Learning Heuristics over Large Graphs via Deep Reinforcement Learning

Sahil Manchanda, Akash Mittal[*], Anuj Dhawan[*]
Indian Institute of Technology Delhi
{sahil.manchanda,cs1150208,Anuj.Dhawan.cs115}@cse.iitd.ac.in

Sourav Medya[1], Sayan Ranu[2], Ambuj Singh[3]
[1]Northwestern University,[2]Indian Institute of Technology Delhi
[3]University of California Santa Barbara
[1]sourav.medya@kellogg.northwestern.edu
[2]sayanranu@cse.iitd.ac.in,[3]ambuj@ucsb.edu

Abstract

There has been an increased interest in discovering heuristics for combinatorial problems on graphs through machine learning. While existing techniques have primarily focused on obtaining high-quality solutions, scalability to billion-sized graphs has not been adequately addressed. In addition, the impact of budget-constraint, which is necessary for many practical scenarios, remains to be studied. In this paper, we propose a framework called GCOMB to bridge these gaps. GCOMB trains a Graph Convolutional Network (GCN) using a novel *probabilistic greedy* mechanism to predict the quality of a node. To further facilitate the combinatorial nature of the problem, GCOMB utilizes a Q-learning framework, which is made efficient through *importance sampling*. We perform extensive experiments on real graphs to benchmark the efficiency and efficacy of GCOMB. Our results establish that GCOMB is 100 times faster and marginally better in quality than state-of-the-art algorithms for learning combinatorial algorithms. Additionally, a case-study on the practical combinatorial problem of Influence Maximization (IM) shows GCOMB is 150 times faster than the specialized IM algorithm IMM with similar quality.

1 Introduction and Related Work

Combinatorial optimization problems on graphs appear routinely in various applications such as viral marketing in social networks [14, 4], computational sustainability [8], health-care [33], and infrastructure deployment [20, 23, 24, 22]. In these *set combinatorial problems*, the goal is to identify the set of nodes that optimizes a given objective function. These optimization problems are often NP-hard. Therefore, designing an exact algorithm is infeasible and polynomial-time algorithms, with or without approximation guarantees, are often desired and used in practice [13, 31]. 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* [2, 14]. Given a budget b, the goal is to select b nodes (users) 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. This problem is NP-hard [14]. Advertising through social networks is a common practice today and needs to solved repeatedly due to the graphs being dynamic

[*]denotes equal contribution

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

in nature. Furthermore, even the greedy approximation algorithm does not scale to large graphs [2] resulting in a large body of research work [31, 13, 16, 26, 14, 5, 32, 6].

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 [7]: *Given a set combinatorial problem P on graph G and its corresponding solution set S, can we learn an approximation algorithm for problem P and solve it on an unseen graph that is similar to G?*

1.1 Limitations of Existing Work

The above observations were first highlighted by S2V-DQN [7], where they show that it is indeed possible to *learn* combinatorial algorithms on graphs. Subsequently, an improved approach was proposed in GCN-TREESEARCH [19]. Despite these efforts, there is scope for further improvement.

• **Scalability**: The primary focus of both GCN-TREESEARCH and S2V-DQN have been on obtaining quality that is as close to the optimal as possible. Efficiency studies, however, are limited to graphs containing only hundreds of thousands nodes. To provide a concrete case study, we apply GCN-TREESEARCH for the Influence Maximization problem on the YouTube social network. We observe that GCN-TREESEARCH takes one hour on a graph containing a million edges (Fig. 3a; we will revisit this experiment in § 4.3). Real-life graphs may contain billions of edges (See. Table 1a).

• **Generalizability to real-life combinatorial problems:** GCN-TREESEARCH proposes a learning-based heuristic for the Maximal Independent Set problem (MIS). When the combinatorial problem is not MIS, GCN-TREESEARCH suggests that we map that problem to MIS. Consequently, for problems that are not easily mappable to MIS, the efficacy is compromised (ex: Influence Maximization).

• **Budget constraints:** Both GCN-TREESEARCH and S2V-DQN solve the decision versions of combinatorial problems (Ex. set cover, vertex cover). In real life, we often encounter their budget-constrained versions, such as max-cover and Influence Maximization [14].

Among other related work, Gasse et al. [9] used GCN for learning branch-and-bound variable selection policies, whereas Prates et al. [27] focused on solving Travelling Salesman Problem. However, the proposed techniques in these papers do not directly apply to our setting of set combinatorial problems.

1.2 Contributions

At the core of our study lies the observation that although the graph may be large, only a small percentage of the nodes are likely to contribute to the solution set. Thus, pruning the search space is as important as prediction of the solution set. Both S2V-DQN [7] and GCN-TREESEARCH [19] have primarily focused on the prediction component. In particular, S2V-DQN learns an end-to-end neural model on the entire graph through reinforcement learning. The neural model integrates node embedding and Q-learning into a single integrated framework. Consequently, the model is bogged down by a large number of parameters, which needs to be learned on the entire node set. As a result, we will show in §. 4 that S2V-DQN fails to scale to graphs beyond $20,000$ nodes.

On the other hand, GCN-TREESEARCH employs a two-component framework: **(1)** a graph convolutional network (GCN) to learn and predict the individual *value* of each node, and **(2)** a *tree-search* component to analyze the dependence among nodes and identify the solution set that collectively works well. Following tree-search, GCN is repeated on a reduced graph and this process continues iteratively. This approach is not scalable to large graphs since due to repeated iterations of GCN and TreeSearch where each iteration of tree-search has $O(|E|)$ complexity; E is the set of edges.

Our method GCOMB builds on the observation that computationally expensive predictions should be attempted only for promising nodes. Towards that end, GCOMB has two separate components: **(1)** a GCN to prune *poor* nodes and learn embeddings of *good* nodes in a *supervised* manner, and **(2)** a Q-learning component that focuses only on the *good* nodes to predict the solution set. Thus, unlike S2V-DQN, GCOMB uses a *mixture* of supervised and reinforcement learning, and does not employ an end-to-end architecture. Consequently, the prediction framework is lightweight with a significantly reduced number of parameters.

When compared to GCN-TREESEARCH, although both techniques use a GCN, in GCOMB, we train using a novel *probabilistic greedy* mechanism. Furthermore, instead of an iterative procedure of repeated GCN and TreeSearch calls, GCOMB performs a single forward pass through GCN

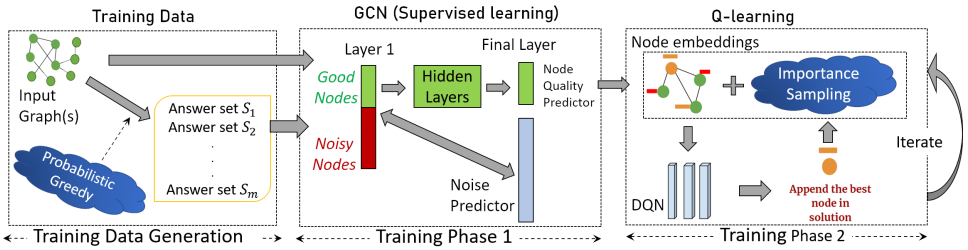


Figure 1: The flowchart of the training phase of GCOMB.

during inference. In addition, unlike TreeSearch, which is specifically tailored for the MIS problem, GCOMB is problem-agnostic [2]. Finally, unlike both S2V-DQN and GCN-TREESEARCH, GCOMB uses lightweight operations to prune *poor* nodes and focus expensive computations only on nodes with a high potential of being part of the solution set. The pruning of the search space not only enhances scalability but also removes noise from the search space leading to improved prediction quality. Owing to these design choices, **(1)** GCOMB is scalable to billion-sized graphs and up to 100 times faster, **(2)** on average, computes higher quality solution sets than S2V-DQN and GCN-TREESEARCH, and **(3)** improves upon the state-of-the-art algorithm for Influence Maximization on social networks.

2 Problem Formulation

Objective: *Given a budget-constrained set combinatorial problem P over graphs drawn from distribution D,* learn *a heuristic to solve problem P on an unseen graph G generated from D.*

Next, we describe three instances of budget-constrained set combinatorial problems on graphs.

Maximum Coverage Problem on bipartite graph (MCP): Given a bipartite graph $G = (V, E)$, where $V = A \cup B$, and a budget b, find a set $S^* \subseteq A$ of b nodes such that coverage is maximized. The coverage of set S^* is defined as $f(S^*) = \frac{|X|}{|B|}$, where $X = \{j | (i, j) \in E, i \in S^*, j \in B\}$.

Budget-constrained Maximum Vertex Cover (MVC): Given a graph $G = (V, E)$ and a budget b, find a set S^* of b nodes such that the coverage $f(S^*)$ of S^* is maximized. $f(S^*) = \frac{|X|}{|E|}$, where $X = \{(i, j) | (i, j) \in E, i \in S^*, j \in V\}$.

Influence Maximization (IM) [2]: Given a budget b, a social network G, and a information diffusion model $\mathcal{M}$, select a set S^* of b nodes such that the expected diffusion spread $f(S^*) = \mathbb{E}[\Gamma(S^*)]$ is maximized. (See App. A in supplementary for more details).

