

In the format provided by the authors and unedited.

Finding key players in complex networks through deep reinforcement learning

Changjun Fan^{1,2}, Li Zeng¹, Yizhou Sun ²✉ and Yang-Yu Liu ^{3,4}✉

¹College of Systems Engineering, National University of Defense Technology, Changsha, China. ²Department of Computer Science, University of California, Los Angeles, CA, USA. ³Channing Division of Network Medicine, Brigham and Women's Hospital and Harvard Medical School, Boston, MA, USA. ⁴Center for Cancer Systems Biology, Dana-Farber Cancer Institute, Boston, MA, USA. ✉e-mail: yzsun@cs.ucla.edu; yyli@channing.harvard.edu

Supplementary Information for
Finding key players in complex networks through deep
reinforcement learning

Changjun Fan,^{1,2} Li Zeng,¹ Yizhou Sun,^{2,*} and Yang-Yu Liu^{3,4,†}

¹*College of Systems Engineering,
National University of Defense Technology, Changsha, 410073, China*

²*Department of Computer Science,
University of California, Los Angeles, CA, 90024, USA*

³*Channing Division of Network Medicine,
Brigham and Women's Hospital and Harvard Medical School, Boston, MA, 02115, USA*

⁴*Center for Cancer Systems Biology,
Dana-Farber Cancer Institute, Boston, MA, 02115, USA*

(Dated: March 25, 2020)

* Correspondence: yzsun@cs.ucla.edu

† Correspondence: yyl@channing.harvard.edu

CONTENTS

List of Figures	3
List of Tables	4
List of Algorithms	4
I. Existing methods of finding key players in complex networks	5
A. Node centrality based methods	5
B. Optimal decycling based methods	7
C. Graph partition based methods	8
II. Deep reinforcement learning for graph combinatorial optimization problems	10
A. Preliminary for graph embedding	10
1. Node embedding	11
2. Graph neural networks (GNNs)	11
B. Preliminary for reinforcement learning	14
1. Bellman equations for state and action value functions	15
2. The optimal policy	17
3. Q -learning and deep Q -learning (DQN)	20
C. S2V-DQN	21
1. Model architecture	22
2. Training algorithm	23
3. Complexity analysis	23
D. FINDER	23
1. Model architecture	23
2. Training algorithm.	26
3. Complexity analysis	28
E. Key differences between S2V-DQN and FINDER	30
1. Key differences	30
2. Ablation study	32
III. Descriptions of experimental datasets	32

IV. More details	34
A. Hyperparameters	34
B. The range of ANC	35
C. Effects of the fraction of removed nodes at each adaptive step	37
D. Several ways to further improve FINDER's performances	37
1. Tailor the training data to the target network	37
2. Enhanced with the NEP algorithm	38
E. FINDER's performance on spatial networks	39
F. FINDER's application to the minimal percolation threshold problem	40
G. Spreading power of the nodes selected by FINDER in influence maximization problems	42
References	87

LIST OF FIGURES

S1	Degree distribution of the real-world networks analyzed in this work	44
S2	Different percentages of nodes removed at each step - CN problem	45
S3	Different percentages of nodes removed at each step - ND problem	46
S4	Scalability analysis	47
S5	ANC curves on real-world networks (CN, node-unweighted)	48
S6	ANC curves on real-world networks (CN, node-degree-weighted)	49
S7	ANC curves on real-world networks (CN, node-random-weighted)	50
S8	ANC curves on real-world networks (ND, node-unweighted)	51
S9	ANC curves on real-world networks (ND, node-degree-weighted)	52
S10	ANC curves on real-world networks (ND, node-random-weighted)	53
S11	FINDER training convergences measured by validation performance	54
S12	ANC curves for MinSum, BPD, FINDER and their NEP-enhanced ones	55
S13	Illustration of FINDER that is designed to the minimal percolation threshold problem	56
S14	Comparison of the spreading power of nodes selected by FINDER with that of nodes selected by Hill-Climbing	58

LIST OF TABLES

S1	Summary of the typical node embedding models	59
S2	Ablation study between FINDER and S2V-DQN.....	60
S3	Basic statistics of real-world networks analyzed in this work	61
S4	List of hyper-parameters and their values	62
S5	Time complexity analysis	63
S6	Effects of batch nodes selection used in FINDER to solve the CN problem ..	64
S7	Effects of batch nodes selection used in FINDER to solve the ND problem ..	65
S8	Effects of batch nodes selection for HDA and FINDER	66
S9	NEP-enhanced results on nine real-world networks	67
S10	Results on synthetic networks (CN, node-unweighted)	68
S11	Results on synthetic networks (CN, node-degree-weighted)	70
S12	Results on synthetic networks (CN, node-random-weighted)	72
S13	Running time comparison on real-world networks (CN problem)	74
S14	Results on synthetic networks (ND, node-unweighted)	75
S15	Results on synthetic networks (ND, node-degree-weighted)	77
S16	Results on synthetic networks (ND, node-random-weighted)	79
S17	Running time comparison on real-world networks (ND problem)	81
S18	FINDER performance for the minimal percolation threshold problem	83
S19	FINDER performance on two power-grid networks	84
S20	Training FINDER using the same type of graphs as the test graphs yields the best performance	85
S21	FINDER performance via training with different types of synthetic graphs ..	86
S22	ANC scores of various methods, including Configuration Model-trained FINDER: FINDER_CM	87

LIST OF ALGORITHMS

S1	Three classical graph neural network models	13
S2	Encoding process in FINDER	25
S3	Training procedure of FINDER	30

I. EXISTING METHODS OF FINDING KEY PLAYERS IN COMPLEX NETWORKS

In this section, we describe a series of representative algorithms that have been developed to identify key players in networks. We treat them as baseline methods to compare with our method FINDER. We classify those methods into three categories, i.e., node centrality based heuristics, optimal decycling based algorithms, and graph partition based methods. Node centrality based heuristics rank nodes according to some local/global structural measures, which suffer from poor performances arising from either considering local information only or prohibitive running time of calculating global measures on large scale networks (e.g., HBA, HCA, and HPRA, which will not be compared with FINDER on real networks due to their high time complexity). Optimal decycling based algorithms are all designed for the network dismantling (ND) problem specifically, and we can not compare FINDER with them in the critical node (CN) problem. Graph partition based methods consider that the key players should be the ones that are essential to the graph partition, of which the removal leads to the fragmentation of a graph into several smaller mutually exclusive groups.

A. Node centrality based methods

Node centrality based methods construct local or global structure measures, such as degree, betweenness, closeness, PageRank, collective influence (CI), etc., and rank nodes according to those measures.

High Degree Adaptive (HDA)[1]. HDA is an adaptive version of high degree method[2], which ranks nodes solely by degree, and sequentially removes the node with the highest degree. HDA recomputes the degrees of the remaining nodes after each node removal.

High Betweenness Adaptive (HBA)[3]. HBA is the adaptive version of the high betweenness method. HBA sequentially removes the node with the highest betweenness centrality (BC), and recomputes BC for the remaining nodes. Note that BC of a node equals the sum of the fraction of all-pairs shortest paths that pass through this node. BC has been used in many related network applications, like community detection[4] and network vulnerability[5]. Empirically, we found HBA performs the best on the node unweighted

ND problem (Table S14). However, the high computing cost prohibits its use in medium and large scale networks. Currently, the best exact algorithm for betweenness centrality computation requires $O(NM)$ time complexity on unweighted networks with N nodes and E edges, and $O(NE + N^2 \log N)$ for weighted networks. And for the adaptive version, the time complexity is $O(N^2 E)$ for unweighted networks and $O(N^2 E + N^3 \log N)$ for weighted ones.

High Closeness Adaptive (HCA)[6]. HCA is the adaptive version of the high closeness method. HCA sequentially removes the node with the highest closeness centrality (CC), and updates CC for the remaining nodes. CC describes how close a node is to all the other nodes in the graph. It is calculated as the reciprocal of average distances from one node to all the others. CC tends to give high scores to nodes that are close to the center of local clusters, and hence it over-allocates spreaders next to each other[7]. Similar to HBA, the high complexity cost of HCA prevents its application in large networks. As such, we just compare it with FINDER on small synthetic data.

High PageRank Adaptive (HPRA)[8]. HPRA is the adaptive version of high PageRank method. HPRA greedily removes the node with the highest PageRank centrality, and then recomputes the PageRank for the remaining nodes. PageRank has been widely employed in search engines, as it provides a global ranking of all web pages, regardless of their content, based solely on their location in the Web’s graph structure[8]. PageRank computes the probabilities for a random-walking agent to reach every node in the network.

Collective Influence (CI)[7]. The Collective Influence measure is defined as the product of the node’s reduced degree (i.e., original degree minus one) and the sum of the reduced degrees of the nodes that are within a constant hop away from it. This measure describes the proportion of other nodes that can be reached from a given node, assuming the nodes with higher CI values play more crucial roles in networks. The CI method sequentially removes the node with the highest CI value and recalculates the CI values of the remaining nodes for the rest following operations. It is also very efficient: the time complexity is $O(N \log N)$ when a fixed fraction of nodes is removed at each adaptive step.

B. Optimal decycling based methods

Recent studies[9–11] reveal that the network dismantling problem is closely related to the optimal decycling problem, which seeks to remove minimum nodes to break all the loops in the network. The insight behind the connection between network decycling and network dismantling is that, short loops rarely exist in small connected components for sparse random networks, and if we cut the long loops in the giant connected component (GCC), the network will break into small tree fragments. As indicated by Braunstein[9], the optimal decycling threshold q_c^{dec} acts as an upper bound of the optimal dismantling threshold q_c . These methods follow very similar three stages: (i) break all the loops in the network; (ii) break the remaining trees into components with sub-extensive sizes; (iii) reintroduce some nodes to close cycles without increasing the size of the largest component, and mainly differ in the first network decycling stage. MinSum and BPD both utilize message-passing algorithms to break all the cycles, while CoreHD greedily removes the highest degree node in the 2-core until there are no loops. Note that these methods work only for the ND problem, they are not applicable to the CN problem.

MinSum[9]. MinSum mapped the decycling process to a 2-core percolation due to the fact that a network is acyclic if and only if its 2-core is empty. Assume a set of nodes $\mathcal{S} \subset \mathcal{V}$ are initially removed from the network. The 2-core percolation can be described by the time-dependent binary variable $x_i^t(\mathcal{S})$ for $1 \leq i \leq N$. Starting from the initial setting $x_i^0(\mathcal{S}) = 1$ for removed nodes $i \in \mathcal{S}$ and $x_i^0(\mathcal{S}) = 0$ for $i \notin \mathcal{S}$ at $t = 0$, the recursion follows:

$$x_i^{t+1} = \begin{cases} 1 & \text{if } x_i^t(\mathcal{S}) = 1 \\ \mathbb{1}[\sum_{j \in \mathcal{N}(i)} (1 - x_j^t(\mathcal{S})) \leq 1] & \text{if } x_i^t(\mathcal{S}) = 0 \end{cases}$$

where the indicator function $\mathbb{1}$ is 1 if the argument is true and 0 otherwise. Since x_i^t can only change from 0 to 1, the equations approach a fixed solution $x_i^*(\mathcal{S})$ as $t \rightarrow \infty$. In practice, $x_i^*(\mathcal{S}) = 1$ iff i belongs to the 2-core of $\mathcal{G} \setminus \mathcal{S}$. If $x_i^*(\mathcal{S}) = 1$ for all nodes, $\mathcal{G} \setminus \mathcal{S}$ contains no loops and $\mathcal{S}$ is called a decycling set. The minimal decycling set can be computed by iterating a set of equations via the MinSum algorithm. In most cases, convergence can be reached within a small number of iterations, in case the Min-Sum equations do not converge,

a reinforcement procedure is applied to damp the system[12]. In the acyclic network $\mathcal{G} \setminus \mathcal{S}^*$, there may still exist some very large tree components, which can be fragmented efficiently via a greedy tree breaking procedure. In addition, for networks containing many short loops, a reverse greedy reinsertion procedure is applied to add back the nodes that do not increase the GCC size.

Belief Propagation-guided Decimation (BPD)[10]. BPD is based on the spin glass model of the minimum FVS problem[13]. FVS denotes feedback vertex set, which is a set of nodes whose removal would break all the loops in the network. To obtain the FVS, BPD calculates the removal probability $q_i(t)$ for node i at time step t , through iterations of a set of belief propagation (BP) equations[10], and the node with the highest probability q_i is removed from the network even if the BP equations do not converge to a fixed point. The process stops when the network becomes acyclic. After that, if the remaining largest component remains extensive, it can be further fragmented by iteratively deleting root nodes from the remaining trees until the GCC size reaches the predefined threshold. Finally, to reduce the total number of removed nodes, a reinsertion process can be proceeded without significantly increasing the GCC size. This process can be done through a greedy algorithm in which the nodes that cause the least increase in GCC size are reinserted one after another until the GCC size meets the requirement. The BPD algorithm is scalable to large networks with computational complexity $O(N \log N)$.

CoreHD[11]. Inspired by the above two methods, a simple and fast heuristic algorithm, CoreHD, was developed. Starting from the 2-core of a network, CoreHD obtains the degrees of every node within this 2-core (edges to outside nodes are not considered); then removes the node with the highest degree in the 2-core. If there are multiple nodes with the same highest degree, one of them will be chosen randomly. Then it updates the 2-core and the degrees of all its nodes after each node's removal. If the 2-core is empty, CoreHD breaks the remaining trees and adds back some nodes that do not increase the GCC size, otherwise it returns to the second step, i.e., removes the highest degree node in the updated 2-core.

C. Graph partition based methods

Graph partition aims to reduce a graph to smaller subgraphs by partitioning its nodes into mutually exclusive groups. Those methods seek to remove minimum edges (nodes) to

partition a graph into several components with the similar number of nodes (RatioCut[14]) or weights of the edges (normalized cut[?]). A graph can be quickly demolished by partitioning in this way recursively. Here we compare FINDER with RatioCut in the CN problem and Generalized Network-Dismantling (GND)[15] in the ND problem. There is another method named equal-graph-partitioning (EGP)[16], which achieves the same objective. In this work, we didn't consider it due to its prohibitive running time for large-scale networks, and its poor performance in comparison with other methods[7].

RatioCut[14, 17]. The RatioCut method seeks to remove the minimum number of edges (or edge weights) to partition a graph into several components of similar sizes. It has been proved that the solution is given by the eigenvector corresponding to the second smallest eigenvalue of the unnormalized graph Laplacian[14, 17]. To handle the CN problem, we recursively applied the RatioCut procedure for smaller and smaller partitions until the network was completely fragmented. For each partition, the nodes were divided into two sets, i.e., $\{v_i \leq 0\}$ and $\{v_i > 0\}$, where v_i is the i th corresponding value in the second eigenvector. The original RatioCut method just removes the separating edges that connect these two sets of nodes, while we focus on the node-removal procedure here. To achieve this, we considered a new sub-graph $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$, where $\mathcal{E}^*$ denotes the separating edges, $\mathcal{V}^*$ denotes the nodes that are adjacent to $\mathcal{E}^*$. Then RatioCut used a greedy method[18] to solve the minimum vertex cover problem on $\mathcal{G}^*$, and the selected nodes were to be removed at this partitioning step. To scale the RatioCut method for large networks, we utilized the power iteration technique[19] to efficiently calculate the second smallest eigenvalue and its corresponding eigenvector of the unnormalized graph Laplacian.

Generalized Network-Dismantling (GND)[15]. GND is the state-of-the-art method to address the ND problem in the node-weighted scenario. GND considers a node-weighted spectral cut problem and proposes a node-weighted objective, where the cost of cutting an edge equals the costs of removing the two adjacent nodes. Based on the objective, GND constructs a node-weighted Laplacian and follows the standard spectral cut procedure. To find the second-smallest eigenvectors for large-scale networks, a power-iteration based approximation algorithm can be employed. GND also puts forward a fine-tuning method (same as what we used in RatioCut), which transfers the edge-removal strategy to the node-removal strategy. In the end, GND reintroduces[20] some nodes to produce a better dismantling solution, since some nodes from early stages of fragmentation do not contribute

to the final stage of complete fragmentation. Note that when the cost matrix equals the identity matrix, GND reduces to the standard RatioCut method.

II. DEEP REINFORCEMENT LEARNING FOR GRAPH COMBINATORIAL OPTIMIZATION PROBLEMS

In this section, we first introduced the prerequisite knowledge for graph embedding and reinforcement learning, then we described the S2V-DQN framework, which is a state-of-the-art deep reinforcement learning architecture for graph combinatorial optimization problems. Afterward, we introduced the architecture and training procedure of FINDER in detail and the key distinctions between FINDER and S2V-DQN. We also conducted ablation studies to analyze the contributions of each different component.

A. Preliminary for graph embedding

Network/graph embedding, a.k.a, network representation learning, seeks to capture the node proximity and graph topology in a continuous low-dimensional embedding space, which can be easily fed into downstream machine learning models.

