

The entire pipeline of GCOMB works as follows.

- **Training Phase:** Given a training graph G, and optimization function $f(S)$, learn parameter set θ_{GCN} and Θ_Q corresponding to the GCN component and Q-learning component. This is a *one-time, offline* computation. The sub-tasks in the training phase are:
  - Learn node embeddings $\boldsymbol{\mu}_v$, $\forall v \in V$ along with θ_{GCN}.
  - Feed $\boldsymbol{\mu}_v$, $\forall v \in V$ to Q-learning and learn Θ_Q.
- **Testing Phase:** Given an unseen graph G,
  - Embed nodes using θ_{GCN}.
  - Iteratively compute the solution set based by adding the node $v^* = \arg\max_{\forall v \in V} Q'(S_i, v; \Theta_Q)$, where S_i is the solution set in the i^{th} iteration.

Experimental Results

In this section, we benchmark GCOMB and establish:

- **Scalability:** GCOMB is significantly faster than S2V-DQN (Dai et al. 2017) and GCN-TreeSearch (Li, Chen, and Koltun 2018) and scales to billion-sized networks without compromising on quality.
- **Influence Maximization (IM):** GCOMB is orders of magnitude faster than the state of the technique for IMM (Tang, Shi, and Xiao 2015), while obtaining similar quality in influence spread.

Experimental Setup

All experiments are performed on a machine running Intel Xeon E5-2698v4 processor with 20 cores, having 8 Nvidia 1080 Ti GPU cards with 12GB GPU memory, and 512 GB RAM with Ubuntu 16.04 operating system.

Datasets: We use both synthetic and real datasets for experiments. The real datasets[1] are listed in Table 1. For synthetic datasets, we generate graphs from two different models:

- **Barabási–Albert (BA):** In BA, the default edge density is set to 4, i.e., $|E| = 4|V|$. We use the notation BA-X to denote the size of the generated network, where X is the number of nodes.

[1] http://snap.stanford.edu/data/index.html

- **Bipartite Graph (BP):** (Dai et al. 2017) proposes a model to generate bipartite graphs as follows: Given the number of nodes, they are partitioned into two sets with 20% nodes in one side and the rest in other. The edge between any pair of nodes from different partitions is generated with probability 0.1.

Combinatorial Problems: We evaluate on the three different NP-hard combinatorial problems: **(1) Influence Maximization (IM)**, **(2) Max Vertex Cover (MVC)**: Given a graph $G = (V, E)$ and budget b, select a subset of nodes $S \subseteq V$, $|S| = b$ such that the maximum number of edges are incident on at least one node in S, and **(3) Max Coverage Problem (MCP)**: Given a bipartite graph over two sets of nodes V_1 and V_2, and a budget b, select a subset of nodes $S \subset V_1$, $|S| = b$, such that the maximum number nodes from V_2 have at least one neighbor from S.

Baselines: We compare GCOMB with *S2V-DQN* (Dai et al. 2017), *GCN-TreeSearch* (Li, Chen, and Koltun 2018) and *Greedy*. In addition, for the problem of IM, we also compare with the state-of-the-art algorithm *IMM*(Tang, Shi, and Xiao 2015). Greedy guarantees a $1 - 1/e$ approximation for all three problems. IMM is not only the fastest algorithm for IM(Arora, Galhotra, and Ranu 2017), but also provides the same theoretical guarantee on quality as Greedy. Since our focus is on really large networks, computing the optimal solution is not feasible since it does not scale on the networks used in our evaluation. For S2V-DQN and IMM we use the code shared by the authors. For GCN-TreeSearch, we extend the original code by authors for IM and MCP problems since these are not natively supported. IMM is implemented in C++ and all remaining codes are in python.

Other Settings: GCN is trained for 200 epochs with a learning rate of 0.0005, a dropout rate of 0.1 and a convolution depth (K) of 2. For training the n-step Q-Learning neural network, n is set to 2 and a learning rate of 0.0001 is used. In each epoch of training, 8 training examples are sampled uniformly from the Replay Memory M as described in Alg. 3. For a fair comparison of GCOMB with S2V-DQN and GCN-TreeSearch, in all our experiments, we keep the training time (1 hour), training dataset and testing dataset identical for all methods. The raw feature x_v of node v is set to the summation of its outgoing edge weights. For undirected, unweighted graphs, this reduces to the degree. All results are reported by averaging over 5 training instances.