3 GCOMB

The input to the training phase is a set of graphs and the optimization function $f(\cdot)$ corresponding to the combinatorial problem in hand. The output is a sequence of two separate neural graphs, GCN [10] and Q-learning network, with their corresponding learned parameters Θ_G and Θ_Q respectively. In the testing phase, the inputs include a graph $G = (V, E)$, the optimization function $f(\cdot)$ and the budget b. The output of the testing part is the solution set of nodes constructed using the learned neural networks. Fig. 1 presents the training pipeline. We will now discuss each of the phases.

3.1 Generating Training Data for GCN

Our goal is to learn node embeddings that can predict "quality", and thereby, identify those nodes that are likely to be part of the answer set. We could adopt a classification-based method, where, given a training graph $G = (V, E)$, budget b and its solution set S, a node v is called *positive* if $v \in S$; otherwise it is negative. This approach, however, assumes all nodes that are not a part of S to be equally bad. In reality, this may not be the case. 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] We are, however, limited to set combinatorial problems only.

Probabilistic greedy: To address the above issue, we *sample* from the *solution space* in a greedy manner and learn embeddings that reflect the *marginal gain* $f(S \cup \{v\}) - f(S)$ provided by a node v towards the solution set S (Alg. 2 in Appendix). To sample from the solution space, 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:

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

Here, $gain_i(v)$ denotes the marginal gain contribution of v to S_i. Specifically, assume v is added to S_i in the $(j+1)_{th}$ iteration and let S_i^j be the set of nodes that were added in the first j iterations while constructing S_i. Then, $gain_i(v) = f\left(S_i^j \cup \{v\}\right) - f\left(S_i^j\right)$. In our experiments, m is set to 30 for all three problems of MCP, MVC and IM.

Termination condition of probabilistic greedy: Probabilistic greedy runs till *convergence* of the marginal gains, i.e., $gain_i(v) \leq \Delta$, where Δ is a small value. The goal here is to identify all nodes that could potentially be part of the solution set for *any* given budget. Δ in our experiments is set to 0.01 for all three problems of MCP, MVC and IM.

3.2 Training the GCN

Our goal in this phase is two-fold: **(1)** Identify nodes that are unlikely to be part of the solution set and are therefore noise in the context of our problem; **(2)** Learn a predictive model for node quality.

Noise predictor: The noise predictor should be lightweight so that expensive computations are reserved only for the good nodes. With this goal, we exploit the first layer information of the GCN and learn a classifier to predict for a given budget b, whether a node can be safely pruned without affecting the quality of the solution set. Typically, the first layer of a GCN contains the raw features of nodes that are relevant for the problem being solved. In GCOMB, we use the summation of the outgoing edge weights as node features. Let $\boldsymbol{x}_v$ denote the total outgoing edge weight of node v. To learn the noise predictor, given a set of training graphs $\{G_1, \cdots, G_t\}$, we first sort all nodes based on $\boldsymbol{x}_v$. Let $rank(v, G_i)$ denote the position of v in the sorted sequence based on $\boldsymbol{x}_v$ in G_i. Furthermore, let S_j^i denote the j^{th} solution set constructed by probabilistic greedy on G_i. Given a budget b, $S_{G_i,b}^j \subseteq S_j^i$ denotes the subset containing the first b nodes added to S_j^i by probabilistic greedy. Therefore, $r_{G_i}^b = \max_{j=0}^m \left\{\max_{\forall v \in S_{G_i,b}^j} \{rank(v, G_i)\}\right\}$ represents the lowest rank of any node in a solution set of budget b in G_i. This measure is further generalized to all training graphs in the form of $r_{max}^b = \max_{\forall G_i} \left\{r_{G_i}^b\right\}$, which represents the lowest rank of any node that has a realistic chance of being included in an answer set of budget b. To generalize across budgets, we compute $r_{max}^{b_i}$ for a series of budgets $\{b_1, \cdots, b_{max}\}$, where $b_{max} = \max_{\forall G_i} \left\{\max_{j=0}^m \left\{|S_j^i|\right\}\right\}$. On this data, we can perform curve fitting [1] to predict r_{max}^b for any (unseen) budget b. In our experiments, we use linear interpolation. To generalize across graph sizes, all of the above computations are performed on *normalized* budgets, where b is expressed in terms of the proportion of nodes with respect to the node set size of the graph. Similarly, rank $rank(v, G_i)$ is expressed in terms of percentile.

Node quality predictor: To train the GCN, we sample a training graph $G_i = (V_i, E_i)$ and a (normalized) budget b from the range $(0, b_{max}^i]$, where $b_{max}^i = \max_{j=0}^m \left\{\frac{|S_j^i|}{|V_i|}\right\}$. This tuple is sent to the noise predictor to obtain the good (non-noisy) nodes. The GCN parameters (Θ_G) are next learned by minimizing the loss function only on the good nodes. Specifically, for each good node v, we want to learn embeddings that can predict $score(v)$ through a surrogate function $score'(v)$. Towards that end, we draw multiple samples of training graphs and budgets, and the parameters are learned by minimizing the *mean squared error* loss (See Alg.3 for detailed pseudocode in the Supplementary).

$$J(\Theta_G) = \sum_{\sim \langle G_i, b \rangle} \frac{1}{|V_i^g|} \sum_{\forall v \in V_i^g} (score(v) - score'(v))^2 \tag{2}$$

In the above equation, V_i^g denotes the set of good nodes for budget b in graph G_i. Since GCNs are trained through message passing, in a GCN with K hidden layers, the computation graph is limited to the induced subgraph formed by the K-hop neighbors of V_i^g, instead of the entire graph.

3.3 Learning Q-function

While GCN captures the individual importance of a node, Q-learning [29] learns the combinatorial aspect in a budget-independent manner. Given a set of nodes S and a node $v \notin S$, we predict the n-step reward, $Q_n(S, v)$, for adding v to set S (action) via the surrogate function $Q'_n(S, v; \Theta_Q)$.

Defining the framework: We define the Q-learning task in terms of state space, action, reward, policy and termination with the input as a set of nodes and their predicted scores.

• **State space:** The state space characterizes the state of the system at any time step t in terms of the candidate nodes being considered, i.e., $C_t = V^g \setminus S_t$, with respect to the partially computed solution set S_t; V^g represents the set of good nodes from a training graph. In a combinatorial problem over nodes, two factors have a strong influence: **(1)** the individual quality of a node, and **(2)** its *locality*. The quality of a node v is captured through $score'(v)$. Locality is an important factor since two high-quality nodes from the same neighborhood may not be good collectively. The locality of a node $v \in C_t$ ($C_t = V^g \setminus S_t$) is defined as:

$$loc(v, S_t) = |N(v) \setminus \cup_{\forall u \in S_t} N(u)| \tag{3}$$

where $N(v) = \{v' \in V \mid (v, v') \in E\}$ are the neighbors of v. Note that $N(v)$ may contain noisy nodes since they contribute to the locality of $v \in V^g$. However, locality (and q-learning in general) is computed only on good nodes. The initial representation $\boldsymbol{\mu_v}$ of each node $v \in C_t$ is therefore the 2-dimensional vector $[score'(v), loc(v, S_t)]$. The representation of the set of nodes C_t is defined as $\boldsymbol{\mu_{C_t}} = \textsc{MaxPool}\{\mu_v \mid v \in C_t\}$. $\boldsymbol{\mu_{S_t}}$ is defined analogously as well. We use MAXPOOL since it captures the best available candidate node better than alternatives such as MEANPOOL. Empirically, we obtain better results as well.

• **Action and Reward:** An action corresponds to adding a node $v \in C_t$ to the solution set S_t. The immediate (0-step) reward of the action is its marginal gain, i.e. $r(S_t, v) = f(S_t \cup \{v\}) - f(S_t)$.

• **Policy and Termination:** The policy $\pi(v \mid S_t)$ selects the node with the highest *predicted* n-step reward, i.e., $\arg\max_{v \in C_t} Q'_n(S_t, v; \Theta_Q)$. We terminate after training the model for T samples.

Learning the parameter set Θ_Q: We partition Θ_Q into three weight matrices Θ_1, Θ_2, Θ_3, and one weight vector Θ_4 such that, $Q'_n(S_t, v; \Theta_Q) = \Theta_4 \cdot \boldsymbol{\mu_{C_t, S_t, v}}$, where $\boldsymbol{\mu_{C_t, S_t, v}} = \textsc{Concat}\left(\Theta_1 \cdot \boldsymbol{\mu}_{C_t}, \Theta_2 \cdot \boldsymbol{\mu}_{S_t}, \Theta_3 \cdot \boldsymbol{\mu}_v\right)$. If we want to encode the state space in a d-dimensional layer, the dimensions of the weight vectors are as follows: $\Theta_4 \in \mathbb{R}^{1 \times 3d}; \Theta_1, \Theta_2, \Theta_3 \in \mathbb{R}^{d \times 2}$. Q-learning updates parameters in a single episode via Adam optimizer[15] to minimize the squared loss.

$$J(\Theta_Q) = (y - Q'_n(S_t, v_t; \Theta_Q))^2, \text{ where } y = \gamma \cdot \max_{v \in V^g}\{Q'_n(S_{t+n}, v; \Theta_Q)\} + \sum_{i=0}^{n-1} r(S_{t+i}, v_{t+i})$$

γ is the *discount factor* and balances the importance of immediate reward with the predicted n-step future reward [29]. The pseudocode with more details is provided in the Supplementary (App. C).