Traditional feature extraction approaches for graphs often rely on statistical graph properties, such as degree distributions, clustering coefficients, kernel functions, etc., to measure local or global graph structures. These features are often hard to design and expensive to compute. Hence, recent efforts use embeddings as graph features for various downstream machine learning tasks, such as node classification[21], graph classification[22], community detection[23], link prediction[24] and so on. This embedding differs fundamentally from traditional feature engineering methods in that it treats the feature extraction as a learning task, which uses a data-driven approach to learn node embeddings to encode the graph structure.

Current embedding-based models share a similar encoder-decoder framework[25], where the encoder part (Eq. S1) maps nodes to low-dimensional vectors, and the decoder part (Eq. S2) infers network structural information from the encoded vectors. Under this framework, there are two main categories, i.e., node embedding models and graph neural

networks (GNNs).

$$\mathbf{E} : \mathcal{V} \rightarrow \mathbb{R}^d \quad (\text{S1})$$

$$\mathbf{D} : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}_{\geq 0} \quad (\text{S2})$$

where $\mathcal{V}$ is the node set, $\mathbb{R}^d$ is a d -dimensional real-value vector, $\mathbb{R}_{\geq 0}$ is a non-negative real number.

1. Node embedding

Node embedding models are direct encoding approaches, where the encoder function is just an embedding lookup function:

$$\mathbf{E}(v_i) = \mathbf{Z}\mathbf{v}_i \quad (\text{S3})$$

where $\mathbf{Z} \in \mathbb{R}^{d \times |\mathcal{V}|}$ is a matrix containing all node embedding vectors, which are also the trainable parameters to be optimized, $\mathbf{v}_i \in I_{\mathcal{V}}$ is a one-hot indicator vector indicating the column of $\mathbf{Z}$, which corresponds to the node v_i . The decoder function is typically based on the inner product of embeddings, which seeks to obtain deterministic measures such as network proximity[26, 27] or statistics derived from random walks[21, 24]. The encoder and decoder functions of typical node embedding models, including DeepWalk[21], LINE[26] and node2vec[24], are summarized in Table S1.

Direct encoding models have been successfully applied in many graph mining tasks, like multi-label node classification, link prediction, and visualization. However, since these models train unique embedding vectors for each node independently, they all suffer from several major limitations. First, they do not consider node attributes, which are quite informative in practice. Second, no parameters are shared across nodes, and the number of parameters necessarily grows as $O(|\mathcal{V}|)$, which is computationally inefficient. Third, they cannot handle previously unseen nodes, which prevents their applications on dynamic networks.

2. Graph neural networks (GNNs)

To address the above limitations, the second category of models have been proposed recently, which are known as the neighborhood aggregation models[28–31], or graph neural

networks (GNNs). For these models, the encoder function iteratively aggregates embedding vectors from the neighborhood, which are initialized as node feature vectors, and followed by a non-linear transformation operator. When nodes are not associated with any attributes, simple network statistics based features are often adopted, such as node degrees and local clustering coefficients. The decoder function can either be the same as the ones in the previous category or be integrated with task-specific supervisions[32]. GNNs turn out to be more effective in practice, due to their flexibility in incorporating node attributes and inductive settings in handling unseen nodes or graphs, which makes them more adaptive to downstream tasks.

Let $h_v^{(l)}$ denote the embedding vector for node v at layer l , a typical GNN can be defined in two steps: (i) the aggregation step that aggregates the embedding vectors from the neighbors of v at layer $l - 1$; (ii) the transformation step that transforms the embedding of v in the last layer and the aggregated neighbor embedding to its current embedding:

$$h_{\mathcal{N}(v)}^{(l)} = \text{AGG}(\{h_u^{(l-1)}, \forall u \in \mathcal{N}(v)\}) \quad (\text{S4})$$

$$h_v^{(l)} = \sigma(W^{(l)} \cdot \text{COM}(h_v^{(l-1)}, h_{\mathcal{N}(v)}^{(l)})) \quad (\text{S5})$$

where $\mathcal{N}(\cdot)$ denotes a set of neighboring nodes of a given node, $W^{(l)}$ is a trainable weight matrix of the l th layer shared by all nodes, and σ is an activation function, e.g., ReLU. AGG is a function that aggregates information from local neighbors, e.g., summation or weighted average, while COM is a function that combines the representation of node v at layer $l - 1$, i.e., $h_v^{(l-1)}$, with the aggregated neighborhood representation at layer l , i.e., $h_{\mathcal{N}(v)}^{(l)}$. The AGG function and the COM function can be defined specifically in different models. Algorithm S1 compares three classical graph neural network models, including structure2vec[31], GCN[29] and GraphSAGE[30].

As shown in Algorithm S1, structure2vec, GCN and GraphSAGE are basically the same. They differ in the design of the AGG function and the COM function. Essentially, most current GNNs focus on addressing the following questions:

- **How to define the neighborhood ?** Some directly use all adjacent nodes as neighbors[29], while others just sample some of them as neighbors[30].
- **How to choose the AGG function ?** Multiple aggregation functions, such as summation[29], weighted average[29, 30] have been proposed. In particular, [30]

introduced some attention-based model that firstly learned weights adaptively for different neighbors and then averaged them according to their weights.

- **How to choose the COM function ?** Summation[29] and concatenation[30] are two most common COM functions. Moreover, [33] proposed to utilize the Gated Recurrent Units (GRU) for the combination operation.
- **How to treat different layers' representations ?** In principle, deeper models get access to more information. However, in practice, more layers, even with residual connections, do not perform better than fewer layers (e.g., 2 layers)[29]. In [34], the authors pointed out that this maybe due to the fact that real-world complex networks possess locally varying structures. They proposed three layer-aggregation mechanisms: concatenation, max-pooling, and LSTM-attention, to enable adaptive, structure-aware representations.

Algorithm S1 Three classical graph neural network models

structure2vec	GCN	GraphSAGE
Input: parameters W , input features X_v	Input: parameters W , input features X_v	Input: W , input features X_v , aggregate function $\text{AGG}^{(k)}$
Output: $\{h_v^K\}, \forall v \in \mathcal{V}$	Output: $\{h_v^K\}, \forall v \in \mathcal{V}$	Output: $\{h_v^K\}, \forall v \in \mathcal{V}$
1: Initialize $h_v^0 = 0, \forall v \in \mathcal{V}$	8: Initialize $h_v^0 = X_v, \forall v \in \mathcal{V}$	15: Initialize $h_v^0 = X_v, \forall v \in \mathcal{V}$
2: for $l = 1$ to K do	9: for $l = 1$ to K do	16: for $l = 1$ to K do
3: for $v \in \mathcal{V}$ do	10: for $v \in \mathcal{V}$ do	17: for $v \in \mathcal{V}$ do
4: $h_{\mathcal{N}(v)}^{(l)} =$ $\sum_{j \in \mathcal{N}(v)} h_j^{(l-1)}$	11: $h_{\mathcal{N}(v)}^{(l)} =$ $\sum_{j \in \mathcal{N}(v)} \frac{1}{\sqrt{d_v+1}\sqrt{d_j+1}} h_j^{(l-1)}$	18: $h_{\mathcal{N}(v)}^{(l)} =$ $\text{AGG}^{(l)}(h_j^{(l-1)})_{j \in \mathcal{N}(v)}$
5: $h_v^{(l)} =$ $\sigma(W_1 \cdot X_v + W_2 \cdot h_{\mathcal{N}(v)}^{(l)})$	12: $h_v^l =$ $\sigma(W^{(l)} \cdot (\frac{1}{d_v+1} h_v^{(l-1)} + h_{\mathcal{N}(v)}^{(l)}))$	19: $h_v^{(l)} =$ $\sigma(W^{(l)} \cdot [h_v^{(l-1)}, h_{\mathcal{N}(v)}^{(l)}])$
6: end for	13: end for	20: end for
7: end for	14: end for	21: end for

In this work, we followed the GNN architecture to design our graph embedding model (Algorithm S2), which is similar to GraphSAGE, but differs in the input features and the neighborhood definition.

B. Preliminary for reinforcement learning

In a general reinforcement learning framework, an agent learns to act and interact with an evolving environment to reach a certain goal by receiving the maximal accumulation of corresponding rewards. The major difference between reinforcement learning and supervised learning is that the former has no direct supervision for the agent to learn. That is to say, the agent is never directly told the best action under certain state, but learns by itself to realize whether its previous sequence of actions are right or not only when an episode ends. Hence, this process can be regarded as a delayed supervised learning with states as features and actions as potential labels.

The reinforcement learning problem can be mathematically formalized as a Markov Decision Process (MDP), which is denoted as a quintuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma \rangle$, where $\mathcal{S}$ is the set of possible states, $\mathcal{A}$ is the set of possible actions, $\mathcal{R}$ is the distribution of rewards given a (state, action) pair, P is the transition probability that represents the distribution over next state if given the (state, action) pairs, and $\gamma \in (0, 1)$ is a discount factor, determining the importance of future rewards. Since it is a Markov Decision Process, there are two hidden Markov assumptions: (i) the Markov transition assumption, defined as $P(s_{t+1}|s_0, a_0, \dots, s_t, a_t) = P(s_{t+1}|s_t, a_t)$, which regulates that each of the agent's next states is only determined by the very last previous state and action, and has no relation to the any earlier ones; (ii) the Markov reward assumption, which is defined as $\mathcal{R}(s_{t+1}|s_0, a_0, \dots, s_t, a_t) = \mathcal{R}(s_{t+1}|s_t, a_t) = \mathcal{R}(s_t, a_t, s_{t+1}) = r_t \in \mathbb{R}$.

For a whole game, or a complete episode, the environment samples an initial state $s_0 \sim p(s_0)$. Then at each time step t , the agent selects an action $a_t \sim \pi(\cdot|s_t)$, where π represents the policy. Correspondingly, the environment provides feedback of the next state $s_{t+1} \sim P(\cdot|s_t, a_t)$ and the agent receives the next reward $r_t = \mathcal{R}(s_t, a_t, s_{t+1})$. Following this process until the episode ends, a trajectory $\tau = (s_0, a_0, r_0, s_1, r_1, a_1, \dots)$ is determined by the agent under the policy π . The learning objective is to find an optimal policy π^* that maximizes the cumulative rewards for this whole trajectory: $\sum_{t \geq 0} \gamma^t r_t$.

This optimal policy can either be learned directly, e.g., via the policy gradient method[35], or be constructed indirectly based on some learned quantities, e.g., Q value[36]. These two models mainly differ in how they approach the action selection. The former learns a direct mapping from states to actions, which can be stochastic and works in continuous action

spaces. The latter learns a single deterministic action from a discrete set of actions by finding the maximum value. The latter has been proven to be more sample efficient than the former[37] and has guarantees of convergence. In our problem, we are asked to learn a deterministic policy that at each step, only a single node is selected from a discrete set of possible actions (nodes). As a result, we choose the value-based reinforcement learning model, i.e., Q -learning, as the learning part in FINDER.

For the value-based models, the agent learns a value, which estimates the cumulative rewards starting from either current state s (state-value function, $V^\pi(s)$, Eq. S6) or current state-action pair (s, a) (action-value function, $Q^\pi(s, a)$, Eq. S7), to the end of the episode, with the following policy π throughout.

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k \geq 0} \gamma^k r_{t+k} \mid s_t = s, \pi \right] \quad (\text{S6})$$

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k \geq 0} \gamma^k r_{t+k} \mid s_t = s, a_t = a \right] \quad (\text{S7})$$

A common way of deriving a new policy from an action-value function is to act ϵ -greedily concerning the action values. In other words, one takes the action with the highest value with probability $(1-\epsilon)$, and otherwise to take a random action with probability ϵ [38]. This strategy is known as *exploration* and *exploitation*, which has been proved that once the action-value function $Q(s, a)$ is optimal, the agent could obtain the maximum cumulative rewards following this ϵ -greedy policy[39].

1. Bellman equations for state and action value functions

Based on Eq. S6, we can deduce the Bellman equation of $V^\pi(s)$ that can be used to recursively approximate the state-value function:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} p(s'|s, a) [\mathcal{R}(s, a, s') + \gamma V^\pi(s')] \quad (\text{S8})$$

where $\pi(a|s)$ is the probability to take action a under the state s , $p(s'|s, a)$ is the environment's transition probability to next state s' given current state s and action a , $\mathcal{R}(s, a, s')$ is the received reward when agent takes action a under state s and next state changes to s' , γ is the discount factor.

Proof. Since $V^\pi(s) = \mathbb{E}_\pi[\sum_{k \geq 0} \gamma^k r_{t+k} | s_t = s, \pi]$, we have:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[r_t + \gamma \sum_{k \geq 0} \gamma^k r_{t+1+k} | s_{t+1} = s', \pi] \\ &= \mathbb{E}_\pi[r_t + \gamma V^\pi(s')] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} p(s'|s, a) [\mathcal{R}(s, a, s') + \gamma V^\pi(s')] \end{aligned}$$

Thus Eq. S8 holds. $\square$

And the Bellman equation for $Q^\pi(s, a)$ is:

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} p(s'|s, a) [\mathcal{R}(s, a, s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s', a')] \quad (\text{S9})$$

Proof. Since $Q^\pi(s, a) = \mathbb{E}_\pi[\sum_{k \geq 0} \gamma^k r_{t+k} | s_t = s, a_t = a]$, we have:

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi[r_t + \gamma \sum_{k \geq 0} \gamma^k r_{t+1+k} | s_{t+1} = s'] \\ &= \mathbb{E}_\pi[r_t + \gamma V^\pi(s')] \\ &= \mathbb{E}_\pi[r_t + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s', a')] \\ &= \sum_{s' \in \mathcal{S}} p(s'|s, a) [\mathcal{R}(s, a, s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s', a')] \end{aligned}$$

Thus Eq. S9 holds. $\square$

From the above derivations, we can also obtain the following relationships between $V^\pi(s)$ and $Q^\pi(s, a)$:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q^\pi(s, a) \quad (\text{S10})$$

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} p(s'|s, a) [\mathcal{R}(s, a, s') + \gamma V^\pi(s')] \quad (\text{S11})$$

The optimal state-value function is defined as $V^*(s) = \max_\pi V^\pi(s) = V^{\pi^*}(s), \forall s \in \mathcal{S}$, where π^* denotes the optimal policy. The Bellman equation for $V^*(s)$ can be written as:

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s, a) [\mathcal{R}(s, a, s') + \gamma V^*(s')] \quad (\text{S12})$$

Proof. Since $V^*(s) = \max_{a \in \mathcal{A}} Q^*(s, a)$, we have:

$$\begin{aligned}
V^*(s) &= \max_{a \in \mathcal{A}} \mathbb{E}_{\pi^*} \left[\sum_{k \geq 0} \gamma^k r_{t+k} \mid s_t = s, a_t = a \right] \\
&= \max_{a \in \mathcal{A}} \mathbb{E}_{\pi^*} \left[r_t + \gamma \sum_{k \geq 0} \gamma^k r_{t+1+k} \mid s_{t+1} = s' \right] \\
&= \max_{a \in \mathcal{A}} \mathbb{E}_{\pi^*} [r_t + \gamma V^*(s')] \\
&= \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s' \mid s, a) [\mathcal{R}(s, a, s') + \gamma V^*(s')]
\end{aligned}$$

Thus Eq. S12 holds. $\square$

Similarly, the optimal action-value function is defined as: $Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) = Q^{\pi^*}(s, a), \forall (s, a) \in \mathcal{S} \times \mathcal{A}$, and the Bellman equation for $Q^*(s, a)$ can be derived as:

$$Q^*(s, a) = \sum_{s' \in \mathcal{S}} p(s' \mid s, a) [\mathcal{R}(s, a, s') + \gamma \max_{a'} Q^*(s', a')] \quad (\text{S13})$$

Proof. Since $Q^*(s, a) = \mathbb{E}_{\pi^*} [\sum_{k \geq 0} \gamma^k r_{t+k} \mid s_t = s, a_t = a]$, we have:

$$\begin{aligned}
Q^*(s, a) &= \mathbb{E}_{\pi^*} \left[r_t + \gamma \sum_{k \geq 0} \gamma^k r_{t+1+k} \mid s_{t+1} = s' \right] \\
&= \mathbb{E}_{\pi^*} [r_t + \gamma V^*(s')] \\
&= \mathbb{E}_{\pi^*} [r_t + \gamma \max_{a'} Q^*(s', a')] \\
&= \sum_{s' \in \mathcal{S}} p(s' \mid s, a) [\mathcal{R}(s, a, s') + \gamma \max_{a'} Q^*(s', a')]
\end{aligned}$$

Thus Eq. S13 holds. $\square$

Note that Eq. S13 inspires the design of loss functions in current Q -learning models (Eq. S20, S22, S25, and S27).