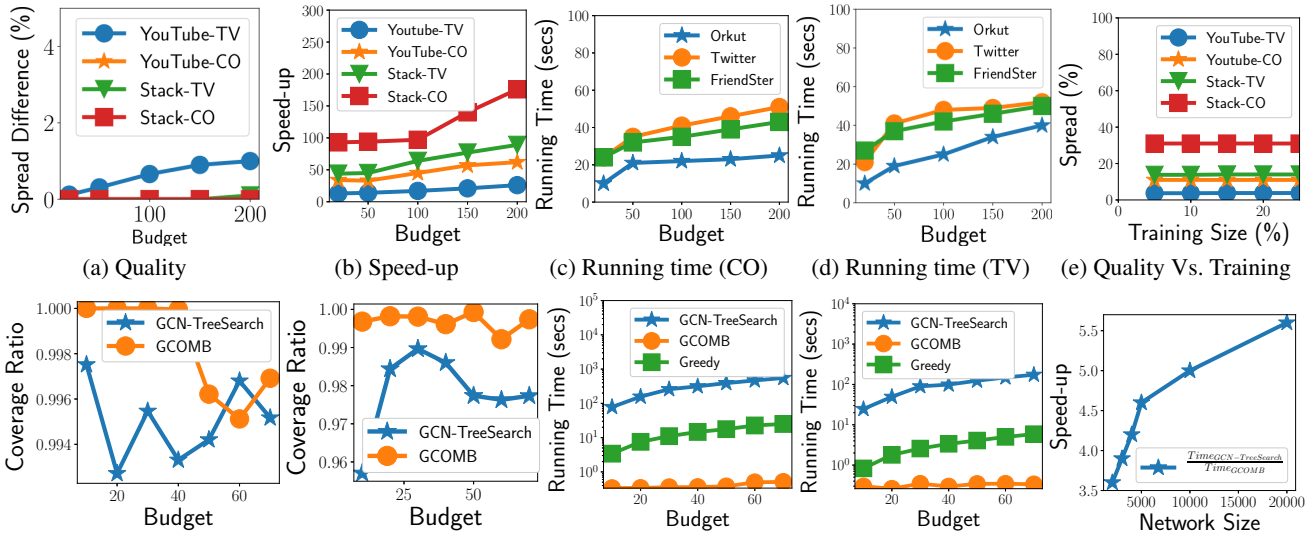
Performance in Influence Maximization (IM)

IM is the hardest of the three combinatorial problems since even estimating the marginal gain of adding a node is #P-hard(Kempe, Kleinberg, and Tardos 2003).

Setup: One important parameter in IM are the edge weights (influence probabilities). We assign edge weights

Name	$\|V\|$	$\|E\|$
Gowalla	196.5K	950.3K
Brightkite	58.2K	214K
YouTube (YT)	1.13M	2.99M
StackOverflow (Stack)	2.69M	5.9M
Orkut	3.07M	117.1M
Twitter (TW)	41.6M	1.5B
FriendSter (FS)	65.6M	1.8B

Table 1: **Datasets used for our empirical evaluation.**



(a) Quality (b) Speed-up (c) Running time (CO) (d) Running time (TV) (e) Quality Vs. Training

(f) Gowalla (g) Brightkite (h) Gowalla (i) Brightkite (j) Speed-up in MCP

Figure 3: **(a) Comparison of spread quality between IMM and GCOMB. (b) Speed-up achieved by GCOMB over IMM. (c-d) Running times of GCOMB. (e) Impact of training set size on spread quality. For IMM, we set $\epsilon = 0.5$ in all experiments. (f-g) Coverage quality of GCOMB and GCN-TreeSearch against the greedy approach in MVC. (h-i) Running times of GCOMB, GCN-TreeSearch and Greedy in MVC. (j) Speed-up achieved by GCOMB over GCN-TreeSearch in the MCP problem.**

using the following two models(Arora, Galhotra, and Ranu 2017): **(1) Constant (CO:)** all edge weights are set to 0.1, **(2) Tri-valency (TV):** Edge weights are sampled randomly from the set $\{0.1, 0.01, 0.001\}$.

Fig. 1 shows that GCN-TreeSearch is more than 1000 times slower than GCOMB. In addition, the quality is also inferior. In Fig. 1, we use the TV edge weight model at budget $b = 20$. The non-scalability of GCN-TreeSearch stems from two factors: first, GCN is called repeatedly and second, TreeSearch is extremely slow as computing the marginal gain is #P-hard. In contrast, GCOMB calls GCN only once and the marginal gain of adding a node is predicted whose complexity is linear to the embedding dimension (256). This non-scalability of GCN-TreeSearch limits us from benchmarking it on larger datasets. In subsequent experiments for IM, we thus compare the performance of GCOMB against IMM, which is specifically built for the IM problem.