3.3.1 Importance Sampling for Fast Locality Computation

Since degrees of nodes in real graphs may be very high, computing locality (Eq. 3) is expensive. Furthermore, locality is re-computed in each iteration. We negate this computational bottleneck through *importance sampling*. Let $N(V^g) = \{(v, u) \in E \mid v \in V^g\}$ be the neighbors of all nodes in V^g. Given a sample size z, we extract a subset $N_z(V^g) \subseteq N(V^g)$ of size z and compute locality only based on the nodes in $N_z(V^g)$. Importance sampling samples elements proportional to their importance. The *importance* of a node in $N(V^g)$ is defined as $I(v) = \frac{score'(v)}{\sum_{\forall v' \in N(V^g)} score'(v')}$.

Determining sample size: Let $\mu_{N(V^g)}$ be the mean importance of all nodes in $N(V^g)$ and $\hat{\mu}_{N_z(V^g)}$ the mean importance of sampled nodes. The sampling is *accurate* if $\mu_{N(V^g)} \approx \hat{\mu}_{N_z(V^g)}$.

Theorem 1 *Given an error bound ϵ, if sample size z is $O\left(\frac{\log |N(V^g)|}{\epsilon^2}\right)$, then $P\left[|\hat{\mu}_{N_z(V^g)} - \mu_{N(V^g)}| < \epsilon\right] > 1 - \frac{1}{|N(V^g)|^2}$.*

Remarks: **(1)** The sample size grows *logarithmically* with the neighborhood size, i.e., $|N(V^g)|$ and thus scalable to large graphs. **(2)** z is an inversely proportional function of the error bound ϵ.

3.4 Test Phase

Given an unseen graph G and budget b, we **(1)** identify the noisy nodes, **(2)** embed good nodes through a single forward pass through GCN, and **(3)** use GCN output to embed them and perform Q-learning to compute the final solution set.

Name	$\|V\|$	$\|E\|$
Brightkite (BK)	58.2K	214K
Twitter-ego (TW-ew)	81.3K	1.7M
Gowalla (GO)	196.5K	950.3K
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

(a) Datasets from SNAP repository [18].

Dataset	BP-500			Gowalla-900		
Budget	GCOMB	Greedy	Optimal	GCOMB	Greedy	Optimal
2	**0.295**	0.295	0.295	**0.75**	0.75	0.75
4	0.495	0.505	0.51	0.902	0.904	0.904
6	0.765	0.77	0.773	**0.941**	0.93	0.941
10	0.843	0.845	0.845	**0.952**	0.952	0.952
15	**0.96**	0.953	0.963	**0.963**	0.963	0.963
20	**0.998**	0.99	1	**0.974**	0.974	0.974
25	**1**	1	1	**0.985**	0.985	0.985
30	–	–	–	**0.996**	0.996	0.996
35	–	–	–	**1**	1	1

(b) Coverage in MCP

Table 1: In (b), the specific cases where GCOMB matches or outperforms Greedy are highlighted in bold. Gowalla-900 is a small subgraph of 900 nodes extracted from Gowalla (See App. I for details).

Complexity analysis: The time complexity of the test phase in GCOMB is $O\left(|V| + |V^{g,K}|\left(dm_G + m_G^2\right) + |V^g|b\left(d + m_Q\right)\right)$, where d is the average degree of a node, m_G and m_Q are the dimensions of the embeddings in GCN and Q-learning respectively, K is the number of layers in GCN, and $V^{g,K}$ represents the set of nodes within the K-hop neighborhood of V^g. The space complexity is $O(|V| + |E| + Km_G^2 + m_Q)$. The derivations are provided in App. D.

4 Empirical Evaluation

In this section, we benchmark GCOMB against GCN-TREESEARCH and S2V-DQN, and establish that GCOMB produces marginally improved quality, while being orders of magnitudes faster. The source code can be found at https://github.com/idea-iitd/GCOMB .

4.1 Experimental Setup

All experiments are performed on a machine running Intel Xeon E5-2698v4 processor with 64 cores, having 1 Nvidia 1080 Ti GPU card with 12GB GPU memory, and 256 GB RAM with Ubuntu 16.04. All experiments are repeated 5 times and we report the average of the metric being measured.

Datasets: Table 1a) lists the real datasets used for our experiments.
Random Bipartite Graphs (BP): We also use the synthetic random bipartite graphs from S2V-DQN [7]. In this model, 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. We use BP-X to denote a generated bipartite graph of X nodes.

Problem Instances: The performance of GCOMB is benchmarked on Influence Maximization (IM), Maximum Vertex Cover (MVC), and Maximum Coverage Problem (MCP) (§ 2). Since MVC can be mapped to MCP, empirical results on MVC are included in App. M.

Baselines: The performance of GCOMB is primarily compared with **(1)** GCN-TREESEARCH [19], which is the state-of-the-art technique to learn combinatorial algorithms. In addition, for MCP, we also compare the performance with **(2)** *Greedy* (Alg.1 in App. B), **(3)** S2V-DQN [7], **(5)** *CELF* [17] and **(6)** the *Optimal* solution set (obtained using CPLEX [12] on small datasets). Greedy and CELF guarantees a $1 - 1/e$ approximation for all three problems. We also compare with **(6)** *Stochastic Greedy(SG)* [21] in App. L. For the problem of IM, we also compare with the state-of-the-art algorithm **(7)** *IMM* [31]. Additionally, we also compare GCOMB with **(8)** *OPIM* [30]. For S2V-DQN, GCN-TREESEARCH, IMM, and OPIM we use the code shared by the authors.

Training: In all our experiments, for a fair comparison of GCOMB with S2V-DQN and GCN-TREESEARCH, we train all models for 12 hours and the best performing model on the validation set is used for inference. Nonetheless, we precisely measure the impact of training time in Fig. 2a. The break-up of time spent in each of the three training phases is shown in App. G in the Supplementary.

Parameters: The parameters used for GCOMB are outlined in App. H and their impact on performance is analyzed in App. N. For S2V-DQN and GCN-TREESEARCH, the best performing parameter values are identified using grid-search. In IMM, we set $\epsilon = 0.5$ as suggested by the authors. In OPIM, ϵ is recommended to be kept in range $[0.01, 0.1]$. Thus, we set it to $\epsilon = 0.05$.

4.2 Performance on Max Cover (MCP)

We evaluate the methods on both synthetic random bipartite (BP) graphs as well as real networks.
Train-Validation-Test split: While testing on any synthetic BP graph, we train and validate on five

Graph	S2V-DQN	GCN-TS	GCOMB	Greedy
BP-2k	0.87	0.86	**0.89**	0.89
BP-5k	0.85	0.84	**0.86**	0.86
BP-10k	0.84	0.83	**0.85**	0.85
BP-20k	NA	0.82	**0.83**	0.83

(a) Coverage achieved in MCP at $b = 15$.

Budget	Speed-up
20	4
50	3.9
100	3.01
150	2.11
200	2.01

(b) Speed-up against CELF in MCP on YT.

Table 2: (a) Coverage on Random Graphs in MCP. (b) Speed-up achieved by GCOMB against CELF on YT in MCP.

BP-1k graphs each. For real graphs, we train and validate on BrightKite (BK) ($50:50$ split for train and validate) and test on other real networks. Since our real graphs are not bipartite, we convert it to one by making two copies of V: V_1 and V_2. We add an edge from $u \in V_1$ to $u' \in V_2$ if $(u, u') \in E$.

Comparison with Greedy and Optimal: Table 1b presents the achieved coverage (Recall § 2 for definition of coverage). We note that Greedy provides an empirical approximation ratio of at least 99% when compared to the optimal. This indicates that in larger datasets where we are unable to compute the optimal, Greedy can be assumed to be sufficiently close to the optimal. Second, GCOMB is sometimes able to perform even better than greedy. This indicates that Q-learning is able to learn a more generalized policy through *delayed* rewards and avoid a myopic view of the solution space.

Synthetic Datasets: Table 2a presents the results. GCOMB and Greedy achieves the highest coverage consistently. While S2V-DQN performs marginally better than GCN-TREESEARCH, S2V-DQN is the least scalable among all techniques; it runs out of memory on graphs containing more than $20,000$ nodes. As discussed in details in § 1.2, the non-scalability of S2V-DQN stems from relying on an architecture with significantly larger parameter set than GCOMB or GCN-TREESEARCH. In contrast, GCOMB avoids noisy nodes, and focuses the search operation only on the good nodes.