2. The optimal policy

In reinforcement learning, we often wish to obtain the optimal policy that accumulates the most rewards for the agent. To achieve that, we have to answer the following three questions:

Q1: Given a policy π , how to compute the value functions $V^{\pi}(s)$ and $Q^{\pi}(s, a)$?

Q2: Given a policy π , can we create another policy π' , such that $\pi' \geq \pi$, which means $V^{\pi'}(s) \geq V^\pi(s), \forall s \in \mathcal{S}$ or $Q^{\pi'}(s, a) \geq Q^\pi(s, a), \forall (s, a) \in \mathcal{S} \times \mathcal{A}$?

Q3: How can we get the optimal policy π^* , such that $\forall \pi$, we have $V^{\pi^*}(s) \geq V^\pi(s), \forall s \in \mathcal{S}$ or $Q^{\pi^*}(s, a) \geq Q^\pi(s, a), \forall (s, a) \in \mathcal{S} \times \mathcal{A}$?

For the first question, we can use Bellman equations for the state-value function (Eq. S8) and the action-value function (Eq. S9) to iteratively compute $V^\pi(s)$ and $Q^\pi(s, a)$, respectively, which are both initialized to be 0, and the process will continue until convergence.

For the second question, given a policy π , we can create a new policy like this:

$$\pi' = \operatorname{argmax}_a Q^\pi(s, a) \quad (\text{S14})$$

Now, let's prove $\pi' \geq \pi$, i.e., $V^{\pi'}(s) \geq V^\pi(s), \forall s \in \mathcal{S}$ and $Q^{\pi'}(s, a) \geq Q^\pi(s, a), \forall (s, a) \in \mathcal{S} \times \mathcal{A}$:

Proof. To prove $V^{\pi'}(s) \geq V^\pi(s)$, we first have $V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q^\pi(s, a) \leq \max_a Q^\pi(s, a)$, since $\max_a Q^\pi(s, a) = Q^\pi(s, \pi'(s))$, we have:

$$\begin{aligned} V^\pi(s) &\leq Q^\pi(s, \pi'(s)) \\ &= \mathbb{E}_{\pi'}[r_t + \gamma V^\pi(s_{t+1}) | s_t = s] \\ &\leq \mathbb{E}_{\pi'}[r_t + \gamma Q^\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s] \\ &= \mathbb{E}_{\pi'}[r_t + \gamma \mathbb{E}_{\pi'}[r_{t+1} + \gamma V^\pi(s_{t+2}) | s_t = s]] \\ &= \mathbb{E}_{\pi'}[r_t + \gamma r_{t+1} + \gamma^2 V^\pi(s_{t+2}) | s_t = s] \\ &\leq \mathbb{E}_{\pi'}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t = s] \\ &= V^{\pi'}(s) \end{aligned}$$

Thus $V^{\pi'}(s) \geq V^\pi(s)$ □

For the last question, usually there are two main categories of methods: (i) the model-based approaches[40], including the policy iteration method and the value iteration method; (ii) the model-free approaches, including the Monte Carlo method[40] and Temporal Difference (TD) learning[41].

Proof. To prove $Q^{\pi'}(s, a) \geq Q^{\pi}(s, a)$, we first have (based on Eq. S11):

$$\begin{aligned} Q^{\pi'}(s, a) &= \sum_{s' \in \mathcal{S}} p(s'|s, a) [\mathcal{R}(s, a, s') + \gamma V^{\pi'}(s')] \\ &\geq \sum_{s' \in \mathcal{S}} p(s'|s, a) [\mathcal{R}(s, a, s') + \gamma V^{\pi}(s')] \\ &= Q^{\pi}(s, a) \end{aligned}$$

Thus, $Q^{\pi'}(s, a) \geq Q^{\pi}(s, a)$ □

To use the model-based approaches, we need to know the transition probability $p(s'|s, a)$. For the policy iteration method, we begin with any random policy, then use Eq. S9 to iteratively compute the stable action-value function under this policy. Afterward, we improve the policy with Eq. S14, and continue computing the action-value function under the new policy until we obtain the stable policy; for the value iteration method, we first use Eq. S13 to iteratively compute the optimal action-value function $Q^*(s, a)$, and then get the optimal policy $\pi^* = \operatorname{argmax}_a Q^*(s, a)$.

However, in most cases, we cannot obtain the transition probability $p(s'|s, a)$, which forces us to utilize the model-free approaches, including Monte Carlo (MC) methods and Temporal Difference (TD) methods, to approximate the value functions.

For the Monte Carlo method, let $R(s_t, a_t)$ be the accumulative rewards that can be achieved from state s_t and action a_t following policy π , as such, the expectation of $R(s_t, a_t)$ approximates the action-value function $Q^{\pi}(s_t, a_t)$:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi} \left[\sum_{k \geq 0} \gamma^k r_{t+k} | s_t, a_t \right] = \mathbb{E}_{\pi} [R(s_t, a_t)] \quad (\text{S15})$$

In practice, we sampled N simulations starting from (s_t, a_t) following the policy π and observed the accumulative rewards $R_1(s_t, a_t), R_2(s_t, a_t), \dots, R_N(s_t, a_t)$ for each simulation. We utilized the average $\frac{1}{N} \sum_{k=1}^N R_k(s)$ as an estimator of $Q^{\pi}(s_t, a_t)$. To incrementally update the $Q^{\pi}(s_t, a_t)$ after the k th simulation finished, we have:

$$Q_k^{\pi}(s_t, a_t) = Q_{k-1}^{\pi}(s_t, a_t) + \frac{1}{k} \cdot (R_k(s_t, a_t) - Q_{k-1}^{\pi}(s_t, a_t)) \quad (\text{S16})$$

The MC method does not need to know the transition probability of the model, however, it has to wait until the end of an episode to obtain $R(s_t, a_t)$, which is still not efficient in

practice. To avoid this, TD learning considers using $r_t + \gamma Q_{k-1}^\pi(s_{t+1}, a_{t+1})$ to approximate $R_k(s_t, a_t)$ (Eq. S17).

$$\begin{aligned} R_k(s_t, a_t) &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \\ &= r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \dots) \\ &= r_t + \gamma Q_{k-1}^\pi(s_{t+1}, a_{t+1}) \end{aligned} \tag{S17}$$

Replacing the $R_k(s_t, a_t)$ with $r_t + \gamma Q_{k-1}^\pi(s_{t+1}, a_{t+1})$ in Eq. S16, we have:

$$Q_k^\pi(s_t, a_t) = Q_{k-1}^\pi(s_t, a_t) + \frac{1}{k} \cdot [(r_t + \gamma Q_{k-1}^\pi(s_{t+1}, a_{t+1})) - Q_{k-1}^\pi(s_t, a_t)] \tag{S18}$$

which means we can sample enough simulations and use Eq. S18 to update $Q^\pi(s_t, a_t)$ until the convergence.

Furthermore, we can use TD method to approximate the optimal action-value function $Q^*(s_t, a_t)$ via the following recursive updates:

$$Q_k^*(s_t, a_t) = Q_{k-1}^*(s_t, a_t) + \frac{1}{k} \cdot [(r_t + \gamma \max_{a'} Q_{k-1}^*(s_{t+1}, a')) - Q_{k-1}^*(s_t, a_t)] \tag{S19}$$

As k approximates infinity, $Q^*(s_t, a_t)$ will converge to a unique value[40].

3. *Q-learning and deep Q-learning (DQN)*

Traditional Q-learning methods use the Q-table to update the value for each state-action pair via Eq. S19. However, it would be very expensive to compute $Q(s, a)$ for each state-action pair in large state and action spaces. Most efforts often use a function approximator to estimate the $Q(s, a)$, e.g., a neural network, we call them as Q-learning methods. If the function approximator is a deep neural network, we refer to as the deep Q network (DQN)[36].

DQN was first proposed by[36] and had achieved human level control in playing the Atari games. It used a convolution neural network (CNN) to approximate the action-value. At each state, DQN was fed with a stack of raw pixel frames (states), then a CNN was used for the ϵ -greedy selection of the next action until the game ended. After each episode, the DQN agent sampled a trajectory and extracted several transitions (s_t, a_t, r_t, s_{t+1}) into a experience replay buffer that holds the last a million transitions. The parameters of the neural network are optimized by using stochastic gradient descents over mini-batch samples

drawn uniformly from the memory buffer to minimize the loss function:

$$(r_t + \gamma \max_a \hat{Q}(s_{t+1}, a; \hat{\Theta})) - Q(s_t, a_t; \Theta) \quad (\text{S20})$$

We can see Eq. S20 can be derived from Eq. S19 if we make some transformations:

$$k(Q_k^*(s_t, a_t) - Q_{k-1}^*(s_t, a_t)) = (r_t + \gamma \max_{a'} Q_{k-1}^*(s_{t+1}, a')) - Q_{k-1}^*(s_t, a_t) \quad (\text{S21})$$

In Eq. S20, the gradients of the loss is back-propagated only into Θ of the on-line network. The term $\hat{\Theta}$ thereof represents the parameters of a target network, which are periodically copied from the on-line network and are treated as constants between the individual updates. [36] discovers that the use of experience replay buffer and target network enables a stable learning Q values, and leads to superhuman performances on several Atari games.

DQN has been a great improvement compared to traditional Q -learning ones, and afterward, several extensions have been proposed to address the limitations of DQN. Such as Double DQN[42], prioritized sampling[43], n -step DQN[44], dueling DQN[45], distributional DQN[46], etc. We have implemented all these DQN extensions in our problem and found that only n -step Q -learning improves the results, while other extensions bring along merely marginal effects or even hinder the performance. Therefore, we only employ the n -step DQN model as the learning part of our framework.

The n -step DQN waits for n steps, rather than the traditional one step, before updating the parameters, to collect a more accurate estimate of the future rewards. The truncated n -step reward from a given state s_t is defined as $r_{t,t+n} = \sum_{i=0}^{n-1} r(s_{t+i}, a_{t+i})$. Therefore, the corresponding loss is then modified as:

$$(r_{t,t+n} + \gamma \max_a \hat{Q}(s_{t+n}, a; \hat{\Theta})) - Q(s_t, a_t; \theta) \quad (\text{S22})$$

C. S2V-DQN

S2V-DQN[37] is a state-of-the-art deep learning architecture to solve graph combinatorial optimization problems. It is a combination of structure2vec[31], a graph embedding algorithm, and deep Q -learning. S2V-DQN has been applied to a diverse range of optimization problems over graphs, including Minimum Vertex Cover, Maximum Cut, and Travelling Salesman problems.

1. Model architecture

S2V-DQN adopts a greedy meta-algorithm design, whereby a feasible solution is constructed by the successive addition of nodes based on the current graph structure, which is represented by the graph embedding algorithm, `structure2vec`. The successive addition of nodes is determined by the evaluation function Q , which will be learned with the Q -learning algorithm using a collection of problem instances.

`structure2vec` is a variant of the graph neural network model, which is inspired by graphical model inference algorithms, where node-specific tags or features x_v are aggregated recursively according to the input graph $\mathcal{G}$'s topology. After several rounds of recursions, `structure2vec` produces a p -dimensional feature embedding μ_v for each node, taking into account the graph characteristics and the interactions between node features. `structure2vec` initializes the embedding $\mu_v^{(0)}$ at each node as 0, and for all $v \in \mathcal{V}$ update the embeddings synchronously at each iteration as:

$$\mu_v^{(t+1)} \leftarrow \text{ReLU}(\theta_1 x_v + \theta_2 \sum_{u \in \mathcal{N}(v)} \mu_u^{(t)} + \theta_3 \sum_{u \in \mathcal{N}(v)} \text{ReLU}(\theta_4 w(v, u))) \quad (\text{S23})$$

where $\mathcal{N}(v)$ is the set of neighbors of node v in graph $\mathcal{G}$, $w(v, u)$ is the edge weight of e_{vu} . $\theta_1 \in \mathbf{R}^p$, $\theta_2, \theta_3 \in \mathbf{R}^{p \times p}$, and $\theta_4 \in \mathbf{R}^p$ are the model parameters, and ReLU is the rectified linear unit function applied element-wise to its input.

We can see that the update defines a process where node features x_v are propagated to other nodes via the nonlinear propagation. The more update iterations indicate the farther away the node features will propagate and aggregated nonlinearly at distant nodes. If one terminates T iterations, each node embedding $\mu_v^{(T)}$ contains information about its T -hop neighborhood as determined by the graph topology.

Once the embedding for each node is computed after T iterations, `structure2vec` uses these embeddings to define the $Q(s, a)$, where s qualifies the long-term gain after selecting action a upon state s . More specifically, `structure2vec` uses $\mu_v^{(T)}$ for node v embedding, and the pooled embedding over the entire graph, $\sum_{u \in \mathcal{V}} \mu_u^{(T)}$ as the state embedding, then,

$$Q(s, a; \Theta) = \theta_5^\top \text{ReLU}([\theta_6 \sum_{u \in \mathcal{V}} \mu_u^{(T)}, \theta_7 \mu_v^{(T)}]) \quad (\text{S24})$$

where $\theta_5 \in \mathbf{R}^{2p}$, $\theta_6, \theta_7 \in \mathbf{R}^{p \times p}$, and $[\cdot, \cdot]$ is the concatenation operator.

2. Training algorithm

The computation of Q depends on the collection of 7 parameters $\Theta = \{\theta_i\}_{i=1}^7$. S2V-DQN uses a combination of n-step Q -learning[40] and fitted Q -iteration[47] to update these parameters. More specifically, it updates the parameters by performing stochastic gradient descent over the samples, uniformly sampled from the experience replay buffer $\mathcal{B}$, to minimize the following loss function:

$$(y - Q(s_t, a_t; \Theta))^2 \quad (\text{S25})$$

where $y = \sum_{i=0}^{n-1} r(s_{t+i}, v_{t+i}) + \gamma \max_{a'} Q(s_t, a'; \Theta)$. The experience replay buffer B is populated during previous episodes, such that at step $t + n$, the tuple $(s_t, a_t, r_{t,t+n}, s_{t+n})$ is added to E , with $r_{t,t+n} = \sum_{i=0}^{(n-1)} r(s_{t+i}, a_{t+i})$.

3. Complexity analysis

To construct a solution during the inference phase, S2V-DQN has time complexity of $O(K|\mathcal{E}|)$, where K is the number of greedy steps (at most the number of nodes $|\mathcal{V}|$), and $|\mathcal{E}|$ is the number of edges. For large graphs, K is usually linear to $|\mathcal{V}|$, as such, time complexity of S2V-DQN scales as $O(|\mathcal{V}||\mathcal{E}|)$, which is computationally prohibitive for large-scale networks.

D. FINDER

1. Model architecture

There are three key steps in FINDER: (i) *Encoding*, which learns the nodes' representations, in which the nodes' structural information and auxiliary information (like removal cost) are encoded; (ii) *Decoding*, which learns the scalar Q values for each node, informing us the maximum expected rewards of an action given a state. (iii) *Greedy selection*, which adopts ϵ -greedy strategy in the training phase or batch nodes selection strategy in the application phase, to take actions based on the computation of Q values. In the following, we describe each step in detail.

Encoding: In this step, we seek to map the input graph into a low-dimensional space, where each possible action (i.e., node) and the state (i.e., the residual network) are encoded

into an embedding vector, and these embeddings should capture informative features about the action and the state, so as to be leveraged to compute $Q(s, a)$, which represents the maximum expected rewards of an action a given a state s . The ideal embeddings should capture the graph structural information (and auxiliary information, like node removal costs, if available), and more importantly, the relative position of node a in graph s . Traditional hand-crafted features, such as degree distribution, motif counts, etc., are hard to describe such complex graph information, and thereby we utilized the GNN (Algorithm S2) architecture, which can better incorporate nodes attributes, and is inductive in unseen graphs. Essentially, this kind of neighbor-aggregation encoding process can be seen as the feature propagation that is carried out throughout the graph topology, where each new round of embedding update, that is sweeping across all nodes, starts only when all nodes' last round of updates have finished. If the updates terminate after K iterations, then each node's representation z_v contains K hops away neighbors' information, as determined by the graph topology. As we know that most real-world networks show a "small-world" property[48], indicating that any two nodes are reachable in less than 6 hops. Hence a small value of K would be able to capture the majority of the graph topology information. To represent the entire graph, which is denoted as the state, instead of simply averaging or summing all the nodes' embeddings, we create a virtual node that connects all the nodes (i.e., the neighbors of this virtual node are the entire set of current nodes), to model more complex information of the entire graph. Note that this connection is single directed, which means this virtual node do not appear in the neighbors of other nodes. Then we run K propagations on this $N + 1$ nodes graph, the embedding vector of the virtual node can be utilized as the state embedding.