The quality of GCN-TreeSearch suffers in IM due to three key reasons. **(1)** In typical problems like vertex cover or set cover, the coverage of a node is deterministic. In IM, the coverage, or spread, is an *expectation*. To estimate this expectation well, we need to perform a large number of MC simulations (1000 or more(Arora, Galhotra, and Ranu 2017)). However, in TreeSearch, where we search the overlap among expected spread of various nodes, we are limited to a small number (50) due to scalability issues already pointed out. Hence, the quality suffers. GCOMB does not suffer since it simply predicts the spread and resultant marginal gains. **(2)** The GCN architectures are different. While we use GraphSage(Hamilton, Ying, and Leskovec 2017), GCN-TreeSearch uses (Kipf and Welling 2017). Furthermore, GCN-TreeSearch does not use any node features. **(3)**. GCN-TreeSearch uses cross-entropy loss to predict the answer set of only a specific budget. Thus, if the input budget is different, the quality may suffer. In contrast, GCOMB uses a budget-independent scoring function (Eq. 1), which is trained using mean squared error. In Fig. 1, the training and testing budget for GCN-TreeSearch is kept the same.

We next evaluate GCOMB and IMM on five of the largest real datasets from Table 1. We train our prediction model on a subgraph sampled out of YT by randomly selecting 30% of the edges. IMM crashes on both the billion-sized datasets of TW and FS, as well as Orkut. This result is not surprising since similar results have been reported in (Arora, Galhotra, and Ranu 2017). Thus, we compare the quality of GCOMB with IMM on YT and Stack. Fig. 3a reports the results in terms of *spread difference*.

$$\text{Spread Difference} = \frac{\sigma(S_{IMM}) - \sigma(S_{\text{GCOMB}})}{\sigma(S_{IMM})} \times 100$$

where S_{IMM} and S_{GCOMB} are answer sets computed by IMM and GCOMB respectively. As visible, the spread difference is $\approx$ 1% in YT-TV and $\approx$ 0% in all of the remaining setups. The true impact of GCOMB is realized when this result is considered in conjunction with Fig. 3b, which shows GCOMB is 30 to 200 times faster than IMM. In this plot, speed-up is measured as simply $\frac{time_{IMM}}{time_{\text{GCOMB}}}$ where $time_{IMM}$ and $time_{\text{GCOMB}}$ are the running times of IMM and GCOMB respectively. On even larger billion-sized datasets where IMM crashes, GCOMB finishes within 50 seconds

Graph	S2V-DQN	GCOMB	GCN-TreeSearch
BA-10k	1935	4625	**4633**
BA-20k	2464	**6533**	**6533**
BA-50k	4138	**10398**	10378
BA-100k	NA	14473	**14484**
BA-500k	NA	30427	**30626**
Gowalla	NA	**76793**	76344
Brightkite	NA	**14759**	14634

Table 2: **Performance in Max Vertex Cover (MVC). The best result in each dataset is highlighted in bold. All models are trained on BA-1k. We omitted S2V-DQN for larger datasets since its performance was not competitive enough.**

(See Figs. 3c-3d). Finally, in Fig. 3e, we evaluate the impact of training data size on quality. We observe that even when we use only 5% of YT to train, the result is almost identical to training with a 25% subgraph. Overall, these results indicate that GCOMB produces high quality solutions while being scalable in both online testing time as well as offline training.

Performance in Max Vertex Cover (MVC) and Max Cover (MCP)

MVC: First, we study the performance in MVC on BA networks as well as Gowalla and Brightkite with budget $b = 30$. All methods are trained on BA graphs with $1k$ nodes. Table 2 presents the results. Both GCOMB and GCN-TreeSearch produce results that are very close to each other and significantly better than S2V-DQN. From an efficiency viewpoint, Table 3 reveals that, as in IM, GCOMB is dramatically faster than GCN-TreeSearch, particularly on larger datasets. As discussed earlier, the higher efficiency of GCOMB stems from the usage of a single call to GCN and prediction of marginal gains using Q-learning instead of TreeSearch.