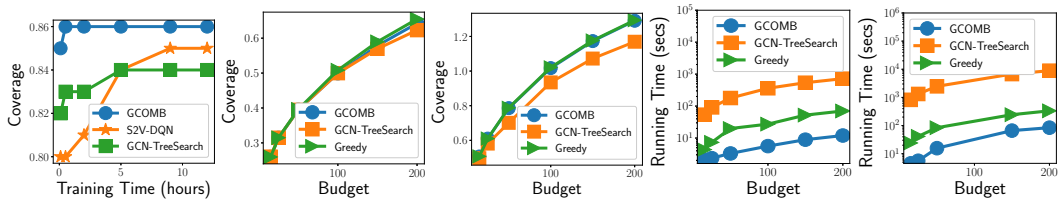
Impact of training time: A complex model with more number of parameters results in slower learning. In Fig. 2a, we measure the coverage against the training time. While GCOMB's performance saturates within 10 minutes, S2V-DQN and GCN-TREESEARCH need 9 and 5 hours respectively for training to obtain its best performance.

Real Datasets: Figs. 2b and 2c present the achieved Coverage as the budget is varied. GCOMB achieves similar quality as Greedy, while GCN-TREESEARCH is marginally inferior. The real impact of GCOMB is highlighted in Figs. 2d and 2e, which shows that GCOMB is up to 2 orders of magnitude faster than GCN-TREESEARCH and 10 times faster than Greedy. Similar conclusion can also be drawn from the results on Gowalla dataset in App. K in Supplementary.

Comparison with CELF: Table 2b presents the speed-up achieved by GCOMB against CELF. The first pass of CELF involves sorting the nodes, which has complexity $O(|V|log|V|)$. On the other hand, no such sorting is required in GCOMB. Thus, the speed-up achieved is higher in smaller budgets.

4.3 Performance on Influence Maximization

Influence Maximization (IM) is the hardest of the three combinatorial problems since estimating the spread of a node is #P-hard [14].



(a) Bipartite graph (5k) (b) Twitter-ego (c) YouTube (d) Twitter-ego (e) YouTube

Figure 2: MCP: (a) Improvement in Coverage against training time at $b = 15$. (b-c) Coverage achieved by GCOMB, GCN-TREESEARCH and Greedy. (d-e) Running times in TW-ew and YT.

Edge weights: We assign edge weights that denote the influence of a connection using the two popular models [2]: **(1) Constant (CO:)** All edge weights are set to 0.1, **(2) Tri-valency (TV):** Edge weights are sampled randomly from the set $\{0.1, 0.01, 0.001\}$. In addition, we also employ a third **(3) Learned (LND)** model, where we *learn* the influence probabilities from the action logs of users. This is only applicable to the Stack data which contain *action logs* from $8/2008$ to $3/2016$. We define the influence of u on v as the probability of v interacting with u's content at least once in a month.

Train-Validation-Test split: In all of the subsequent experiments, for CO and TV edge weight models, we train and validate on a subgraph sampled out of YT by randomly selecting 30% of the edges (50% of this subset is used for training and 50% is used for validation). For LND edge weight models, we train and validate on the subgraph induced by the 30% of the earliest edges from Stack in terms of temporal order. While testing, on YT and Stack, we use the graph formed by the remaining 70% of the edges that are not used for training. On other datasets, we use the entire graph for testing since neither those datasets nor their subsets are used for training purposes.

GCOMB vs.GCN-TREESEARCH: Fig. 3a compares the running time in IM on progressively larger subgraphs extracted from YT. While GCN-TREESEARCH consumes ≈ 3 hours on the 70% sub-graph, GCOMB finishes in 5 seconds.

GCOMB vs. NOISEPRUNER+CELF NOISEPRUNER+CELF, i.e., running CELF only on non-noisy nodes, is orders of magnitude slower than GCOMB in IM (See Fig 3d). Pruning noisy nodes does not reduce the graph size; it only reduces the number of candidate nodes. To compute expected spread in IM, we still require the entire graph, resulting in non-scalability.

Billion-sized graphs: IMM crashes on both the billion-sized datasets of TW and FS, as well as Orkut. Unsurprisingly, similar results have been reported in [2]. IMM strategically samples a subgraph of the entire graph based on the edge weights. On this sampled subgraph, it estimates the influence of a node using *reverse reachability sets*. On large graphs, the sample size exceeds the RAM capacity of 256GB. Hence, it crashes. In contrast, GCOMB finishes within minutes for smaller budgets ($b < 30$) and within 100 minutes on larger budgets of 100 and 200 (Figs. 3g-3h). This massive scalability of GCOMB is a result of low storage overhead (only the graph and GCN and Q-learning parameters; detailed Space complexity provided in App. D in the Supplementary) and relying on just forwarded passes through GCN and Q-learning. The speed-up with respect to OPIM on billion-sized graphs can be seen in App. J.

Performance on YT and Stack: Since IMM crashes on Orkut, TW, and FS, we compare the quality of GCOMB with IMM on YT and Stack. Table 3a reports the results in terms of *spread difference*, where Spread Difference $= \frac{f(S_{IMM}) - f(S_{\text{GCOMB}})}{f(S_{IMM})} \times 100$. S_{IMM} and S_{GCOMB} are answer sets computed by IMM and GCOMB respectively. A negative spread difference indicates better performance by GCOMB. The expected spread of a given set of nodes S, i.e. $f(S)$, is computed by taking the average spread across $10,000$ Monte Carlo simulations.

Table 3a shows that the expected spread obtained by both techniques are extremely close. The true impact of GCOMB is realized when Table 3a is considered in conjunction with Figs. 3b-3c, which shows GCOMB is 30 to 160 times faster than IMM. In this plot, speed-up is measured as $\frac{time_{IMM}}{time_{\text{GCOMB}}}$ where $time_{IMM}$ and $time_{\text{GCOMB}}$ are the running times of IMM and GCOMB respectively.

Similar behavior is observed when compared against OPIM as seen in Table 3b and Figs. 3e- 3f.

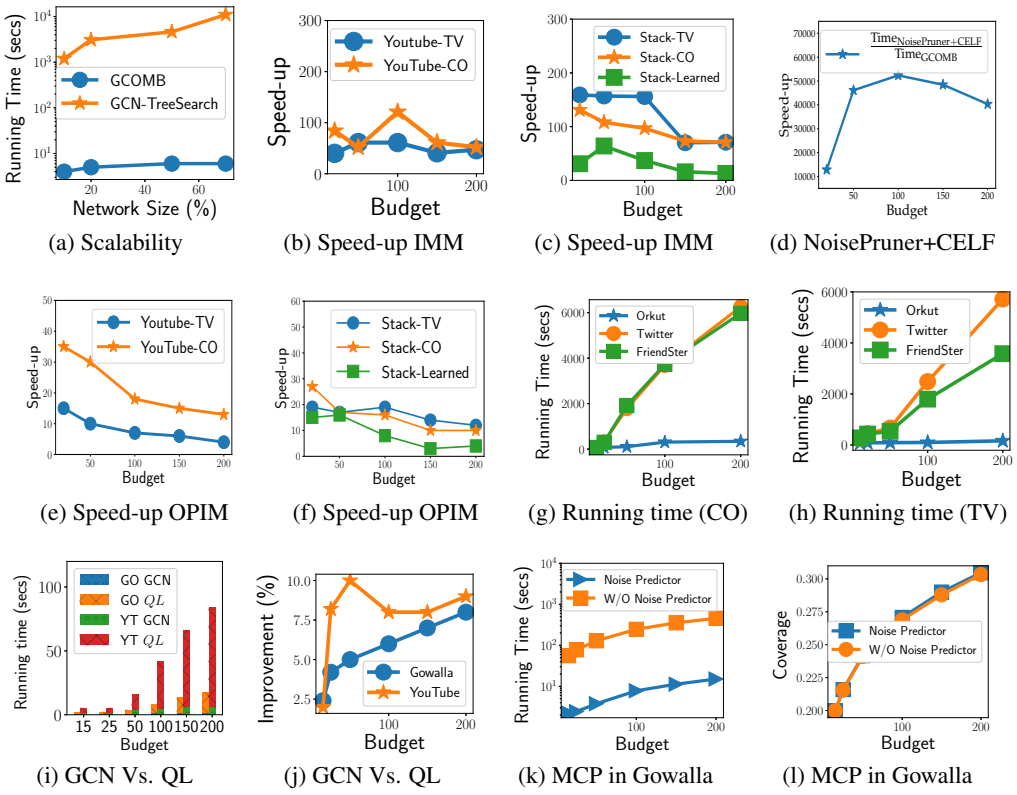
b	YT-TV	YT-CO	Stack-TV	Stack-CO	Stack-LND
10	$\mathbf{-1 \times 10^{-3}}$	1×10^{-4}	2×10^{-5}	≈ 0	1×10^{-5}
20	$\mathbf{-2 \times 10^{-3}}$	2×10^{-4}	3×10^{-5}	3×10^{-5}	$\mathbf{-7 \times 10^{-5}}$
50	$\mathbf{-3 \times 10^{-3}}$	$\mathbf{-5 \times 10^{-5}}$	2×10^{-5}	6×10^{-5}	$\mathbf{-7 \times 10^{-5}}$
100	$\mathbf{-1 \times 10^{-3}}$	6×10^{-4}	2×10^{-4}	2×10^{-4}	$\mathbf{-1 \times 10^{-4}}$
150	$\mathbf{-6 \times 10^{-4}}$	3×10^{-4}	1×10^{-4}	1×10^{-4}	$\mathbf{-3 \times 10^{-5}}$
200	$\mathbf{-2 \times 10^{-3}}$	2×10^{-5}	2×10^{-4}	2×10^{-4}	$\mathbf{-1 \times 10^{-4}}$