Decoding: The decoding step aims to learn a mapping from state-action pair (s, a) to a scalar value $Q(s, a)$ that predicts the maximal rewards after taking action a given state s , for which traditionally a nonlinear function is utilized, such as SVM or multi-layered perceptrons (MLPs). Here we parameterize the Q function with the two-layered MLPs employed with rectified linear unit (ReLU) ($\text{ReLU}(z) = z$ if $z > 0$ and 0 otherwise) activation. More specifically, it is defined as below:

$$Q(s, a) = W_5^\top \text{ReLU}(z_a^\top \cdot z_s \cdot W_4) \quad (\text{S26})$$

Here $W_4 \in \mathbf{R}^{p \times 1}$, $W_5 \in \mathbf{R}^{p \times 1}$, are weight parameters, z_s and $z_a \in \mathbf{R}^{1 \times p}$ are the output

embeddings in Algorithm S2, for state and action respectively. Some recent related work[37] usually concatenate the state and action embeddings before feeding them into the MLPs, we argue that the concatenation does not make full use of the relations between state s and action a . Since $[u,v] \cdot [\frac{a}{s}] = ua + vs$, given the same state s , different actions' Q values share the same second term vs , and only differentiate the first one. This means the decisions based on Q values are basically determined by action embeddings only. To model finer dependencies between states and actions, we utilize the outer product operation on the state embedding and action embedding, and apply the MLPs on top of that, to map to a scalar Q value. For more efficient computations, we use an architecture in which there is a separate output unit for each possible action and feed only the state embeddings as the input, to compute Q values for all possible actions under a given state with only a single forward pass through the neural networks[36].

Algorithm S2 Encoding process in FINDER

Input: Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, input features $\{X_v \in \mathbf{R}^{1 \times c}, \forall v \in \mathcal{V}\}$, depth K , weight parameters $W_1 \in \mathbf{R}^{c \times p}, W_2 \in$

$\mathbf{R}^{p \times (p/2)}, W_3 \in \mathbf{R}^{p \times (p/2)}$

Output: Embedding vector $z_v, \forall v \in \mathcal{V} \cup \{s\}$

1: Create a virtual node s which connects all nodes in the graph, denoted as the graph state

2: Initialize $h_v^{(0)} \leftarrow \text{ReLU}(X_v \cdot W_1), h_v^{(0)} \leftarrow h_v^{(0)} / \|h_v^{(0)}\|_2, \forall v \in \mathcal{V} \cup \{s\}$

3: **for** $l = 1$ to K **do**

4: **for** $v \in \mathcal{V} \cup \{s\}$ **do**

5: $h_{\mathcal{N}(v)}^{(l-1)} \leftarrow \sum_{j \in \mathcal{N}(v)} h_j^{(l-1)}$

6: $h_v^{(l)} \leftarrow \text{ReLU}([W_2 \cdot h_v^{(l-1)}, W_3 \cdot h_{\mathcal{N}(v)}^{(l-1)}])$

7: **end for**

8: $h_v^{(l)} \leftarrow h_v^{(l)} / \|h_v^{(l)}\|_2, \forall v \in \mathcal{V} \cup \{s\}$

9: **end for**

10: $z_v \leftarrow h_v^{(K)}, \forall v \in \mathcal{V} \cup \{s\}$

Note that X_s contains all-one vectors of size c , which leads the state embedding to be determined by the graph structure only, and $\mathcal{N}(s)$ contains all nodes in the graph.

Greedy selection: On top of the above two steps, we adopt a greedy selection procedure to take actions based on the nodes Q -values. More specifically, in the training phase, we

use ϵ -greedy strategy which takes the highest Q action with probability $1 - \epsilon$, and take random actions otherwise, ϵ is linearly annealed from 1 to a very small value over a constant number of steps to balance between exploration and exploitation. Large value of ϵ tends to be similar as the random policy which encourages the exploration of the search space, small value of ϵ tends to facilitate the exploitation of the learned Q values. While in the application phase, since the model has been trained well, we simply utilize the greedy policy which take the highest Q -value actions. In our experiments, we found that the performances were empirically unaffected if we remove a finite fraction of nodes at each step, instead of the one-by-one removal in the training phase. As such, we hereby adopt a *batch nodes selection* strategy in the application phase, which takes the top-k nodes with the highest Q at each adaptive step, avoiding the iterative select-and-recomputation of the embeddings and Q values. This strategy leads to several orders of magnitude faster while sacrificing very marginal accuracy.

2. Training algorithm.

There are two sets of parameters to be learned, i.e., encoding parameters $\Theta_E = \{W_1, W_2, W_3\}$, and decoding parameters $\Theta_D = \{W_4, W_5\}$, and we have predicted the Q values for state-action pair (s, a) , i.e., $Q_{\text{pred}}(s, a) = f(s, a; \Theta_E, \Theta_D)$. If given the ground truth $Q_{\text{true}}(s, a)$, we can perform the end-to-end gradient descents on minimizing these two values to update the parameters. Unfortunately, there are no direct labels available in practice.

Eq. S19 indicates an iterative process to compute the optimal function $Q^*(s, a)$, for each iteration, we consider $Q_{\text{pred}}(s, a)$ as $Q_k(s, a)$, and $Q_{\text{true}}(s, a)$ as $r + \gamma \max_{a'} Q_k(s', a')$, thus we are able to define the Q -learning loss to minimize $Q_{\text{pred}}(s, a)$ and $Q_{\text{true}}(s, a)$ (Eq. S20). To deal with the delayed rewards, we wait for n steps before updating parameters, to collect a more accurate estimate of the future rewards, which corresponds to the n -step Q -learning loss (Eq. S22). Besides minimizing the gap between predicted Q values and target Q values, we also seek to preserve the original network structures in the embedding space, to support incorporating structure information in our model, since the key players are highly related to the graph topology.

Therefore, we design the loss function in Eq. S27, which aggregates both Q -learning loss

and graph reconstruction loss, and perform stochastic gradient descents over it to update the model parameters.

$$\begin{aligned} \text{Loss}(\Theta_Q, \Theta_{\text{Embed}}) = & \underbrace{\mathbb{E}_{(s_t, a_t, r_{t,t+n}, s_{t+n}) \sim U(D)} [(r_{t,t+n} + \gamma \max_{a'} \hat{Q}(s_{t+n}, a'; \hat{\Theta}_Q) - Q(s_t, a_t; \Theta_Q))^2]}_{\text{Q-learning loss}} \\ & + \underbrace{\alpha \sum_{i,j=1}^N s_{i,j} \|y_i - y_j; \Theta_{\text{Embed}}\|_2^2}_{\text{Graph reconstruction loss}} \end{aligned} \quad (\text{S27})$$

where $\Theta_Q = \{\Theta_E, \Theta_D\}$, $\Theta_{\text{Embed}} = \Theta_E$. Eq. S27 consists of two parts: (i) Q -learning loss, to minimize the difference between predicted Q values and target Q values; (ii) Graph reconstruction loss[49], to preserve the original network structures in the embedding space. α is a positive hyper-parameter that weighs between these two losses. For the Q -learning loss, we randomly sample mini-batch experiences $(s_t, a_t, r_{t,t+n}, s_{t+n}) \sim U(\mathcal{B})$ from the experience replay buffer $\mathcal{B}$, note that here we use n -step Q -learning updates[37], which waits for n steps to collect a more accurate estimate of the future rewards. γ is the discount factor that determines the importance of future rewards. A factor of 0 will make the agent short-sighted by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward. If it exceeds 1, the action values may diverge. $\hat{\Theta}_Q$ is the target network parameters, which are only updated with Θ_Q every C steps, and are fixed between the individual updates. For the graph reconstruction loss, N is the number of graph nodes, $s_{i,j}$ denotes whether nodes i and j are adjacent, and y_i, y_j are the embedding vectors of node i and j respectively. Techniques of the experience replay buffer and target network are utilized to enable a more stable training process[36].

During the training phase, we use the BA model[50] to generate synthetic graphs for training and validation. To describe this process more clearly, we introduce some terminology: an **episode** is the entire influencer finding (node removal) process on a graph until the terminal state, a **terminal state** can be an user-defined state, e.g., maximum budget nodes or minimum connectivity threshold, a **trajectory** produced in the episode corresponds to a state-action sequence $(s_0, a_0, r_0, s_1, r_1, s_2, \dots, s_T)$, where s_T denotes the terminal state, **PlayGame** is to finish an entire episode on this graph, including generating the trajectory, collecting the transitions $(s_t, a_t, r_{t,t+n}, s_{t+n})$, where $r_{t,t+n} = \sum_{i=t}^{t+n} r_i$, and storing them into the experience replay buffer. During training, we use the ϵ -greedy policy

with ϵ linearly annealing from 1.0 to 0.05 over the 10,000 episodes to balance between exploration and exploitation[40]. In the application phase, we utilize a batch nodes greedy policy, which removes the top-k highest Q nodes at each adaptive step until the game is over. We train the model for a total of one million episodes and use a experience replay buffer of size 50,000 to store the most recent transitions. Every 300 episodes, we evaluate the agent on 200 synthetic graphs with the same scale as the training graphs and average the performances. After each episode, we randomly sample a mini-batch of 4-tuple transitions from the experience replay buffer and perform the stochastic gradient descent updates to minimize the loss function (Eq. S27). Algorithm S3 describes the whole training procedure in detail.

3. Complexity analysis

We analyze the time complexity of FINDER for the training phase and application phase respectively.

Training complexity. During training, the time complexity is proportional to the training iterations, which are hard to be theoretically analyzed. Empirically, FINDER converges (measured by the averaged validation ANC) well on the validation data, as shown in Fig. S11. It took about 50 seconds every 300 training iterations, which was about 40 hours for the total training phase (one million iterations). And for the node-weighted scenario, FINDER converges very quickly on both CN and ND problems (Fig. S11B and Fig. S11D), taking an estimated time of less than 100 iterations (about 1.4 hours). Notably, the training phase is performed only once for one application scenario, and we can utilize the trained model for any input network in the application phase.

Inference complexity. In the application phase, we apply the trained model for a given network. The time complexity of the inference process is determined by three parts. First is from the encoding step. As shown in Algorithm S2, the encoding complexity takes $O(KN\mathcal{N}(\cdot))$, where K is the number of propagation steps, and usually is a small constant (e.g., 5), N is the number of nodes, $\mathcal{N}(\cdot)$ is the average number of node neighbors. In practice, we use the adjacency matrix multiplication for Line 4-8 in Algorithm S2, and due to the fact that most real-world networks are sparsely connected, the adjacency matrix in our setting is processed as a sparse matrix. As such, the encoding complexity in our

implementations turns to be $O(E)$, where E is the number of edges. Second is from the decoding step. Once the graph and nodes are encoded, the quality of nodes can be computed with Eq. S26, thus to compute Q values for all nodes, the time complexity is $O(N)$. The last comes from the greedy selection step. We adopt the batch nodes selection strategy, which takes the top- k nodes with highest Q values at a time, the time complexity of this step takes $O(N\log N)$. Consequently, the total inference time complexity for each adaptive step (including encoding, decoding and selection) is $O(E + N + N\log N)$. Suppose FINDER finishes after n adaptive steps, and in the worst case, $n = N/k$, where $k = \alpha\% * N$ (e.g., $\alpha = 1$), as a result, we can see n basically equals to a constant. Therefore, the total time complexity for the application phase should be $O(E + N + N\log N)$.

Time complexity comparison with other baseline methods. We compare the time complexity of FINDER with that of other baseline methods in Table. S5. For HDA, since we conduct the one-by-one iterative select-and-recompute high degree node, its time complexity is determined by the total number of greedy steps, which in the worst scenario could deteriorate to the total number of nodes in the network. Our FINDER instead selects a batch of nodes each step, rendering several orders of magnitude reduction in time complexity (Table. S7). We also show in Table. S8 that HDA would suffer a great accuracy loss if selected with a batch each time, despite being much faster than the vanilla one-by-one mode. By contrast, FINDER’s performance is practically not affected if selecting a suitable finite fraction of nodes each adaptive step (Table. S7(B)). For MinSum and BPD, or GND, their time complexities are mainly determined by the number of message passing steps or power iteration steps, which can be very large, unfortunately, in order to have a good approximation result.

It should be noted that even with the same time complexity, two algorithms’ practical running time can vary a lot due to the time cost difference on each iteration. Therefore, the wall-clock running time comparison is still useful in practice. We try to keep all the factors, expect for time complexity, the same to ensure the comparisons of algorithms’ wall-clock running time fairly reflect their efficiency differences. For instance, we run all methods in the same computing environment, we do not use GPU-accelerated FINDER since other baseline methods do not have GPU implementations; all implementations are written with C++ language, etc. See Table. S10(B), S11(B), S12(B), S13, S14(B), S15(B), S16(B), S17, and S18(B) for running time comparisons.

In summary, both time complexity analysis and practical running time clearly demonstrate the efficiency of FINDER over conventional methods.

Algorithm S3 Training procedure of FINDER

- 1: Initialize experience replay buffer $\mathcal{B}$ with size M
 - 2: Initialize the action-value function Q with random weights $\Theta = \{\Theta_E, \Theta_D\}$
 - 3: Initialize target Q network with weights $\hat{\Theta} = \Theta$
 - 4: **for** $episode = 1$ to N **do**
 - 5: Generate a random graph with the BA model
 - 6: Initialize the state to an empty sequence $s_1 = ()$
 - 7: **for** $t = 1$ to T **do**
 - 8: With probability ε , select a random action a_t
 - 9: Otherwise select $a_t = \arg\max_a Q(S_t, a; \Theta)$
 - 10: Execute action a_t and observe reward r_t
 - 11: Add a_t to partial solution $s_{t+1} = s_t \cup a_t$
 - 12: **if** $t \geq n$ **then**
 - 13: Store transition $(s_{t-n}, a_{t-n}, r_{t-n,t}, s_t)$ in $\mathcal{B}$
 - 14: Sample a (or mini-batch) random transition $(s_j, a_j, r_{j,j+n}, s_{j+n})$ from $\mathcal{B}$
 - 15: Set $y_j = \begin{cases} r_{j,j+n}, & \text{if } s_{j+n} \text{ is the terminal state} \\ r_{j,j+n} + \gamma \max_{a'} Q(s_{j+n}, a'; \hat{\Theta}), & \text{otherwise} \end{cases}$
 - 16: Perform stochastic gradient descents over (Eq. S27) for the network parameters Θ
 - 17: Every C steps reset $\hat{\Theta} = \Theta$
 - 18: **end if**
 - 19: **end for**
 - 20: **end for**
-

E. Key differences between S2V-DQN and FINDER

1. Key differences

The framework of S2V-DQN is similar to ours, and it claims superior performances on many graph combinatorial optimization problems, like Minimum Vertex Cover, MAXCUT,

Traveling Salesman Problem and Set Cover Problem. However, it has not been systematically tested on real-world networks. Moreover, it has not been compared with state-of-the-art approximate or heuristic methods that were designed specifically for each combinatorial optimization problem.

Overall, FINDER differs from S2V-DQN in the encoding step (including both states and actions), decoding step and greedy selection step. The training objective is also different. More specifically, there are five aspects:

(i) In the encoding step, our graph embedding model (Algorithm S2) is more similar to **GraphSAGE**[30] (although differs in the input features and the neighborhood definition) rather than structure2vec[31]. In Algorithm S1, we can see that for each iteration, structure2vec simply adds nodes’ initial features and neighbor embeddings, while GraphSAGE concatenates the self-embedding from the previous iteration and neighbor embeddings. Extensive experiments indicate that concatenation leads to significant gains in capturing graph structures[30].

(ii) We create a **virtual node** to represent the entire graph, while S2V-DQN simply sums over all the nodes. We claim that the virtual node could inherently capture more complex graph information than the traditional sum pooling method.

(iii) For decoding, we compute Q values via **cross product** to model finer state-action dependencies. S2V-DQN simply concatenates the state embedding and action embedding before feeding them into the MLPs. We argue the concatenation operation weakens the contribution of current state in the computation of Q values.

(iv) During the greedy selection step, we adopt the **batch nodes selection** action strategy, which picks a finite fraction (e.g., 1%) of highest- Q nodes at each adaptive step, rather not the one-by-one iterative select-and-recompute of the embeddings and Q values. The strategy doesn’t affect the final result, but it renders several orders of magnitude reduction in the time complexity (Fig. S2, Fig. S3, Table S6, Table S7).

(v) During training, we not only consider the traditional Q -learning loss (S2V-DQN), but also the **graph reconstruction loss**. The Q -learning loss enables the learned node embeddings to minimize the gap between the predicted Q values and target Q values, while neglects their roles in preserving the graph structure in the embedding space. Although the GNNs architecture inherently maintains the local neighborhood structure, we claim there is still necessary to add a graph reconstruction loss, through which the obtained

embeddings could not only precisely predicts the Q values but also preserves the graph topology information, since the key players in our problem are highly determined by the network structures.

2. Ablation study

To explicitly understand the contributions of the above differences in the performance enhancement, we perform the ablation experimental analysis by creating different FINDER variants, with replacing one component each time while keeping the rest unchanged, i.e., *No-GraphSAGE*, *No-VirtualNode*, *No-CrossProduct*, and *No-ReconsLoss*.