To further analyze the quality and efficiency, we next compare both aspects with the Greedy algorithm (Alg 1). For MVC, Greedy provides an $1 - \frac{1}{e}$ approximation guarantee on quality. Figs. 3f-3g present the quality of GCOMB and GCN-TreeSearch in terms of *coverage ratio* as the budget is varied. Coverage ratio computes the ratio of the edge coverage achieved by the benchmarked method against the coverage achieved by Greedy, i,e, $\frac{Coverage_X}{Coverage_{Greedy}}$, where X is either GCOMB or GCN-TreeSearch. We observe that although both techniques are very close to Greedy, neither is able to beat Greedy in quality. However, as shown in Figs. 3h-3i, GCOMB is 10 times faster than Greedy and up to 500 times faster GCN-TreeSearch. We also note the GCN-TreeSearch is inferior to Greedy in both quality and efficiency.

MCP: Finally, we benchmark performance in MCP on bipartite graphs. All methods are trained on a bipartite graph with 1000 nodes and tested for the MCP problem with budget 15. As visible in Table 4, GCOMB outperforms both S2V-DQN and GCN-TreeSearch across all graph sizes. GCN-TreeSearch produces results that are very close to GCOMB, while S2V-DQN is significantly weaker. As in IM and MVC, the true impact of GCOMB is visible in Fig. 3j where we plot the speed-up achieved by GCOMB over GCN-TreeSearch against the network size. As visible, GCOMB is 4 to 5 times faster and, therefore, better in both quality and efficiency.

Previous Work

There has been recent interest in solving graph combinatorial problems with neural networks and reinforcement learning (Bello et al. 2016; Dai et al. 2017). Learning-based approaches are useful in producing good empirical results for NP-hard problems. The methods proposed in (Bello et al. 2016) are generic and do not explore structural properties of graphs, and use sample-inefficient policy gradient methods. Our work is very relevant to the work by Dai et al. (Dai et al. 2017). They have investigated the same problem with a network embedding approach combined with a reinforcement learning technique. The main difference in this paper is the supervised part in our framework. Instead of an end-to-end learning, we use GCN to filter out the potential nodes in the solution. The second major difference is that our method is scalable and we apply it to the very popular data mining problem, Influence Maximization.

The same problem has been studied by Li et al. (Li, Chen, and Koltun 2018) via a supervised approach using GCN. However, their technique involves multiple calls of GCN after selection of one element in the solution set. Their method also has an exploration part in a brute-force manner. Unsurprisingly besides independent set problem, this method is not scalable in other problems (e.g. Influence Maximization). Our method applies GCN once and finds the potential solution. Unlike in the previous one, we also apply reinforcement learning to construct our solution. Experimental results show that our method is more applicable to important problems such as Influence Maximization.

Among other interesting work, for branch-and-bound algorithms, He et al. studied the problem of learning a node selection policy (He, Daume III, and Eisner 2014). Another examples include pointer networks (Vinyals, Fortunato, and Jaitly 2015) proposed by Vinyals et al. and reinforcement learning approaches by Silver et al. (Silver et al. 2016) to learn strategies for the game Go.

Graph	GCOMB (secs)	GCN-TreeSearch (secs)
BA-10k	**0.29**	21
BA-20k	**0.29**	32
BA-50k	**0.35**	66
BA-100k	**0.49**	126
BA-500k	**0.69**	626
Gowalla	**0.49**	262
Brightkite	**0.33**	81

Table 3: **Prediction times in Max Vertex Cover. The best result in each dataset is highlighted in bold.**

Graph	S2V-DQN	GCOMB	GCN-TreeSearch
BP-2k	1210	**1357**	1316
BP-3k	1726	**2031**	1979
BP-4k	2307	**2678**	2649
BP-5k	2920	**3344**	3292
BP-10k	5940	**6603**	6479
BP-20k	12071	**13040**	12972

Table 4: **Performance in Max Coverage Problem. The best result in each dataset is highlighted in bold. All algorithms are trained on BP-1k.**

Conclusion

In this paper, we have proposed a *deep reinforcement learning* based framework called GCOMB to learn algorithms for combinatorial problems on billion-scale graphs. GCOMB first generates node embeddings through *Graph Convolutional Networks (GCN)*. These embeddings encode the effect of a node on the budget-constrained solution set. Next, these embeddings are fed to a neural network to learn a Q-function and predict the solution set. Through extensive experiments on both real and synthetic datasets containing up to a billion edges, we show that GCOMB is dramatically faster than the state-of-the-art techniques, while being at least as good in quality. In addition, GCOMB also improves the state of the art for the Influence Maximization problem, where it achieves same quality as IMM, while be-

ing up to 200 times faster. Overall, for the first time, GCOMB unleashes the potential of learning combinatorial algorithms on graphs at scale.