(a) Spread difference between IMM and GCOMB.

b	YT-TV	YT-CO	Stack-TV	Stack-CO	Stack-LND
10	$\mathbf{-5 \times 10^{-5}}$	$\mathbf{-1 \times 10^{-5}}$	2×10^{-5}	≈ 0	1×10^{-5}
20	$\mathbf{-1 \times 10^{-4}}$	1×10^{-5}	3×10^{-5}	2×10^{-5}	$\mathbf{-2 \times 10^{-5}}$
50	$\mathbf{-2 \times 10^{-4}}$	$\mathbf{-3 \times 10^{-5}}$	2×10^{-5}	5×10^{-5}	$\mathbf{-6 \times 10^{-4}}$
100	$\mathbf{-3 \times 10^{-4}}$	2×10^{-5}	1×10^{-4}	7×10^{-5}	$\mathbf{-2 \times 10^{-4}}$
150	$\mathbf{-3 \times 10^{-4}}$	$\mathbf{-2 \times 10^{-5}}$	1×10^{-4}	1×10^{-4}	$\mathbf{-3 \times 10^{-4}}$
200	$\mathbf{-4 \times 10^{-4}}$	$\mathbf{-7 \times 10^{-5}}$	2×10^{-4}	2×10^{-4}	$\mathbf{-3 \times 10^{-4}}$

(b) Spread difference between OPIM and GCOMB.

Table 3: Comparison with respect to (a) IMM and (b) OPIM on YT and Stack. A *negative* value, highlighted in bold, indicates *better performance by* GCOMB.



(a) Scalability (b) Speed-up IMM (c) Speed-up IMM (d) NoisePruner+CELF

(e) Speed-up OPIM (f) Speed-up OPIM (g) Running time (CO) (h) Running time (TV)

(i) GCN Vs. QL (j) GCN Vs. QL (k) MCP in Gowalla (l) MCP in Gowalla

Figure 3: (a) Comparison of running time between GCOMB and GCN-TREESEARCH in YT at $b = 20$. (b-c) Speed-up achieved by GCOMB over IMM. (d) Speed-up achieved by GCOMB over NoisePruner+CELF on IM. (e-f) Speed-up achieved by GCOMB over OPIM. (g-h) Running times of GCOMB in IM in large graphs under the CO and TV edge models. (i) Distributions of running time between GCN and Q-learning in GO and YT datasets for MCP. (j) Improvement of Q-learning over GCN in MCP. (k-l) Impact of noise predictor on (k) running time and (l) quality.

4.4 Design Choices

Impact of Q-learning: Since GCN predicts the expected marginal gain of a node, why not simply select the top-b nodes with the highest predicted marginal gains for the given budget b? This is a pertinent question since, as visible in Fig. 3i, majority of the time in GCOMB is spent on Q-learning. Fig. 3j shows that Q-learning imparts an additional coverage of up to 10%. Improvement $(\%)$ is quantified as $\frac{Coverage_{\text{GCOMB}} - Coverage_{GCN}}{Coverage_{GCN}} \times 100$.

Impact of Noise Predictor: Fig. 3k presents the impact of noise predictor which is close to two orders of magnitude reduction in running time. This improvement, however, does not come at the cost of efficacy (Fig. 3l). In fact, the quality improves slightly due to the removal of noisy nodes.

5 Conclusion

S2V-DQN [7] initiated the promising direction of learning combinatorial algorithms on graphs. GCN-TREESEARCH [19] pursued the same line of work and enhanced scalability to larger graphs. However, the barrier to million and billion-sized graphs remained. GCOMB removes this barrier with a new lightweight architecture. In particular, GCOMB uses a phase-wise mixture of supervised and reinforcement learning. While the supervised component predicts individual node qualities and prunes those that are unlikely to be part of the solution set, the Q-learning architecture carefully analyzes the remaining high-quality nodes to identify those that collectively form a good solution set. This architecture allows GCOMB to generalize to unseen graphs of significantly larger sizes and convincingly outperform the state of the art in efficiency and efficacy. Nonetheless, there is scope for improvement. GCOMB is limited to set combinatorial problems on graphs. In future, we will explore a bigger class of combinatorial algorithms such as sequential and capacity constrained problems.

Broader Impact

The need to solve NP-hard combinatorial problems on graphs routinely arise in several real-world problems. Examples include facility location problems on road networks [20], strategies to combat rumor propagation in online social networks [3], computational sustainability [8] and health-care [33]. Each of these problems plays an important role in our society. Consequently, designing effective and efficient solutions are important, and our current work is a step in that direction. The major impact of this paper is that good heuristics for NP-hard problems can be learned for large-scale data. While we are not the first to observe that heuristics for combinatorial algorithms can be learned, we are the first to make them scale to billion-size graphs, thereby bringing an algorithmic idea to practical use-cases.

Acknowledgments and Disclosure of Funding

The project was partially supported by the National Science Foundation under award IIS-1817046. Further, Sahil Manchanda acknowledges the financial support from the Ministry of Human Resource Development (MHRD) of India and the Department of Computer Science and Engineering, IIT Delhi.

References

[1] Sandra L Arlinghaus. *Practical Handbook of Curve Fitting*. CRC press, 1994.

[2] Akhil Arora, Sainyam Galhotra, and Sayan Ranu. Debunking the myths of influence maximization: An in-depth benchmarking study. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 651–666, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Ceren Budak, Divyakant Agrawal, and Amr El Abbadi. Limiting the spread of misinformation in social networks. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, page 665–674, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Vineet Chaoji, Sayan Ranu, Rajeev Rastogi, and Rushi Bhatt. Recommendations to boost content spread in social networks. In *Proceedings of the 21st international conference on World Wide Web*, pages 529–538, 2012.

[5] Wei Chen, Chi Wang, and Yajun Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1029–1038, 2010.

[6] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F Werneck. Sketch-based influence maximization and computation: Scaling up with guarantees. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 629–638, 2014.

[7] Hanjun Dai, Elias Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

[8] Bistra Dilkina, Katherine J. Lai, and Carla P. Gomes. Upgrading shortest paths in networks. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 76–91. Springer, 2011.

[9] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 15554–15566, 2019.

[10] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[11] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

[12] IBM. Cplex 12.9, 2019.

[13] Kyomin Jung, Wooram Heo, and Wei Chen. Irie: Scalable and robust influence maximization in social networks. In *ICDM*, pages 918–923. IEEE, 2012.

[14] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *KDD*, 2003.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[16] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. Cost-effective outbreak detection in networks. In *KDD*, pages 420–429, 2007.

[17] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 420–429, 2007.

[18] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2020.

[19] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *NIPS*, pages 537–546, 2018.

[20] Sourav Medya, Jithin Vachery, Sayan Ranu, and Ambuj Singh. Noticeable network delay minimization via node upgrades. *Proceedings of the VLDB Endowment*, 11(9):988–1001, 2018.

[21] Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, Amin Karbasi, Jan Vondrák, and Andreas Krause. Lazier than lazy greedy. *arXiv preprint arXiv:1409.7938*, 2014.

[22] Shubhadip Mitra, Sayan Ranu, Vinay Kolar, Aditya Telang, Arnab Bhattacharya, Ravi Kokku, and Sriram Raghavan. Trajectory aware macro-cell planning for mobile users. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 792–800. IEEE, 2015.

[23] Shubhadip Mitra, Priya Saraf, Richa Sharma, Arnab Bhattacharya, and Sayan Ranu. Netclus: A scalable framework to mine top-k locations for placement of trajectory-aware services. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, pages 27–35, 2019.

[24] Shubhadip Mitra, Priya Saraf, Richa Sharma, Arnab Bhattacharya, Sayan Ranuy, and Harsh Bhandari. Netclus: A scalable framework for locating top-k sites for placement of trajectory-aware services. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 87–90. IEEE, 2017.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[26] Naoto Ohsaka, Takuya Akiba, Yuichi Yoshida, and Ken-ichi Kawarabayashi. Fast and accurate influence maximization on large networks with pruned monte-carlo simulations. In *AAAI*, pages 138–144, 2014.

[27] Marcelo Prates, Pedro HC Avelar, Henrique Lemos, Luis C Lamb, and Moshe Y Vardi. Learning to solve np-complete problems: A graph neural network for decision tsp. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4731–4738, 2019.

[28] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *ECML*, pages 317–328. Springer, 2005.

[29] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[30] Jing Tang, Xueyan Tang, Xiaokui Xiao, and Junsong Yuan. Online processing algorithms for influence maximization. In *Proceedings of the 2018 International Conference on Management of Data*, pages 991–1005, 2018.