To provide a more fair comparison with S2V-DQN¹, we replace the rewards in S2V-DQN with network connectivity measures, i.e, σ_{pair} for the CN problem and σ_{gcc} for the ND problem, and utilize the *batch nodes selection* strategy to scale it for large-scale networks. We choose the best hyperparameter settings for S2V-DQN via validation performances. The comprehensive ablation results on both CN problem and ND problem are shown in Table S2. We can see FINDER is superior to S2V-DQN to a large extent on most networks, for both CN problem and ND problem. All the differences between FINDER and S2V-DQN help to improve the performances, among which the GraphSAGE embedding contributes the most, and followed by the graph reconstruction loss.

III. DESCRIPTIONS OF EXPERIMENTAL DATASETS

We compare FINDER with conventional methods on both synthetic graphs and real-world networks.

We used the Barabási-Albert (BA) model[50] to generate synthetic graphs for the test purpose. We controlled the generator to distribute the generated graphs into six different ranges of node set sizes, i.e., 30-50, 50-100, 100-200, 200-300, 300-400, 400-500. Within each range, we randomly drew 100 graph instances and aggregated the average performance for each method.

We also chose nine real-world networks with different scales, which cover a wide range of domains, including criminal networks, biological networks, communication networks,

¹ https://github.com/Hanjun-Dai/graph_comb-opt

infrastructure networks, and social networks. Specifically, they are:

Crime[51], a criminal network obtained from the projection of a bipartite network of persons and crimes, each node denotes a person, an edge represents that two persons are involved in the same crime.

HI-II-14², a human protein-protein interaction network, where each node represents a protein, each edge denotes the interaction between the corresponding proteins.

Digg[51], a reply network of the social news website Digg. Each node in the network is a user of the website, and each edge denotes that a user replied to another user.

Enron[4], the Enron email communication network that covers all the email communication within a dataset of around half a million emails. Each node is an email address, and an edge denotes at least one email communication.

Gnutella31[52], a sequence of snapshots of the Gnutella peer-to-peer file-sharing network from August 2002. Nodes represent hosts in the Gnutella network topology and edges represent connections between the Gnutella hosts.

Epinion[51], the trust network of the online social network Epinions. Nodes are users of Epinions and directed edges represent trust between the users.

Facebook[51], contains friendship data of a small subset of Facebook users. A node represents a user and an edge represents friendship between two users.

Youtube[51], the friendship network of the video-sharing site Youtube. Nodes are users and an edge between two nodes indicates friendship.

Flickr[51], the social network of Flickr users and their connections.

Besides, we use two real networks in the main text for illustrations, they are: i) **Krebs**[53] in Fig. 1 and Fig. 2, which is a 9/11 terrorist network, containing 62 nodes and 159 edges. Nodes represent terrorists involved in the 9/11 attack, and edges represent their social communications. ii) **mammalia-raccoon**³ in Fig. 3, which is a real-world animal interaction network, containing 14 nodes and 20 edges (GCC). Each node denotes a mammal raccoon, each edge represents their interactions, which is defined as the close proximity (within 1-1.5m), and any contacts < 1 s in duration were excluded.

We treat all the networks as undirected ones and remove all the self-loops. We extract the largest connected component as the test data sets. The basic statistics of the extracted

² [http://interactome.baderlab.org/data/Rolland-Vidal\(Cell2014\).psi](http://interactome.baderlab.org/data/Rolland-Vidal(Cell2014).psi)

³ <http://networkrepository.com/mammalia-raccoon-proximity-21.php>

networks are reported in Table S3. We also plot the degree distributions of these networks in Fig. S1, which indicates most real networks pose the property of degree heterogeneity.

The above networks are all node-unweighted. To simulate the node-weighted scenario, we assign each node a weight value. Here we consider two types of weights: (i) **degree weight**: $c(i) = d(i)$, where $d(i)$ is node i 's degree; (ii) **random weight**, where cost is defined as:

$$c(i) = 0.5 * (d(i) + \bar{d} \epsilon) \quad (\text{S28})$$

here $\bar{d}$ is the median value of the degree distribution, ϵ is a random variable drawn from a normal distribution with mean 0 and variance 1.

IV. MORE DETAILS

A. Hyperparameters

All experiments were conducted on a 20-core server with 512GB memory and a 16GB Tesla V100 GPU. We generated small scale (30-50 nodes) BA graphs as the training set and validation set. We randomly generated 200,000 synthetic networks for training and 200 instances for validation. The model was implemented with Tensorflow 1.8 and was trained with Adam optimizer. Hyperparameters (Table S4) were tuned according to the performance on the validation set. The training process terminated with early stopping based on the validation set. The trained model and the implementation codes will be available upon requests.

For the baselines implementations, we ran HDA, HBA, HCA, and HPRA with SNAP⁴, a general-purpose, high-performance system for graph analysis. For CI⁵, MinSum⁶, BPD⁷, CoreHD⁸ and GND⁹, we used the source codes released online, and adopted the best parameter settings for each method. For RatioCut, we modified the traditional RatioCut method based on the GND implementation⁹.

Note that most existing baseline methods do not have GPU implementations, while FINDER can be easily GPU-accelerated. To have a fair execution time comparison with

⁴ <http://snap.stanford.edu/snap/>

⁵ <https://github.com/zhfkt/ComplexCi>

⁶ <https://github.com/abraunst/decycler>

⁷ <http://power.itp.ac.cn/zhohj/codes.html>

⁸ <https://github.com/hcmidt/corehd>

⁹ <https://github.com/renxiaolong/Generalized-Network-Dismantling>

baseline methods, here we did not deploy GPU-acceleration for FINDER in the application phase, but only utilized GPU to speed up the offline training phase.

B. The range of ANC

Theorem IV.1. *Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the node-set, $\mathcal{E}$ is the edge-set, and $|\mathcal{V}| = N$. Let R be the accumulated normalized connectivity (ANC), then we have $R_{\text{CN}} \in [0, \frac{1}{3}(1 - \frac{2}{N})]$, and $R_{\text{ND}} \in [\frac{1}{N}, \frac{1}{2}(1 - \frac{1}{N})]$, for the critical node (CN) problem and network dismantling (ND) problem respectively.*

Proof. Among all networks of N nodes, we know the most vulnerable network is the star, where removing the central hub can lead to the complete fragmentation; whereas the most robust type is the fully-connected network, in which all nodes have to be removed for the complete demolition. These two types correspond to the smallest and largest value of R respectively, for both CN and ND problems.

For the star network, it would be broken into isolated nodes if we first remove the central hub, then we have:

$$\begin{aligned} R_{\text{CN}}^{\text{star}}(v_1, v_2, \dots, v_N) &= \frac{1}{N} \sum_{k=1}^N \frac{\sigma_{\text{CN}}(\mathcal{G} \setminus \{v_1, v_2, \dots, v_k\})}{\sigma_{\text{CN}}(\mathcal{G})} \\ &= \frac{1}{N} \frac{\sum_{k=1}^N 0}{N(N-1)/2} = 0 \end{aligned}$$

where $\sigma_{\text{CN}}(\mathcal{G} \setminus \{v_1, v_2, \dots, v_k\}) = \sum_{C_i \in \mathcal{G}[\mathcal{V} \setminus \mathcal{K}]} \frac{\delta_i(\delta_i - 1)}{2}$, and C_i is the i -th connected component in the remaining graph $\mathcal{G}[\mathcal{V} \setminus \mathcal{K}]$ after removing the node-set $\mathcal{K} = \{v_1, v_2, \dots, v_k\}$, δ_i is the size of C_i .

$$\begin{aligned} R_{\text{ND}}^{\text{star}}(v_1, v_2, \dots, v_N) &= \frac{1}{N} \sum_{k=1}^N \frac{\sigma_{\text{ND}}(\mathcal{G} \setminus \{v_1, v_2, \dots, v_k\})}{\sigma_{\text{ND}}(\mathcal{G})} \\ &= \frac{1}{N} \frac{\sum_{k=1}^N 1}{N} = \frac{1}{N} \end{aligned}$$

where $\sigma_{\text{ND}} = \max\{\delta_i; C_i \in \mathcal{G}[\mathcal{V} \setminus \mathcal{K}]\}$, δ_i and C_i are the same as the above.

For the fully connected graph, any node removal sequence achieves the same fragmenta-

tion, i.e., decreases linearly as the node removal. As such, we have:

$$\begin{aligned}
R_{\text{CN}}^{\text{complete}}(v_1, v_2, \dots, v_N) &= \frac{1}{N} \sum_{k=1}^N \frac{\sigma_{\text{CN}}(\mathcal{G} \setminus \{v_1, v_2, \dots, v_k\})}{\sigma_{\text{CN}}(\mathcal{G})} \\
&= \frac{1}{N} \frac{\sum_{k=1}^N (N-k)(N-k-1)/2}{N(N-1)/2} \\
&= \frac{1}{N^2(N-1)} \sum_{k=1}^N [N^2 - N - (2N-1)k + k^2] \\
&= \frac{1}{N(N-1)} \frac{(N-1)(N-2)}{3} = \frac{1}{3} \left(1 - \frac{2}{N}\right)
\end{aligned}$$

And,

$$\begin{aligned}
R_{\text{ND}}^{\text{complete}}(v_1, v_2, \dots, v_N) &= \frac{1}{N} \sum_{k=1}^N \frac{\sigma_{\text{ND}}(\mathcal{G} \setminus \{v_1, v_2, \dots, v_k\})}{\sigma_{\text{ND}}(\mathcal{G})} \\
&= \frac{1}{N} \frac{\sum_{k=1}^N (N-k)}{N} \\
&= \frac{1}{N^2} \frac{N(N-1)}{2} = \frac{1}{2} \left(1 - \frac{1}{N}\right)
\end{aligned}$$

As a consequence, $R_{\text{CN}} \in [R_{\text{CN}}^{\text{star}}, R_{\text{CN}}^{\text{complete}}] = [0, \frac{1}{3}(1 - \frac{2}{N})]$, $R_{\text{ND}} \in [R_{\text{ND}}^{\text{star}}, R_{\text{ND}}^{\text{complete}}] = [\frac{1}{N}, \frac{1}{2}(1 - \frac{1}{N})]$. $\square$

Theorem IV.2. *Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the node-set, $\mathcal{E}$ is the edge-set, and $|\mathcal{V}| = N$. Let R_{cost} be the weighted accumulated normalized connectivity (ANC), each node v_k comes with a different cost $c(v_k) \in (0, 1)$, and $\sum_{k=1}^N c(v_k) = 1$. We have $R_{\text{CN}}^{\text{cost}} \in (0, 1 - \frac{2}{N})$, and $R_{\text{ND}}^{\text{cost}} \in (\frac{1}{N}, 1 - \frac{1}{N})$, for the critical node (CN) problem and network dismantling (ND) problem respectively.*

Proof. For both CN and ND problems, since $c(v_k) \in (0, 1)$, we have:

$$\begin{aligned}
R_{\text{cost}} &= \sum_{k=1}^N \frac{\sigma(\mathcal{G} \setminus \{v_1, v_2, \dots, v_k\})}{\sigma(\mathcal{G})} c(v_k) \\
&\in (f_{\min}(v_1, v_2, \dots, v_k), f_{\max}(v_1, v_2, \dots, v_k))
\end{aligned}$$

where $f(v_1, v_2, \dots, v_k) = \sigma(\mathcal{G} \setminus \{v_1, v_2, \dots, v_k\})/\sigma(\mathcal{G})$.

For CN, we have:

$$\begin{aligned}
f_{\text{CN}}^{\min} &= \frac{\sigma_{\text{CN}}(\mathcal{G} \setminus \{v_1, v_2, \dots, v_k\})_{\min}}{\sigma_{\text{CN}}(\mathcal{G})} = \frac{0}{N(N-1)/2} = 0 \\
f_{\text{CN}}^{\max} &= \frac{\sigma_{\text{CN}}(\mathcal{G} \setminus \{v_1, v_2, \dots, v_k\})_{\max}}{\sigma_{\text{CN}}(\mathcal{G})} = \frac{(N-1)(N-2)/2}{N(N-1)/2} = 1 - 2/N
\end{aligned}$$

For ND, we have:

$$f_{\text{ND}}^{\min} = \frac{\sigma_{\text{ND}}(\mathcal{G} \setminus \{v_1, v_2, \dots, v_k\})_{\min}}{\sigma_{\text{ND}}(\mathcal{G})} = \frac{1}{N} = 1/N$$

$$f_{\text{ND}}^{\max} = \frac{\sigma_{\text{ND}}(\mathcal{G} \setminus \{v_1, v_2, \dots, v_k\})_{\max}}{\sigma_{\text{ND}}(\mathcal{G})} = \frac{N-1}{N} = 1 - 1/N$$

As a result, $R_{\text{CN}}^{\text{cost}} \in (f_{\text{CN}}^{\min}, f_{\text{CN}}^{\max}) = (0, (1 - \frac{2}{N}))$, $R_{\text{ND}}^{\text{cost}} \in (f_{\text{ND}}^{\min}, f_{\text{ND}}^{\max}) = (\frac{1}{N}, (1 - \frac{1}{N}))$. $\square$

C. Effects of the fraction of removed nodes at each adaptive step

As is discussed before, our model’s inference complexity is $O(nKE)$, where n is the number of greedy selection steps, K is the number of neighbor aggregation iterations, E is the number of edges. In the worst case, n equals the number of nodes N . In our experiments, we found that the performance was practically unaffected if we remove a finite fraction (α) of nodes each step, instead of the one-by-one removal in the training phase. With this *batch nodes selection* strategy, the time complexity of FINDER can reduce to $O(E + N + N \log N)$. The effects of different values of *StepRatio* (fraction of fixed nodes at each step) on real-world networks for both CN problem and ND problem are shown in Table S6, Fig. S2 and Table S7, Fig. S3, respectively. We found that, the performances are nearly unaffected by the removal of up to 1% of nodes. In our experiments, we let FINDER remove 1% nodes at each adaptive step throughout the inferences on real networks, and compare the results with other methods.

D. Several ways to further improve FINDER’s performances

There are several ways to further improve FINDER’s performances, implying that the results presented in the main text are rather conservative.

1. Tailor the training data to the target network

Currently, we simply generate BA synthetic graphs as the training data, can we further improve FINDER’s performances via tailoring the learning set towards the target network of interest? To this end, we utilize the Configuration Model (CM)[54], which offers greater

flexibility to create model networks with arbitrary degree distributions. For illustration purpose, here we show the results for the ND problem only.

To systematically test the idea of CM-trained FINDER, we performed extensive calculations. Note that in this work all the training graphs are much smaller than that of real ones. To implement the idea of CM-trained FINDER, we first calculate the degree distribution $P(k)$ from large-scale real graph first, and then use $P(k)$ to guide us to generate small-scale training graphs. More specifically, we first calculate the degree distribution $P(k)$ for each real-world network, where each degree k is normalized by the maximum degree to be in $(0, 1]$; then we sample degrees for a number of times (e.g., 30-50), and each degree is sampled with a probability that is proportional to its frequency in $P(k)$. Note that the sampled degrees are the normalized values, we multiply them with a constant value, denoted as the expected maximum degree in the sampled sequence, chosen that it assures the mean degree for the sampled degree sequence and original degree sequence are the same. Finally, we fine-tune the sampled degrees to make it a valid sequence input to CM (e.g., the sum of degrees should be even), and use CM to output a synthetic graph that is expected to have a similar degree distribution as the target real network.

For each real network, we generate smaller synthetic training graphs in the above way, and train FINDER on them for the ND problem. The ANC score for each network is shown in Table. S22. We find that most test networks benefits from this kind of training, especially for those that do not show clear power-law degree distributions (e.g., Gnutella31 and Facebook, see Fig. S1 for their degree distribution plots), they tend to obtain a more obvious ANC gain.

These new results suggest that **the superior performance of BA-trained FINDER presented in our main text is really conservative**. It definitely can be further improved by using CM-trained graphs. But the advantage of BA-trained FINDER is that it needs to be trained only once, and then could be applied multiple times, making it a more practical option than the CM-trained FINDER .

2. Enhanced with the NEP algorithm

For the network dismantling problem, there exists a well-known refinement technique, named the node explosive percolation (NEP) algorithm[55], a.k.a., inverse target[56] or

reinsertion technique[7]. It was firstly proposed as a novel effective immunization or destruction strategy. A recent work[57] noticed that some percolation-based algorithms, like MinSum or BPD, can be enhanced by NEP, to minimize the ANC score. The hybrid algorithm starts from the percolation point, where the network consists of fragmented small components with sub-extensive sizes after removing the set of nodes detected by MinSum or BPD, then it restores one of the removed node with its attached edges, chosen such that, once reinserted, it joins the smallest number of clusters. When the node is reinserted, restore the edges with its neighbors that are in the network (but not the ones with neighbors not yet reinserted, if any). Repeat the above procedure until all the removed nodes are added back to the network. Finally, the hybrid algorithm return a dismantling process in the inverse order of the restoring process, which would reduce GCC size as quickly as possible.