References

[Arora, Galhotra, and Ranu 2017] Arora, A.; Galhotra, S.; and Ranu, S. 2017. Debunking the myths of influence maximization: An in-depth benchmarking study. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 651–666. ACM.

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

[Chen, Yuan, and Zhang 2010] Chen, W.; Yuan, Y.; and Zhang, L. 2010. Scalable influence maximization in social networks under the linear threshold model. In *ICDM*, 88–97.

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

[Dilkina, Lai, and Gomes 2011] Dilkina, B.; Lai, K. J.; and Gomes, C. P. 2011. Upgrading shortest paths in networks. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 76–91. Springer.

[Goyal, Lu, and Lakshmanan 2011a] Goyal, A.; Lu, W.; and Lakshmanan, L. V. 2011a. Simpath: An efficient algorithm for influence maximization under the linear threshold model. In *2011 IEEE 11th international conference on data mining*, 211–220. IEEE.

[Goyal, Lu, and Lakshmanan 2011b] Goyal, A.; Lu, W.; and Lakshmanan, L. V. 2011b. Simpath: An efficient algorithm for influence maximization under the linear threshold model. In *ICDM*, 211–220.

[Hamilton, Ying, and Leskovec 2017] Hamilton, W.; Ying, Z.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, 1024–1034.

[He, Daume III, and Eisner 2014] He, H.; Daume III, H.; and Eisner, J. M. 2014. Learning to search in branch and bound algorithms. In *Advances in neural information processing systems*, 3293–3301.

[Jung, Heo, and Chen 2012a] Jung, K.; Heo, W.; and Chen, W. 2012a. Irie: Scalable and robust influence maximization in social networks. In *2012 IEEE 12th International Conference on Data Mining*, 918–923. IEEE.

[Jung, Heo, and Chen 2012b] Jung, K.; Heo, W.; and Chen, W. 2012b. IRIE: Scalable and robust influence maximization in social networks. In *ICDM*, 918–923.

[Kempe, Kleinberg, and Tardos 2003] Kempe, D.; Kleinberg, J.; and Tardos, É. 2003. Maximizing the spread of influence through a social network. In *KDD*.

[Kipf and Welling 2017] Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

[Kumar et al. 2013] Kumar, V.; Bhaskaran, V.; Mirchandani, R.; and Shah, M. 2013. Practice prize winner—creating a measurable social media marketing strategy: Increasing the value and roi of intangibles and tangibles for hokey pokey. *Marketing Science* 32(2):194–212.

[Leskovec et al. 2007] Leskovec, J.; Krause, A.; Guestrin, C.; Faloutsos, C.; VanBriesen, J.; and Glance, N. 2007. Cost-effective outbreak detection in networks. In *KDD*, 420–429.

[Li, Chen, and Koltun 2018] Li, Z.; Chen, Q.; and Koltun, V. 2018. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, 537–546.

[Medya et al. 2018] Medya, S.; Vachery, J.; Ranu, S.; and Singh, A. 2018. Noticeable network delay minimization via node upgrades. *Proceedings of the VLDB Endowment* 11(9):988–1001.

[Mnih et al. 2013] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[Ohsaka et al. 2014] Ohsaka, N.; Akiba, T.; Yoshida, Y.; and Kawarabayashi, K. 2014. Fast and accurate influence maximization on large networks with pruned monte-carlo simulations. In *AAAI*, 138–144.

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

[Silver et al. 2016] Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature* 529(7587):484.

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

[Tang, Shi, and Xiao 2015] Tang, Y.; Shi, Y.; and Xiao, X. 2015. Influence maximization in near-linear time: A martingale approach. In *SIGMOD*, 1539–1554.

[Tang, Xiao, and Shi 2014] Tang, Y.; Xiao, X.; and Shi, Y. 2014. Influence maximization: Near-optimal time complexity meets practical efficiency. In *SIGMOD*, 75–86.

[Vinyals, Fortunato, and Jaitly 2015] Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer networks. In *Advances in Neural Information Processing Systems*, 2692–2700.

[Wilder et al. 2018] Wilder, B.; Ou, H. C.; de la Haye, K.; and Tambe, M. 2018. Optimizing network structure for preventative health. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, 841–849.