[31] Youze Tang, Yanchen Shi, and Xiaokui Xiao. Influence maximization in near-linear time: A martingale approach. In *SIGMOD*, pages 1539–1554, 2015.

[32] Chi Wang, Wei Chen, and Yajun Wang. Scalable influence maximization for independent cascade model in large-scale social networks. *Data Mining and Knowledge Discovery*, 25(3):545–576, 2012.

[33] Bryan Wilder, Han Ching Ou, Kayla de la Haye, and Milind Tambe. Optimizing network structure for preventative health. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 841–849, 2018.

6 Appendix

A Influence Maximization

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.*

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.

Definition 4 (Independent Cascade [14]) *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, $\Gamma(S) = \frac{|V_a|}{|V|}$.*

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

Influence Maximization (IM) Problem [14]: Given a budget b, a social network G, and a information diffusion model $\mathcal{M}$, select a set S^* of b nodes such that the expected diffusion spread $f(S^*) = \mathbb{E}[\Gamma(S^*)]$ is maximized.

B The greedy approach

Greedy provides an $1 - \frac{1}{e}$-approximation for all three NP-hard problems of MCP, MVC, and IM[14]. Algorithm 1 presents the pseudocode. The input to the algorithm is a graph $G = (V, E)$, an optimization function $f(S)$ and the budget b. Starting from an empty solution set S, Algorithm 1 iteratively builds the solution by adding the "best" node to S in each iteration (lines 3-5). The best node $v^* \in V \backslash S$ is the one that provides the highest *marginal gain* on the optimization function (line 4). The process ends after b iterations.

Limitations of greedy: Greedy itself has scalability challenges depending on the nature of the problem. Specifically, in Alg. 1 there are two expensive computations. First, computing the optimization function $f(\cdot)$ itself may be expensive. For instance, computing the expected spread in IM is $\#P$-hard [2]. Second, even if $f(\cdot)$ is efficiently computable, computing the marginal gain is often expensive. To elaborate, in MCP, computing the marginal gain involves a setminus operation on the neighborhood lists of all nodes $v \notin S$ with the neighborhood of all nodes $u \in S$, where S is the set of solution nodes till now. Each setminus operation consumes $O(d)$ time where d is the average degree of nodes, resulting in a total complexity of $O(bd|V|)$. In IM, the cost is even higher with a complexity of $O(b|V|^2)$. In GCOMB, we overcome these scalability bottlenecks without compromising on the quality. GCOMB utilizes GCN [10] to solve the first bottleneck of predicting $f(\cdot)$. Next, a deep Q-learning network is designed to estimate marginal gains efficiently. With this unique combination, GCOMB can scale to billion-sized graphs.

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

Algorithm 2 The probabilistic greedy approach

Require: $G = (V, E)$, optimization function $f(.)$, convergence threshold Δ
Ensure: solution set S, $|S| = b$
1: $S \leftarrow \emptyset$
2: **while** $(gain > \Delta)$ **do**
3: $\quad v \leftarrow$ Choose with probability $\frac{f(S\cup\{v\})-f(S)}{\sum_{\forall v' \in V \setminus S} f(S\cup\{v'\})-f(S)}$
4: $\quad gain \leftarrow f(S \cup \{v\}) - f(S)$
5: $\quad S \leftarrow S \cup \{v\}$
6: **Return** S

B.1 Training the GCN:

For each node v, and its $score(v)$, which is generated using probabilistic greedy algorithm, we learn embeddings to predict this score via a Graph Convolutional Network (GCN) [10]. The pseudocode for this component is provided in Alg. 3.

From the given set of training graphs $\{G_1, \cdots, G_t\}$, we sample a graph G_i and a normalized budget b from the range of budgets $(0, b^i_{max}]$, where $b^i_{max} = \max^m_{j=0}\left\{\frac{|S^i_j|}{|V_i|}\right\}$. To recall, S^i_j denotes the j^{th} solution set constructed by probabilistic greedy on graph G_i. Further, quantity r^b_{max} is computed from the set of training graphs and their probabilistic greedy solutions as described in § 3.2. It is used to determine the nodes which are non-noisy for the budget b.

For a sampled training graph G_i and budget b, only those nodes that have a realistic chance of being in the solution set are used to train the GCN (line 2). Each iteration in the outer loop represents the *depth* (line 4). In the inner loop, we iterate over all nodes which are non-noisy and in their K-hop neighborhood (line 5). While iterating over node v, we fetch the current representations of v's neighbors and *aggregate* them through a MEANPOOL layer (lines 6-7). Specifically, for dimension i, we have: $\mathbf{h}^k_N(v)_i = \frac{1}{|N(v)|}\sum_{\forall u \in N(v)} h^{k-1}_{u_i}$. 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 8), 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 $h^0_v = \mathbf{x}_v$, while the final embedding is $\boldsymbol{\mu}_v = h^K_v$ (line 9). In hidden layers, Alg. 3 requires the parameter set $\mathbb{W} = \{\mathbb{W}^k, k = 1, 2, \cdots, K\}$ to compute the node representations (line 8). Intuitively, $\mathbb{W}^k$ is used to propagate information across different depths of the model. 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 $score'(v)$ (line 10). Further, the inclusion of 1-hop neighbors($V^{g,1}$) of V^g in line 9 and line 10 is only for the importance sampling procedure. The parameters Θ_G for the proposed framework are therefore the weight matrices $\mathbb{W}$ and the weight vector $\mathbf{w}$. We draw multiple samples of graphs and budget and minimize the next equation using Adam optimizer [15] to learn the GCN parameters, Θ_G.

$$J(\Theta_G) = \sum_{\sim\langle G_i, b\rangle} \frac{1}{|V^g_i|} \sum_{\forall v \in V^g_i} (score(v) - score'(v))^2 \tag{4}$$

Algorithm 3 Graph Convolutional Network (GCN)

Require: $G = (V, E)$, $\{score(v)$, input features $\mathbf{x}_v \; \forall v \in V\}$, budget b, noisy-node cut off r^b_{max}, depth K, weight matrices $\mathbb{W}^k$, $\forall k \in [1, k]$ and weight vector $\mathbf{w}$, dimension size m_G.
Ensure: Quality score $score'(v)$ for good nodes and nodes in their 1-hop neighbors

$\mathbf{h}^0_v \leftarrow \mathbf{x}_v, \; \forall v \in V$
$V^g \leftarrow \{v \in V \mid rank(v, G) < r^b_{max}\}$
$V^{g,K} \leftarrow$ K-hop neighborhood of V^g
for $k \in [1, K]$ **do**
  for $v \in V^g \cup V^{g,K}$ **do**
    $N(v) \leftarrow \{u | (v, u) \in E\}$
    $\mathbf{h}^k_N(v) \leftarrow \textsc{MeanPool}\left(\left\{h^{k-1}_u, \forall u \in N(v)\right\}\right)$
    $\mathbf{h}^k_v \leftarrow ReLU\left(\mathbb{W}^k \cdot \textsc{Concat}\left(\mathbf{h}^k_N(v), h^{k-1}_v\right)\right)$
$\boldsymbol{\mu}_v \leftarrow \mathbf{h}^K_v, \; \forall v \in V^g \cup V^{g,1}$
$score'(v) \leftarrow \mathbf{w}^T \cdot \boldsymbol{\mu}_v \, , \forall v \in V^g \cup V^{g,1}$

In the above equation, V^g_i denotes the set of good nodes for budget b in graph G_i.

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.

C Q-learning

The pseudocode of the Q-learning component is provided in Algorithm 4.

Exploration vs. Exploitation: In the initial phases of the training procedure, the prediction may be inaccurate as the model has not yet received enough training data to learn the parameters. Thus, with $\epsilon = \max\{0.05, 0.9^t\}$ probability we select a random node from C_t. Otherwise, we trust the model and choose the predicted best node. Since ϵ decays exponentially with t, as more training samples are observed, the likelihood to trust the prediction goes up. This policy is commonly used in practice and inspired from bandit learning [7].

n-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 in Alg. 4). The key idea here is to wait for n steps before the approximator's parameters are updated and therefore, more accurately estimate future rewards.

Fitted Q-learning: For efficient learning of the parameters, we perform *fitted Q-iteration* [28], which results in faster convergence using a neural network as a function approximator [25]. Specifically, instead of updating the Q-function sample-by-sample, fitted Q-iteration uses *experience replay* with a batch of samples. Note that the training process in Alg. 4 is independent of budget. The Q-learning component learns the best action to take under a given circumstance (state space).

D Complexity Analysis of the Test Phase

For this analysis, we assume the following terminologies. d denotes the average degree of a node. m_G and m_Q denote the embedding dimensions in the GCN and Q-learning neural network respectively. As already introduced earlier, b denotes the budget and V is the set of all nodes.