We emphasize that the NEP algorithm can be utilized as a refinement technique for all dismantling methods. In Table. S9, we compare the performance of different methods and that of their NEP-enhanced ones. We find that the NEP algorithm did consistently decreases the ANC score of the original methods, and FINDER_NEP yields the best results over all methods, including both NEP-enhanced or original methods. In Fig. S12, we further show that the ANC score decrease is due to the reordering of the early part of the ANC curve, which makes it much lower than before.

E. FINDER’s performance on spatial networks

Most real networks analyzed in this work have heterogeneous degree distribution, similar to the BA model which was used in training. We wonder if FINDER will also work well if applied to networks without such characteristic, e.g., spatial networks, like power-grid networks.

To test if FINDER works well for spatial networks such as the power grid, we considered the following two power-grid (PG) networks:

Degree distributions for these two networks are shown in Fig. S1. FINDER’s results on these two networks are shown in Table. S19. As shown in Table. S19(A), BA-trained (and even CM-trained) FINDER does not yield the best result. They cannot beat GND, a normalized cut algorithm. Maybe for these sparse networks, the most effective disintegrate strategy is to iteratively break the bridging edges that connect different components.

Name	Nodes	Edges	Description
PG-WestState ^a	4,941	6,594	Power grid in Western States of US
PG-Texas ^b	4,889	5,855	Power grid in Texas

^a D. J. Watts and S. H. Strogatz, Nature 393, 440-442 (1998)

^b G. Bianconi, N. Gulbahce, and A. E. Motter Phys. Rev. Lett. 100, 118701 (2008)

Interestingly, if we enhance FINDER with the NEP algorithm, our results would beat all NEP-enhanced baseline methods (Table. S19(B)). The results suggest that for certain sparse and spatial networks (such as power grid) a refined FINDER is needed.

F. FINDER’s application to the minimal percolation threshold problem

In some applications, we are interested to disintegrate the network by removing the smallest number of nodes. In this case, the intermediate values of ANC are not as important as the total number of steps (node removals) necessary to drive ANC sufficiently close to zero. This problem is also referred to as the minimal percolation threshold problem or the vanilla network dismantling problem. Here we will show compelling evidence that our framework can be easily adapted to this case without making too many changes.

Before we show the successful story, we want to share some lessons we learned from failures. At first, we simply changed the reward to be -1, and considered terminal state as the network broken into GCC of 1% size. We followed the training schema of FINDER, i.e., train on small synthetic graphs, and test the learned model on large real-world networks. We used the fraction of node removals as the evaluation metric. However, we found the models trained in this way cannot perform well on different scale instances than they were trained on, since the terminal state (1% size) may vary largely in different scale graphs, e.g., GCC of 1% size represents single nodes for graphs with 100 nodes, while represents components of 10 nodes for graphs with 1000 nodes. Intuitively, the agent would feel very “confused” if it was trained on small scale graphs and then tested on larger instances, since it comes across a totally different terminal state when the game ends. As a consequence, this would greatly hinder the model’s generality.

Fortunately, we later noticed the close relationship between network dismantling and network decycling[9–11]. Network decycling aims to remove the minimal set of nodes to destroy all loops in the network (so that the residual network has tree-like topology). Once achieving that, the network can be considered as completely fragmented, since any tree can be destroyed with a sub-extensive number of node removals (provided that we remove the root nodes each step). In graph theory, *it is easy to see that a graph contains cycles if and only if its 2-core is nonempty*. Therefore, to remove the minimum nodes to destroy all loops in one graph, we actually only need to remove minimum nodes to empty its 2-core, and the terminal graph state without 2-core looks similar across different scale graphs — they all contain no loops or have tree-like topology. Training FINDER in this way makes it generalize well on different scale instances.

As illustrated in Fig. S13, building upon the vanilla framework, we define the graph state in this problem as the 2-core of the residual graph, rather than the previous whole residual graph; the terminal state is set to be the absence of 2-core. For the reward function, we set as -1 at each step, since reinforcement learning aims to maximize the cumulative rewards while we want to minimize the node removals for this task. Those are the only changes made to FINDER.

We follow the previous training procedure to update the parameters to improve its ability in destroying loops in a graph. When applied to dismantle a graph, we firstly apply the well-trained model to iteratively remove nodes to destroy all the loops; after that, considering that some of the residual tree components may still be larger than the desired threshold, we break them into small components by greedily removing the root nodes, and then reinsert some back that would close cycles without increasing too much the size of the largest component. See [9] for efficient algorithms of tree-breaking and reinsertion.

Similarly, we trained the model on random graphs generated from the Barabási-Albert (BA) model, and tested it on the same nine real-world networks. We compared the percolation threshold, i.e., fraction of nodes’ removal that leads to the 1% residual GCC size, and the running time of different methods. As clearly shown in Table. S18, FINDER outperformed most baselines for this task, especially those specifically designed for this task, e.g., BPD and CoreHD. FINDER is just slightly worse than MinSum (Table. S18(A)), which achieves the current best results. However, we emphasize that FINDER is much more efficient than MinSum, especially on very large networks

(Table. S18(B)).

Although there may be other ways to further improve FINDER for this task, FINDER’s potential in handling the minimal percolation threshold problem has been clearly demonstrated here. We notice that there is a related long-standing NP-hard problem — *the k -core percolation or minimal contagious set problem*, which aims to remove the minimal set of nodes to destroy the k -core in a graph. It has been proven that destroying the k -core is equivalent to the activation of the entire graph under the linear threshold dynamics with each node’s threshold as $\theta_i = d_i - k + 1$, where d_i is its degree, k is the core number[58]. Since FINDER could handle the 2-core case, we believe it can be easily adapted to the general k -core case and hence provide a promising approach to the minimal contagious set problem.

G. Spreading power of the nodes selected by FINDER in influence maximization problems

Morone *et al.*[7] have proved that the optimal attack problem with minimizing GCC size is dual to the optimal spreading problem with specific linear threshold spreading dynamics, under which each node’s threshold is $\theta_i = d_i - 1$, where d_i is its degree. Under this spreading dynamic, a node switches from inactive to active only if it has $d_i - 1$ or more active neighbors, or equivalently, a node remains inactive only if it has 2 or more inactive neighbors. Here we demonstrate this claim with experimental results.

We compare the spreading power of the nodes selected by FINDER with that of nodes selected by a greedy algorithm developed by Kempe *et al.*[59], i.e., the Hill-Climbing algorithm. This algorithm simply activates the node that gives the largest marginal gain at each step, where the marginal gain is defined as the total number of active nodes when the spreading process converges after activating this node. Kempe *et al.* proved its solution is provably within 63% of the optimal for several spreading models. Here we simply utilize it under the above specific spreading model.

In Fig. S14, we plot the curve with x-axis being the fraction of seeds to be activated, and y-axis being the fraction of nodes that remain inactive after the dynamical process has converged. Note that it is very time consuming to run the Hill-Climbing algorithm on large networks. Hence, we only tested the Crime network for illustration purpose. The solution

obtained by FINDER is using the agent designed to solve the minimal percolation threshold problem (see IV F).

As is shown in Fig. S14, FINDER needs only 110 nodes to activate the entire graph, while the Hill-Climbing algorithm needs 117 nodes. The results clearly demonstrates the strong spreading power of the nodes selected by FINDER for the optimal spreading problems under specific dynamics. Note that since FINDER only seeks to minimize the seeds to activate the whole graph, while Hill-Climbing maximizes node’s marginal spreading gain each step, hence Hill-Climbing produces an overall lower curve. We can enable FINDER to learn to produce the curve as low as possible via setting the rewards to minimize the area under this curve, just like what we did in minimizing Equation (1).

For other spreading problems with more complicated spreading dynamics, we can either attempt to map them to a network attack problem that we have already addressed here, or define a new reward with regard to the specific spreading dynamic and then train the agent under similar framework, both of them deserve further explorations, which, however, are beyond the scope of the current work.

In summary, there exists a very close relationship between the optimal attack problems and the optimal spreading problems. The current FINDER framework could handle the general class of network attack problems, and one particular spreading problem. There remains potential for FINDER in solving more general spreading problems.

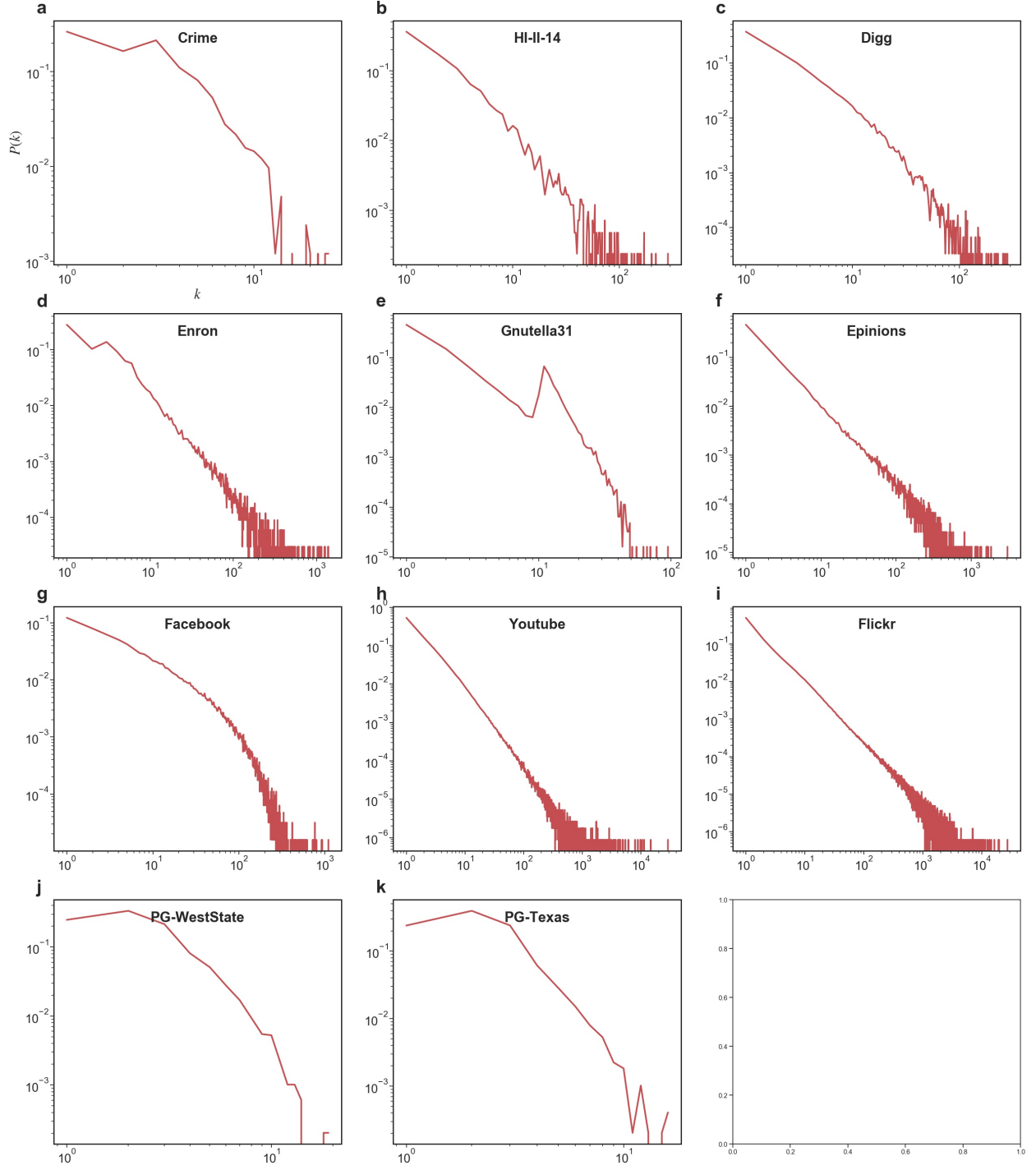


Figure S1: Degree distribution of the real-world networks analyzed in this work. Here k denotes the node degree, $P(k)$ indicates the density of nodes corresponding to degree k .

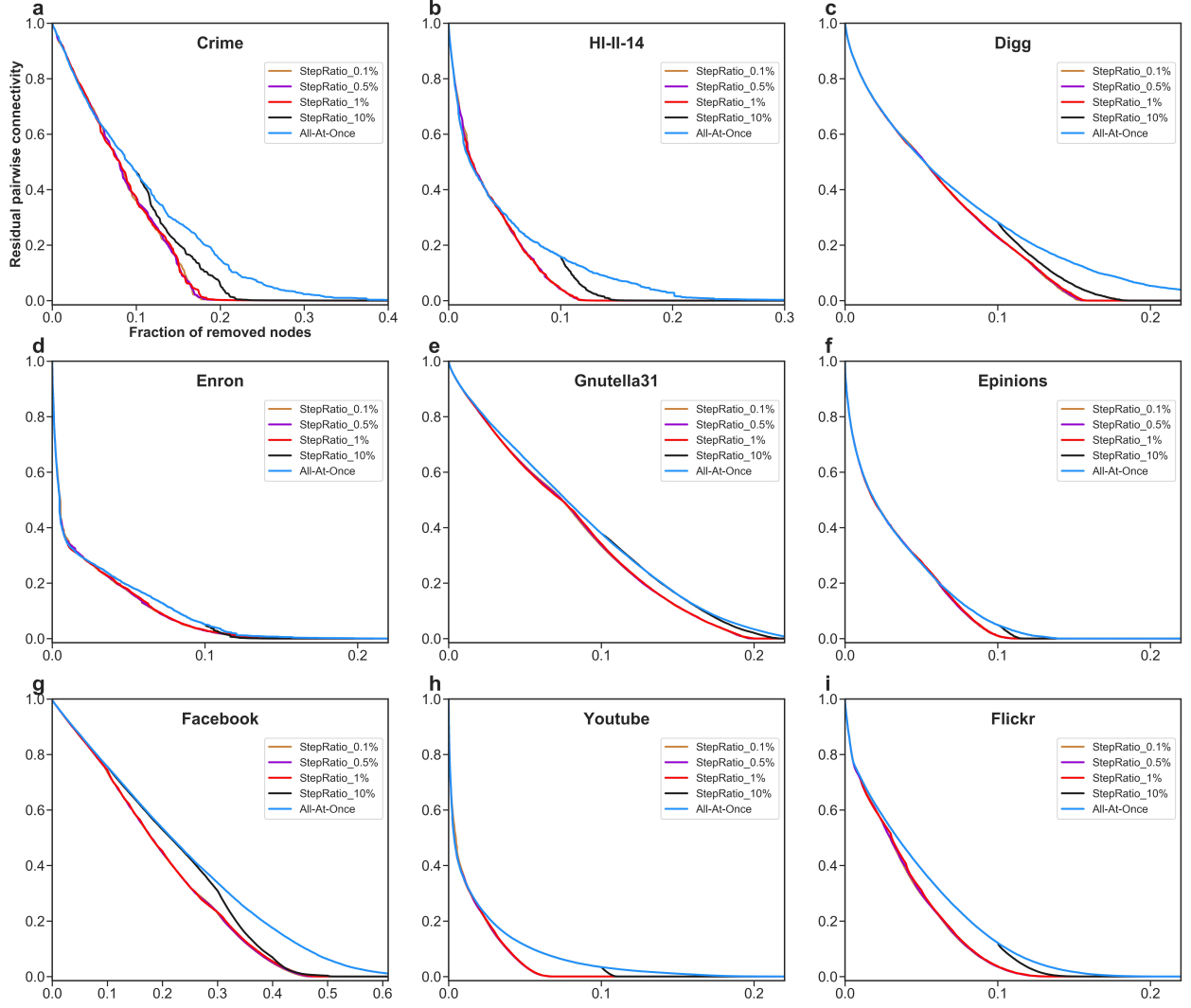


Figure S2: **Different percentages of nodes removed at each step - CN problem.**

ANC curves on nine real-world networks for the critical node problem, under different percentages of nodes removed at each step. StepRatio_ q denotes the removal of qN nodes according to the descending order of Q values calculated at each step.

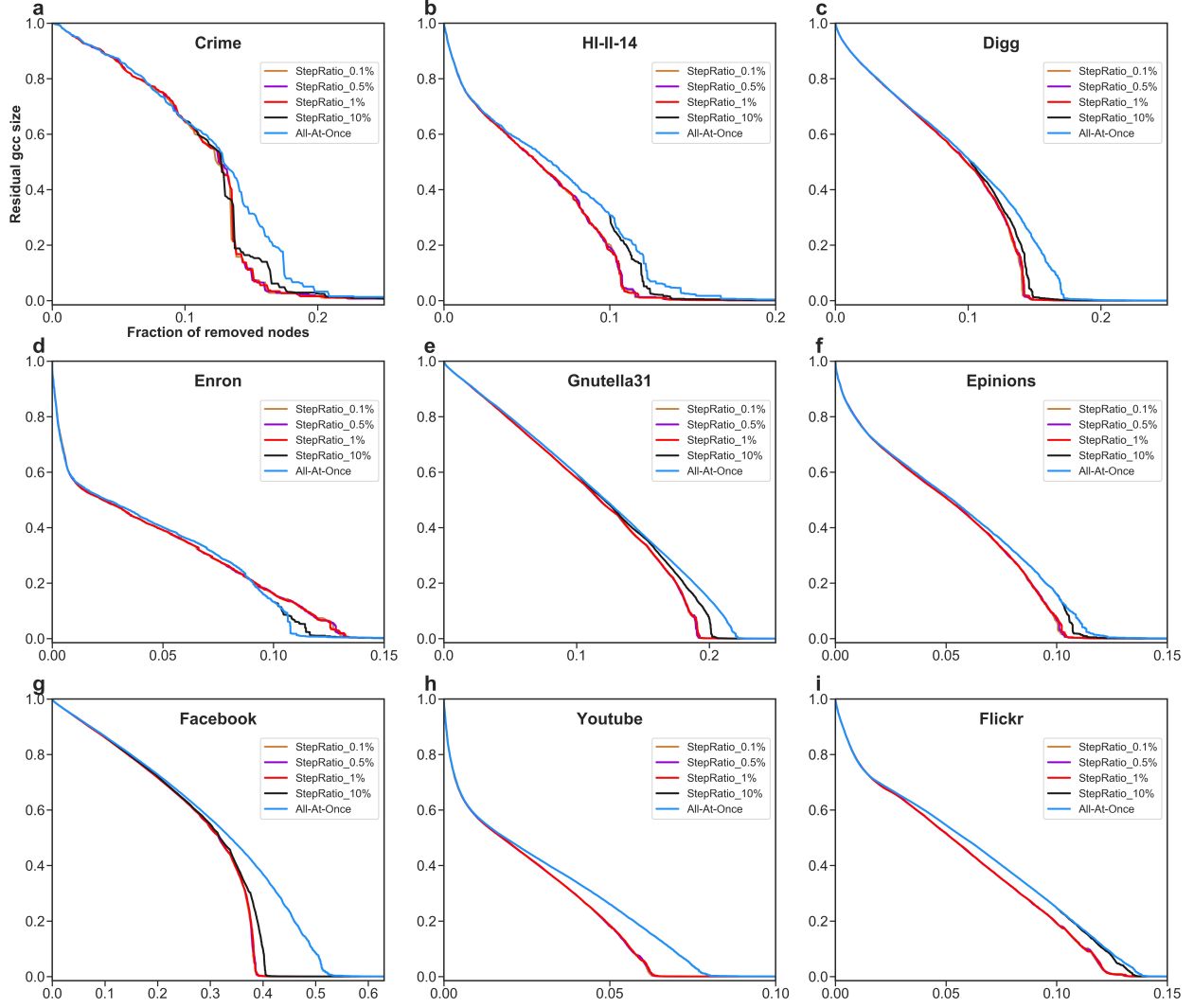


Figure S3: **Different percentages of nodes removed at each step - ND problem.** ANC curves on nine real-world networks for the network dismantling problem, under different percentages of nodes removed at each step. StepRatio- q denotes the removal of qN nodes according to the descending order of Q values calculated at each step.

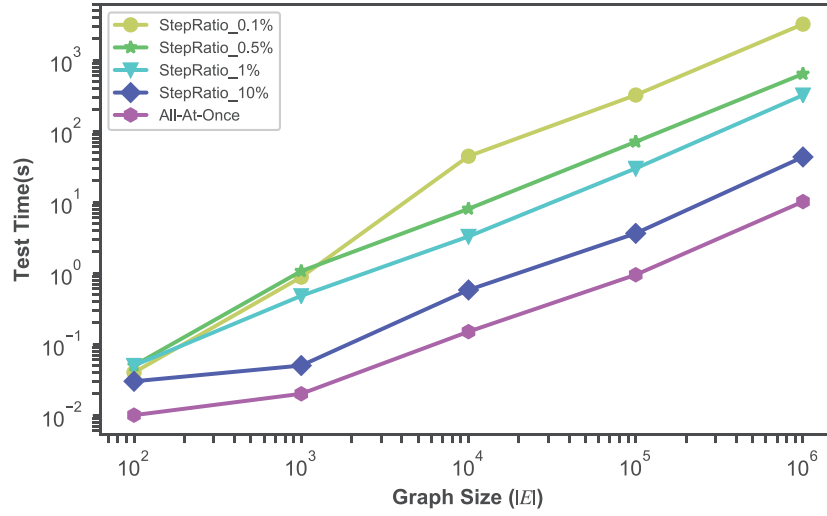


Figure S4: **Scalability analysis.** We evaluate FINDER on synthetic BA graphs of different sizes, ranging from 100 nodes(edges) to 10 million nodes(edges), and plot the running time for each size. The curve shows a linear trend with respect to the number of edges. Different StepRatios are also evaluated, and larger StepRatio yields significantly shorter running time.

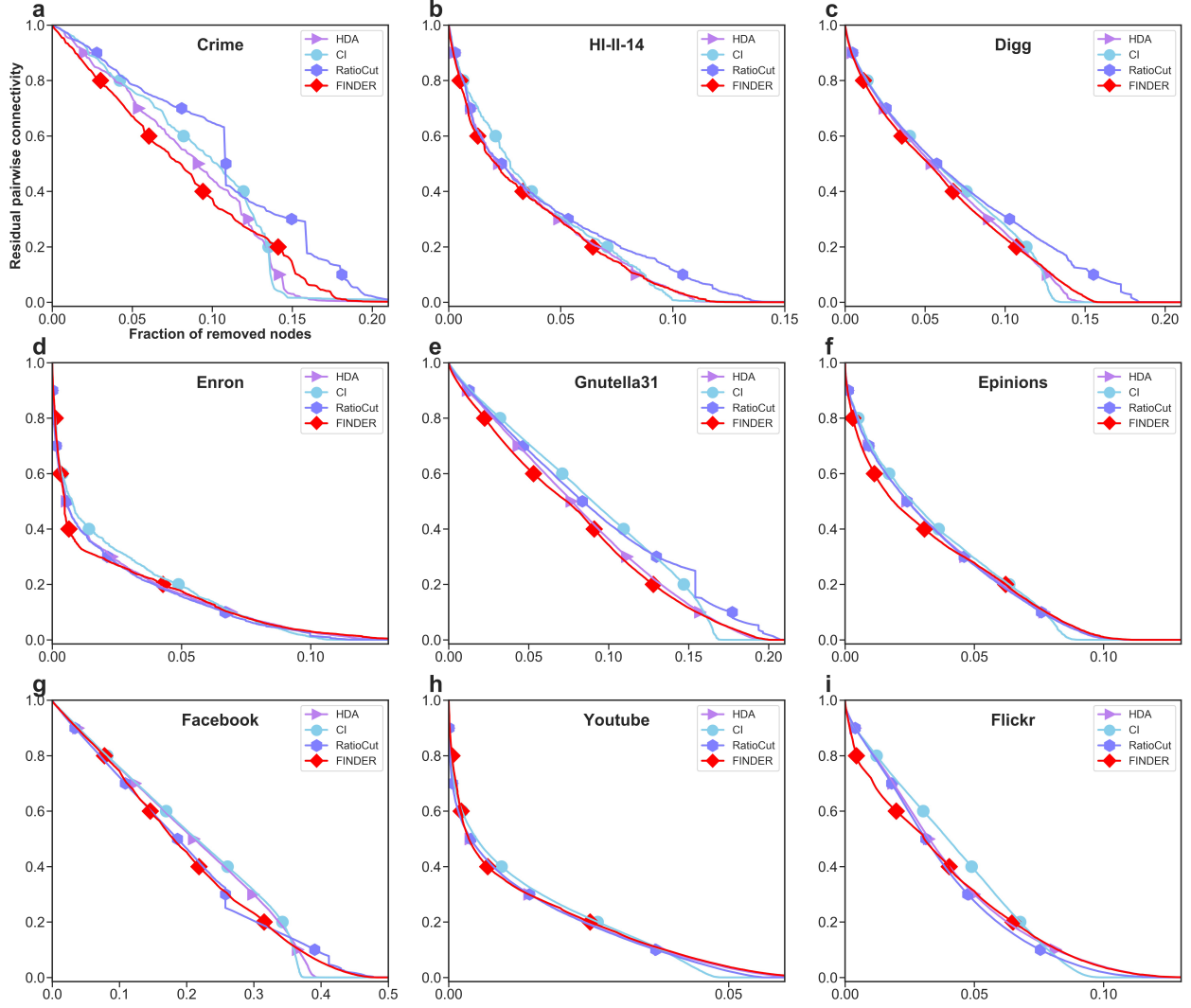


Figure S5: ANC curves on real-world networks (CN, node-unweighted). The horizontal axis is the fraction of the removed nodes, and the vertical axis is the pairwise connectivity of the residual graph.

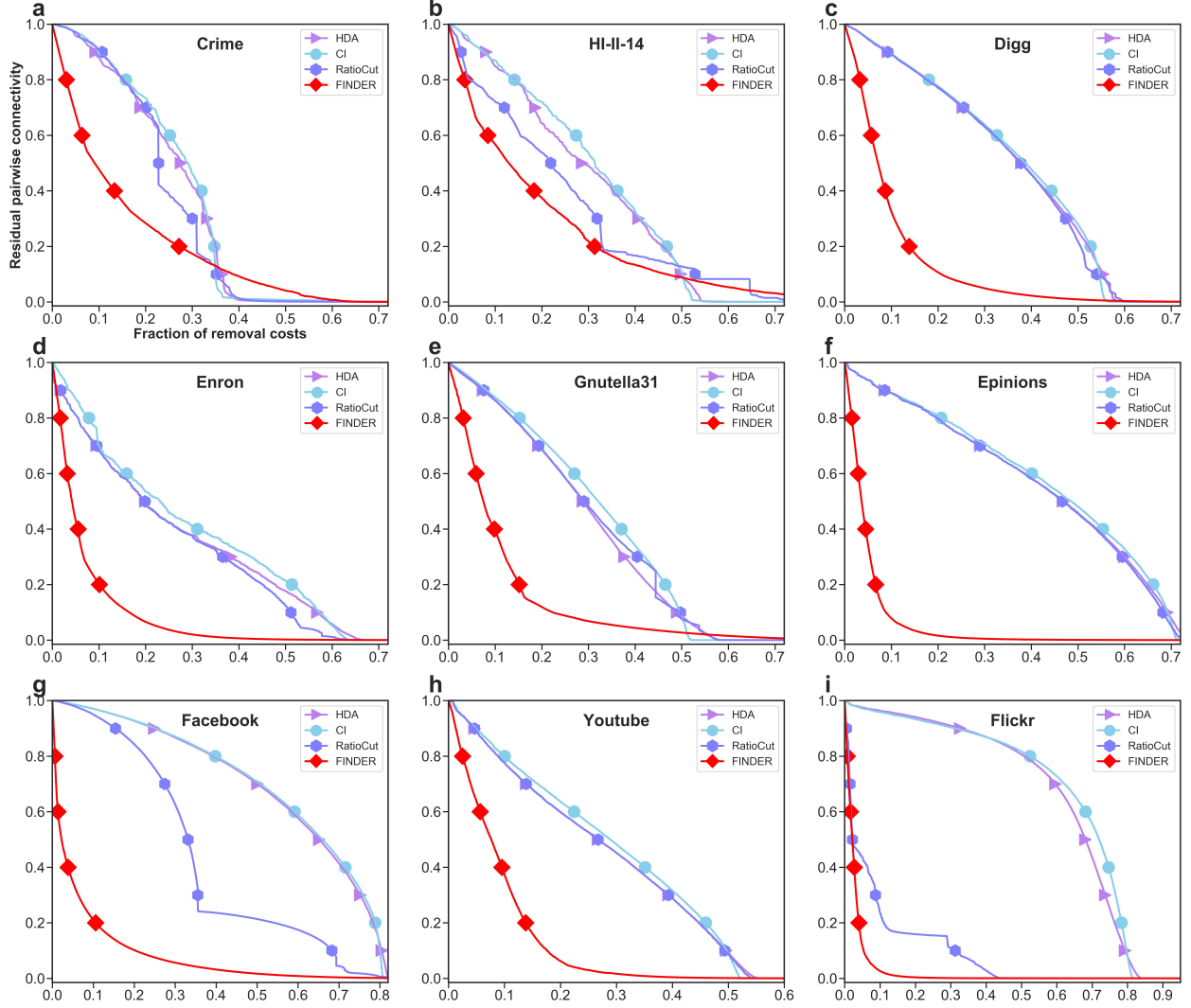


Figure S6: **ANC curves on real-world networks (CN, node-degree-weighted).** The original networks are all node-unweighted. We assign each node a normalized weight, which is proportional to its degree, to measure its importance and the associated removal cost. The horizontal axis is the fraction of the costs for the removed nodes, and the vertical axis is the pairwise connectivity of the residual graph.

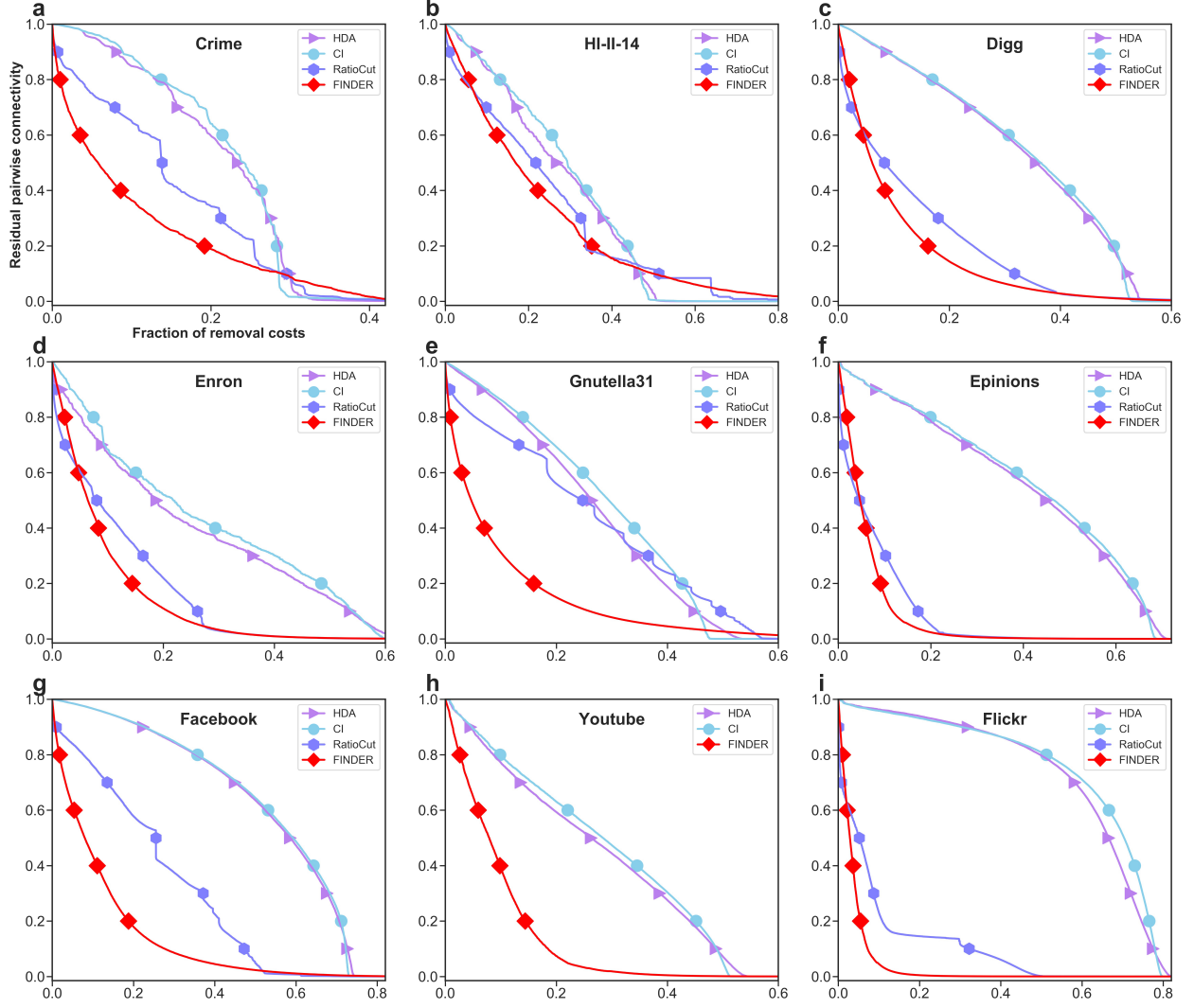


Figure S7: ANC curves on real-world networks (CN, node-random-weighted). The original networks are all node-unweighted. We assign each node a weight according to Eq. S28. The horizontal axis is the fraction of the costs for the removed nodes, and the vertical axis is the pairwise connectivity of the residual graph. Note that RatioCut cannot obtain results for Youtube within acceptable time.