D.1 Time Complexity

In the test phase, a forward pass through the GCN is performed. Although the GCN's loss function only minimizes the prediction with respect to the good nodes, due to message passing from neighbors, in a K-layered GCN , we need the K-hop neighbors of the good nodes (we will denote this set as $V^{g,K}$). Each node in $V^{g,K}$ draws messages from its neighbors on which first we perform MEANPOOL and then dot products are computed to embed in a m_G-dimensional space. Applying MEANPOOL consumes $O(dm_G)$ time since we need to make a linear pass over d vectors of m_G dimensions. Next, we perform m_G dot-products on vectors of m_G dimensions. Consequently, this consumes $O(m^2_G)$ time. Finally, this operation is repeated in each of the K layers of the GCN. Since K is typically 1 or 2, we ignore this factor. Thus, the total time complexity of a forward pass is $O(|V^{g,K}|(dm_G + m^2_G))$.

Algorithm 4 Learning Q-function

Require: $\forall v \in V^g$, $score'(v)$, hyper-parameters M, N relayed to fitted Q-learning, number of episodes L and sample size T.
Ensure: Learn parameter set Θ_Q

```
Initialize experience replay memory M to capacity N
for episode e ← 1 to L do
    for step t ← 1 to T do
        v_t ← { random node v ∉ S_t with probability ε = max{0.05, 0.9^t}
              { argmax_{v∉S_t} Q'_n(S_t, v, Θ_Q) otherwise
        S_{t+1} ← S_t ∪ {v_t}
        if t ≥ n then
            Add tuple (S_{t−n}, v_{t−n}, Σ_{i=t−n}^{t} r(S_i, v_i), S_t) to M
            Sample random batch B from M
            Update Θ_Q by Adam optimizer for B
return Θ_Q
```

The budget (b) number of forward passes are made in the Q-learning component over only V^g (the set of good non-noisy nodes). In each pass, we compute locality and the predicted reward. To compute locality, we store the neighborhood as a hashmap, which consumes $O(d)$ time per node. Computing predicted reward involves dot products among vectors of $O(m_Q)$ dimensions. Thus, the total time complexity of the Q-learning component is $O(|V^g|b(d + m_Q))$.

For noise predictor, we need to identify the top-l nodes based on $\mathbf{x_v}$ (typically the out-degree weight). l is determined by the noise predictor as a function of b. This consumes $|V|log(l)$ time through the use of a min-Heap.

Combining all three components, the total time complexity of GCOMB is $O(|V|log(l) + |V^{g,K}|(dm_G + m_G^2) + |V^g|b(d + m_Q))$. Typically, $l << |V|$ (See Fig. 8a) and may be ignored. Thus, the total time complexity is $\approx O(|V| + |V^{g,K}|(dm_G + m_G^2) + |V^g|b(d + m_Q))$.

D.2 Space Complexity

During testing, the entire graph is loaded in memory and is represented in linked list form which takes $O(|V| + |E|)$ space. The memory required for K layer GCN is $O(Km_G^2)$. Overall space complexity for GCN phase is $O(|V| + |E| + Km_G^2)$.

For the Q-learning component, entire graph is required for importance sampling purpose. It requires $O(|V| + |E|)$ space. Further, the space required for parameters for Q-network is $O(m_Q)$, since input dimension for Q-network is fixed to 2. Thus, space complexity of Q-network is $O(|V| + |E| + m_Q)$. Therefore, total space complexity of GCOMB is $O(|V| + |E| + Km_G^2 + m_Q)$.

E Number of parameters

GCN: If m_G is the embedding dimension, each $\mathbb{W}_k$ is a matrix of dimension m_G^2. Other than $\mathbb{W}_k$, we learn another parameter $\boldsymbol{w}$ in the final layer (line 10 of Alg. 3) of m_G dimension. Thus, the total parameter size is $K \times m_G^2 + m_G$, where K is the number of layers in GCN.

Q-learning: If m_Q is the dimension of the hidden layer in Q-learning, each of Θ_1, Θ_2, and Θ_3 is a matrix of dimension $m_Q \times 2$. Θ_4 is a vector of dimension $3m_Q$. Thus, the total number of parameters is $9m_Q$.

F Proof of Theorem 1

A sampling procedure is *unbiased* if it is possible to estimate the mean of the target population from the sampled population, i.e., $\mathbb{E}[\hat{\mu}_{N_z(V^g)}] = \mu_{N(V^g)} = \frac{\sum_{v \in N(V^g)} I(v)}{|N(V^g)|} = \frac{1}{|N(V^g)|}$, where $\hat{\mu}_{N_z(V^g)}$ is the *weighted average* over the samples in $N_z(V^g)$. Specifically,

$$\hat{\mu}(N_z(V^g)) = \frac{1}{\sum_{v \in N_z(V^g)} \hat{w}_v} \sum_{v \in N_z(V^g)} \hat{w}_v \cdot I(v) \tag{5}$$

where $\hat{w}_v = \frac{1}{I(v)}$.

Lemma 1 *Importance sampling is an unbiased estimate of $\mu_{N(V^g)}$, i.e., $\mathbb{E}\left[\hat{\mu}_{N_z(V^g)}\right] = \mu_{N(V^g)}$, if $\hat{w}_v = \frac{1}{I(v)}$.*

Proof 1

$$\mathbb{E}[\hat{\mu}_{N_z(V^g)}] = \frac{1}{\mathbb{E}[\sum_{v \in N_z(V^g)} \hat{w}_v]} \cdot \mathbb{E}\left[\sum_{v \in N_z(V^g)} \hat{w}_v \cdot I(v)\right]$$

If we simplify the first term, we obtain

$$\mathbb{E}\left[\sum_{v \in N_z(V^g)} \hat{w}_v\right] = z \times \mathbb{E}[\hat{w}_v]$$
$$= z \times \sum_{\forall v \in N(V^g)} \hat{w}_v \cdot I(v) = |N(V^g)| \times z$$

From the second term, we get,

$$\mathbb{E}\left[\sum_{v \in N_z(V^g)} \hat{w}_v \cdot I(v)\right] = z \times \mathbb{E}[\hat{w}_v \cdot I(v)] = z$$

Combining these two, $\mathbb{E}[\hat{\mu}_{N_z(V^g)}] = \frac{z}{|N(V^g)| \times z} = \mu_{N(V^g)}$.

Armed with an unbiased estimator, we show that a bounded number of samples provide an accurate estimation of the locality of a node.

Lemma 2 ***[Theorem 1 in main draft]*** *Given ϵ as the error bound, $P\left[|\hat{\mu}_{N_z(V^g)} - \mu_{N(V^g)}| < \epsilon\right] > 1 - \frac{1}{|N(V^g)|^2}$, where z is $O\left(\frac{\log |N(V^g)|}{\epsilon^2}\right)$.*

Proof 2 *The samples can be viewed as random variables associated with the selection of a node. More specifically, the random variable, X_i, is the importance associated with the selection of the i-th node in the importance sample $N_z(V^g)$. Since the samples provide an unbiased estimate (Lemma 1) and are i.i.d., we can apply* Hoeffding's inequality *[11] to bound the error of the mean estimates:*

$$P\left[|\hat{\mu}_{N_z(V^g)} - \mu_{N(V^g)}| \geq \epsilon\right] \leq \delta$$

where $\delta = 2\exp\left(-\frac{2z^2\epsilon^2}{\mathcal{T}}\right)$, $\mathcal{T} = \sum_{i=1}^{z}(b_i - a_i)^2$, and each X_i is strictly bounded by the intervals $[a_i, b_i]$. Since we know that importance is bounded within $[0, 1]$, $[a_i, b_i] = [0, 1]$. Thus,

$$\delta = 2\exp\left(-\frac{2z^2\epsilon^2}{z}\right) = 2\exp\left(-2z\epsilon^2\right)$$

By setting the number of samples $z = \frac{\log(2|N(V^g)|^2)}{2\epsilon^2}$, we have,

$$p\left[|\hat{\mu}_{N_z(V^g)} - \mu_{N(V^g)}| < \epsilon\right] > 1 - \frac{1}{|N(V^g)|^2}$$

G Training time distribution of different phases of GCOMB

Fig. 4 shows the distribution of time spent in different phases of training of GCOMB. Prob-Greedy refers to the phase in which probabilistic greedy algorithm is run on training graphs to obtain training labels for GCN component. Train-GCN and Train-QL refers to the training phases of GCN and Q-network respectively.

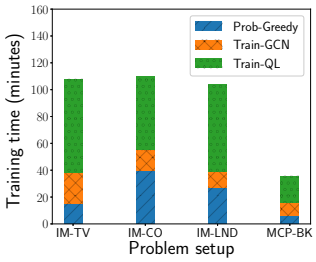
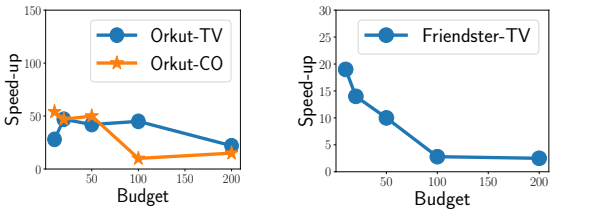


Figure 4: Phase-wise training time distribution of GCOMB



(a) Speed-up OPIM Orkut (b) Speed-up OPIM Friendster

Figure 5: Speed up obtained by GCOMB over OPIM

H Parameters

GCOMB has two components: GCN and the Q-Learning part. GCN is trained for 1000 epochs with a learning rate of 0.001, a dropout rate of 0.1 and a convolution depth (K) of 2. The embedding dimension is set to 60. For training the n-step Q-Learning neural network, n and discount factor γ are set to 2 and 0.8 respectively, and a learning rate of 0.0005 is used. The raw feature x_v of node v in the first layer of GCN is set to the summation of its outgoing edge weights. For undirected, unweighted graphs, this reduces to the degree. In each epoch of training, 8 training examples are sampled uniformly from the Replay Memory with capacity $N = 50$ as described in Alg. 4. The sampling size z, in terms of percentage, is varied at $[1\%, 10\%, 30\%, 50\%, 75\%, 99\%]$ on the validation sets, and the best performing value is used. As we will show later in Fig.8c, 10% is often enough.

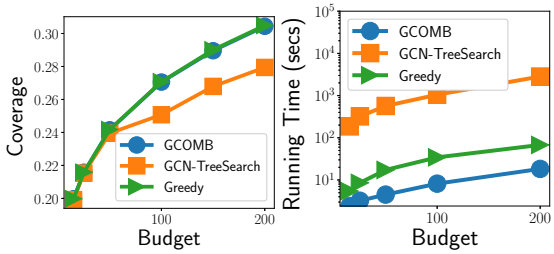
For all train sets, we split into two equal halves, where the first half is used for training and the second half is used for validation. For the cases where we have only one training graph, like BrightKite(BK) in MCP and MVC, we randomly pick 50% of the edges for the training graph and the remaining 50% for the validation graph. The noise predictor interpolators in MCP are fitted on 10% randomly edge-sampled subgraphs from Gowallah, Twitter-ew and YouTube. During testing, the remaining 90% subgraph is used, which is edge disjoint to the 10% of the earlier sampled subgraph.

I Extracting Subgraph from Gowalla

To extract the subgraph, we select a node proportional to its degree. Next, we initiate a breadth-first-search from this node, which expands iteratively till X nodes are reached, where X is the target size of the subgraph to be extracted. All of these X nodes and any edge among these nodes become part of the subgraph.

J Comparison with OPIM on billion sized graphs

Figs. 5a- 5b present the speed-up achieved by GCOMB over OPIM on Orkut and Friendster. Speed-up is measured as $\frac{time_{OPIM}}{time_{\text{GCOMB}}}$ where $time_{OPIM}$ and $time_{\text{GCOMB}}$ are the running times of OPIM and GCOMB respectively. OPIM crashes on Friendster-CO and Twitter dataset.



(a) Quality (b) Scalability

Figure 6: MCP : Gowalla: a) Quality comparison of GCOMB and GCN-TREESEARCH against greedy. b) Running times of GCOMB and GCN-TREESEARCH against the greedy approach.

Budget	**Speed-up** $\epsilon = 0.2$	**Coverage Difference** $\epsilon = 0.2$	**Coverage Difference** $\epsilon = 0.05$
20	2	-0.09	$\mathbf{-0.001}$
50	2	-0.13	$\mathbf{-0.003}$
100	2	-0.16	$\mathbf{-0.005}$
150	2	-0.18	$\mathbf{-0.005}$
200	2	-0.20	$\mathbf{-0.006}$

Table 4: Comparison with Stochastic Greedy(SG) algorithm. The ϵ parameter controls the accuracy of SG. A negative number means GCOMB is better than SG.

K Results on Max Cover Problem (MCP) on Gowalla

Fig. 6a presents the impact of budget on Coverage on Gowalla dataset. The quality achieved by GCOMB is similar to Greedy, while GCN-TREESEARCH is inferior. GCOMB is up to two orders of magnitude faster than GCN-TREESEARCH and 10 times faster than Greedy as can be seen in Fig. 6b.

L Comparison with Stochastic Greedy (SG) on MCP

We compare the performance of GCOMB with *SG* on MCP. As can be seen in Table 4, GCOMB is up to 20% better in quality at $\epsilon = 0.2$ and yet 2 times faster. SG fails to match quality even at $\epsilon = 0.05$, where it is even slower. Furthermore, SG is not drastically faster than CELF in MCP due to two reasons: (1) cost of computing marginal gain is $O(Avg.degree)$, which is fast. (2) The additional data structure maintenance in SG to access sampled nodes in sorted order does not substantially offset the savings in reduced marginal gain computations.

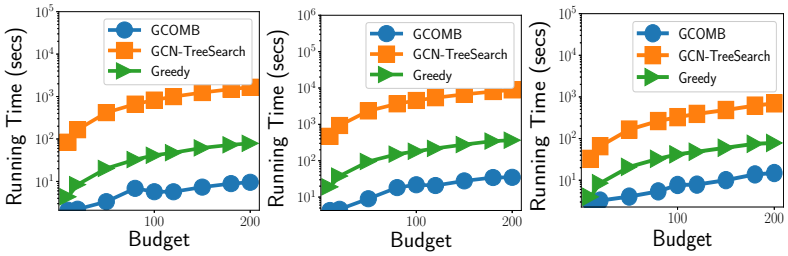
M Results on Max Vertex Cover (MVC)

To benchmark the performance in MVC, in addition to real datasets, we also use the Barabási–Albert (BA) graphs used in S2V-DQN [7].

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 graph, where X is the number of nodes.

For synthetic datasets, all three techniques are trained on BA graphs with $1k$ nodes. For real datasets, the model is trained on Brightkite. Table 5 presents the coverage achieved at $b = 30$. Both GCOMB and GCN-TREESEARCH produce results that are very close to each other and slightly better than S2V-DQN. As in the case of MCP, S2V-DQN ran out of memory on graphs larger than BA-20k.

To analyze the efficiency, we next compare the prediction times of GCOMB with Greedy (Alg 1) and GCN-TREESEARCH. Figs. 7a-7c present the prediction times against budget. Similar to the results in MCP, GCOMB is one order of magnitude faster than Greedy and up to two orders of magnitude faster than GCN-TREESEARCH.



(a) Gowalla (b) YouTube (c) Twitter-ego

Figure 7: MVC: Running times of GCOMB and GCN-TREESEARCH against the greedy approach in (a) Gowalla (b) YouTube and c) Twitter-Ego.

Graph	Greedy	S2V-DQN	GCN-TREESEARCH	GCOMB
BA-10k	**0.11**	0.096	0.109	**0.11**
BA-20k	**0.0781**	0.0751	**0.0781**	**0.0781**
BA-50k	**0.0491**	NA	0.0490	**0.0491**
BA-100k	**0.0346**	NA	0.0328	**0.0346**
Gowalla	**0.081**	NA	**0.081**	**0.081**
YouTube	**0.060**	NA	**0.060**	**0.060**
Twitter-ego	**0.031**	NA	**0.031**	**0.031**

Table 5: Coverage achieved in the Max Vertex Cover (MVC) problem. The best result in each dataset is highlighted in bold.

N Impact of Parameters

N.1 Size of training data

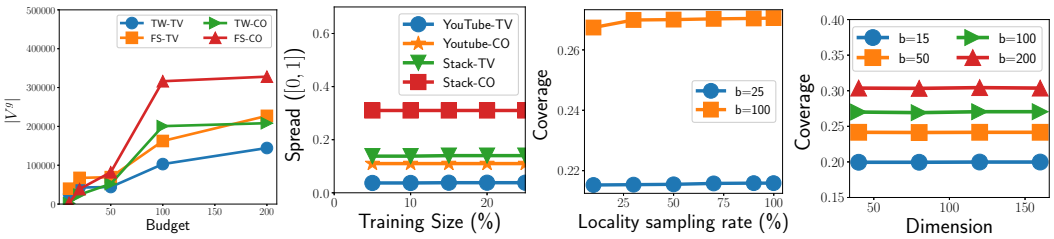
In Fig. 8b, we evaluate the impact of training data size on expected spread in IM. The budget for this experiment is set to 20. We observe that even when we use only 5% of YT to train, the result is almost identical to training with a 25% subgraph. This indicates GCOMB is able to learn a generalized policy even with small amount of training data.

N.2 Effect of sampling rate

We examine how the sampling rate in locality computation affects the overall coverage in MCP. In Fig. 8c, with the increase of samples, the accuracy at $b = 100$ increases slightly. At $b = 25$, the increase is negligible. This indicates that our sampling scheme does not compromise on quality.

N.3 Dimension

We vary the GCN embedding dimension from 40 to 160 and measure its impact on coverage in MCP (Fig. 8d). We observe minute variations in quality, which indicates that GCOMB is robust and does not require heavy amount of parameter optimization.



(a) TW and FS (b) Quality Vs. Training (c) Gowalla (d) Gowalla

Figure 8: (a) Number of nodes included in V^g in Twitter and Friendster for different budgets b. (b) Impact of training set size on spread quality in IM. (c-d) Effect of sampling rate and embedding dimension across different budgets on MCP coverage.