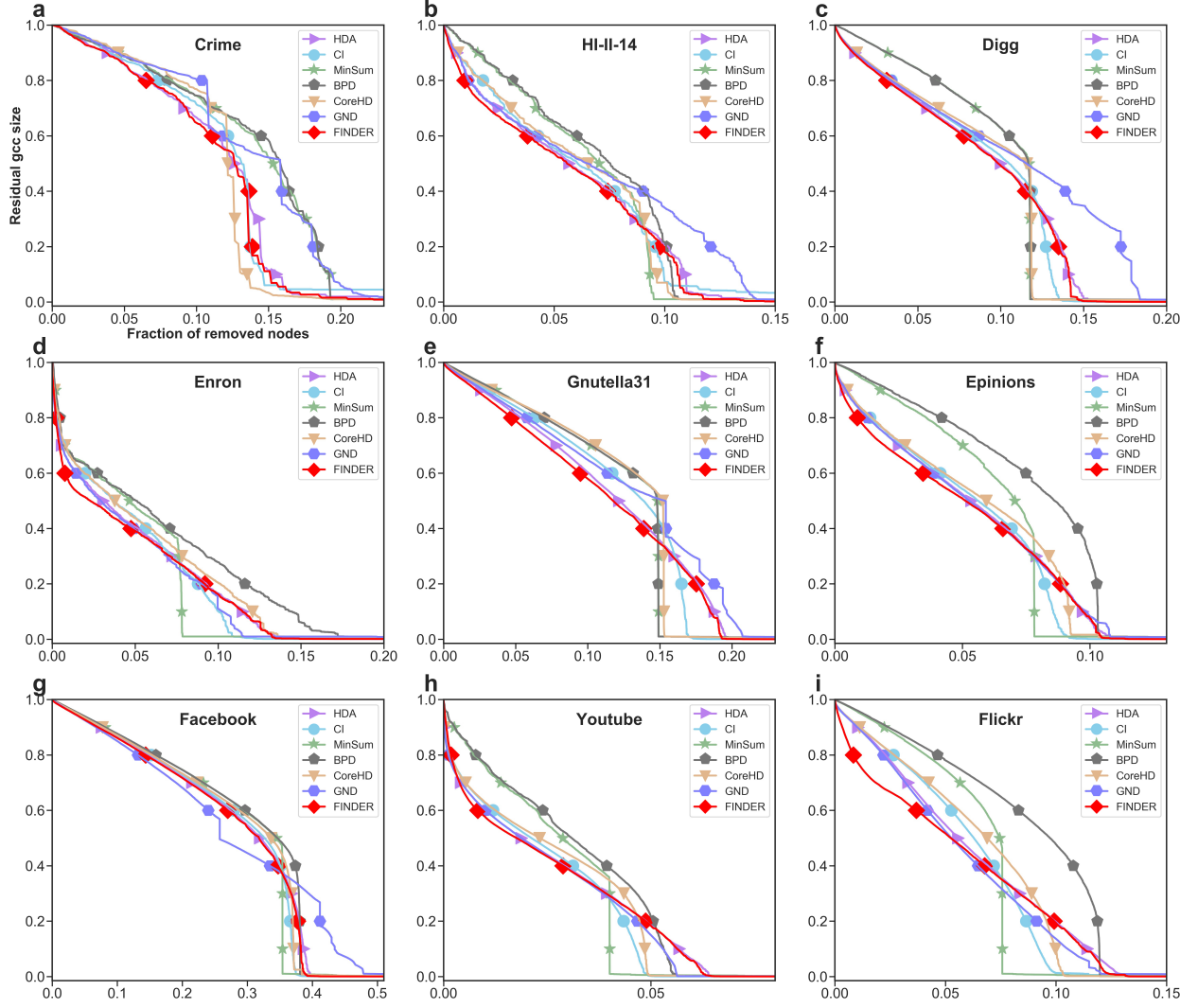


Figure S8: ANC curves on real-world networks (ND, node-unweighted). The horizontal axis is the fraction of the removed nodes, and the vertical axis is the relative size of the giant connected component (GCC) of the residual graph.

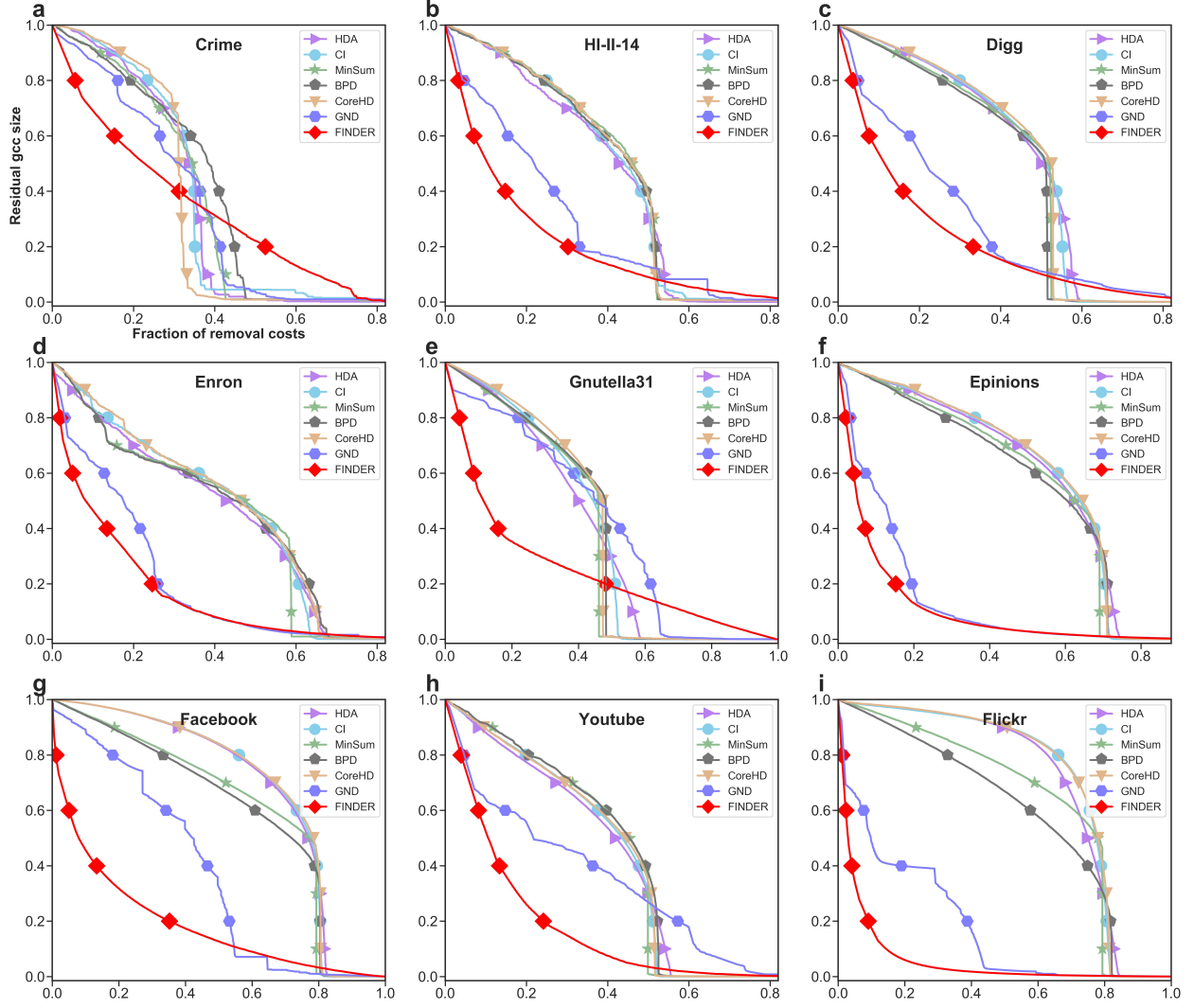


Figure S9: ANC curves on real-world networks (ND, node-degree-weighted). The original networks are all node-unweighted. We assign each node a normalized weight, which is proportional to its degree, to measure its importance and the associated removal cost. The horizontal axis is the fraction of the costs for the removed nodes, and the vertical axis is the relative size of the giant connected component (GCC) of the residual graph.

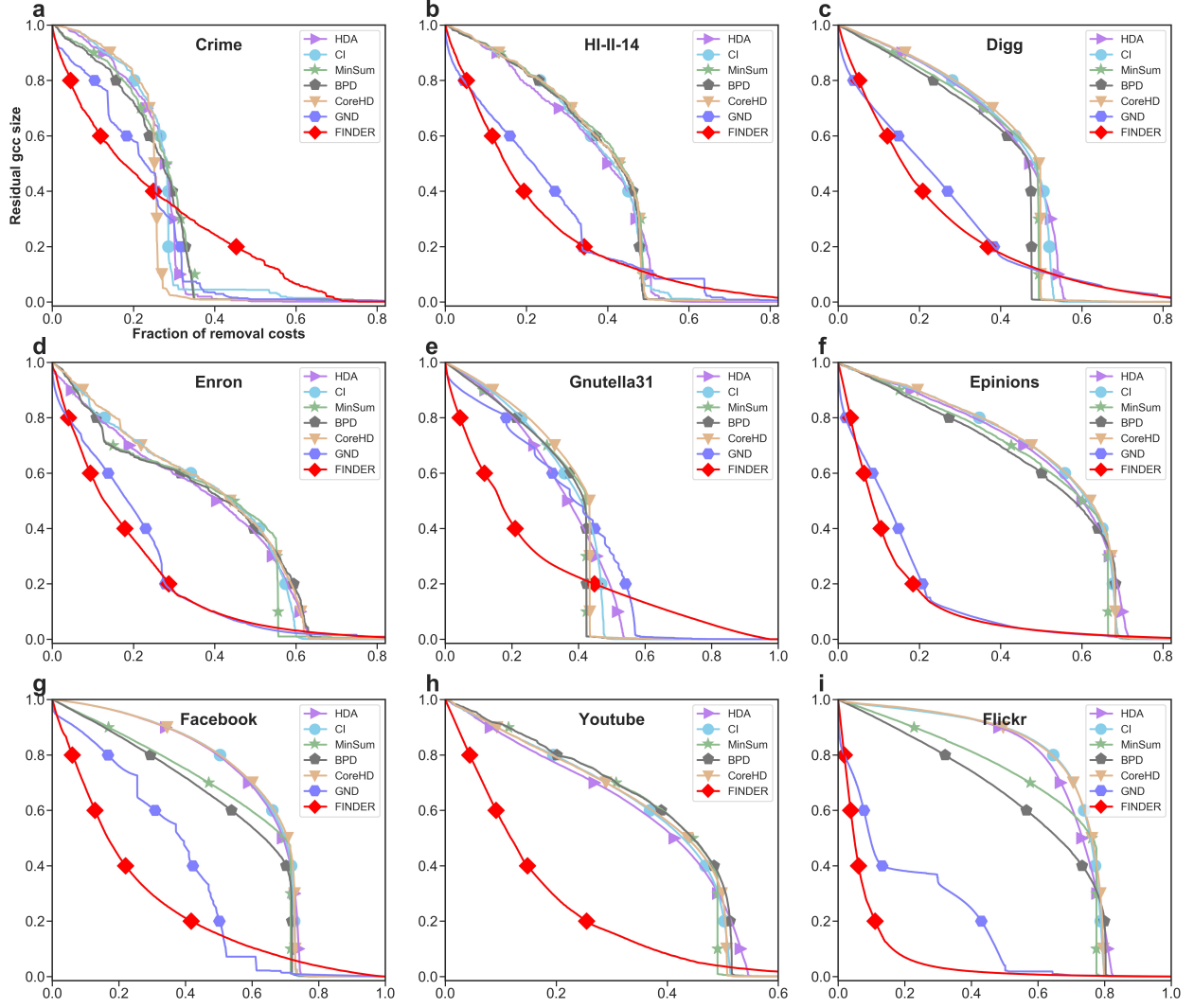


Figure S10: ANC curves on real-world networks (ND, node-random-weighted). The real networks all all node-unweighted. We assign each node a weight according to Eq. S28. The horizontal axis is the fraction of the removed nodes, and the vertical axis is the relative size of the giant connected component (GCC) of the residual graph. Note that GND cannot obtain results for Youtube within acceptable time.

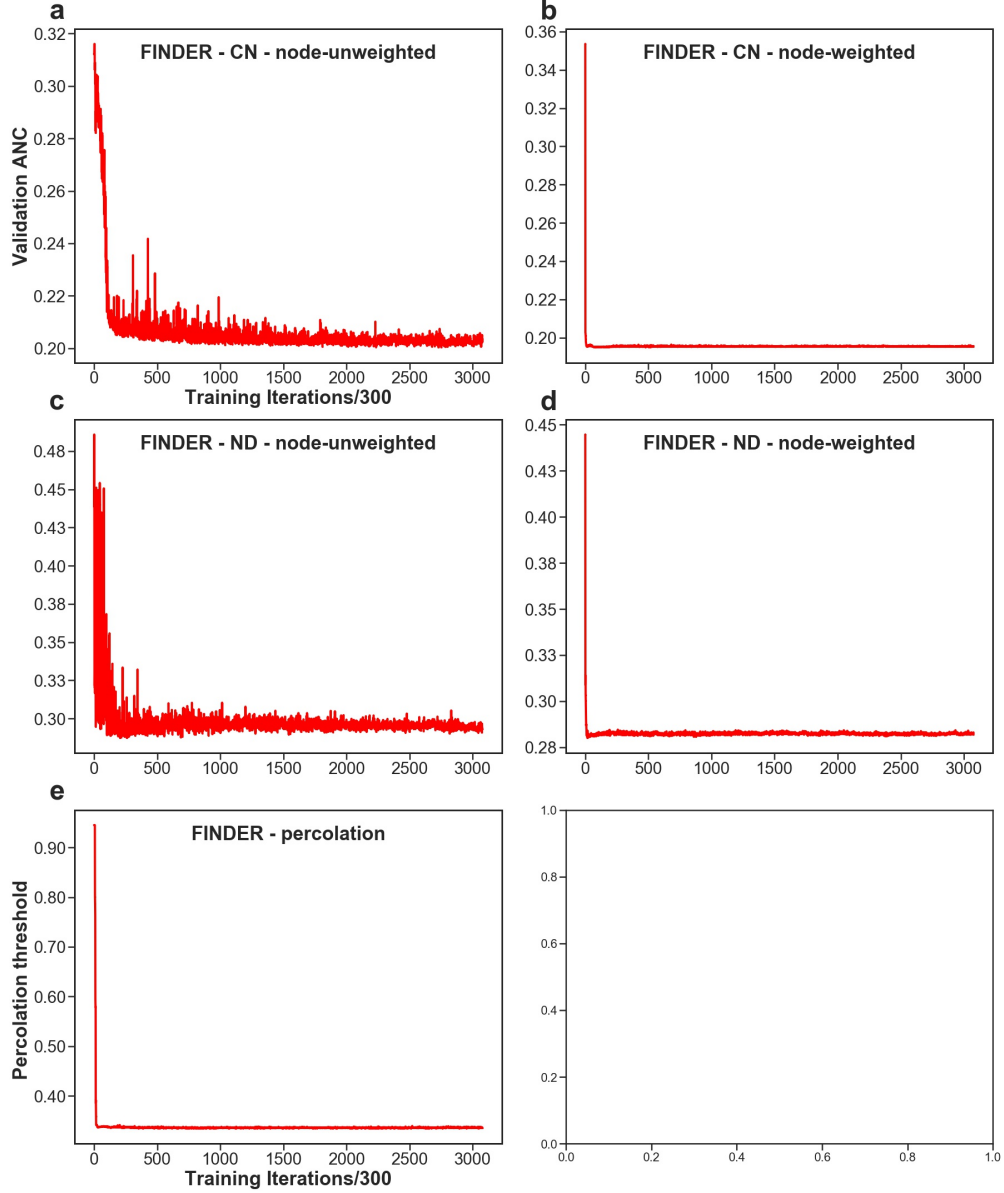


Figure S11: **FINDER training convergences measured by validation performance.** We train five FINDER agents to solve the critical node problem and the network dismantling problem, under node-unweighted and node-weighted scenarios, respectively, together with the minimal percolation problem. We measure these agents' training convergences in terms of the validation performance. All the agents are trained on synthetic BA graphs with node-set size 30-50, and they are all under the same FINDER framework (For FINDER-percolation, it's slightly different from the other four, which would be elaborated more in Section IV.F), only differs by the reward function, which is specified for each problem.

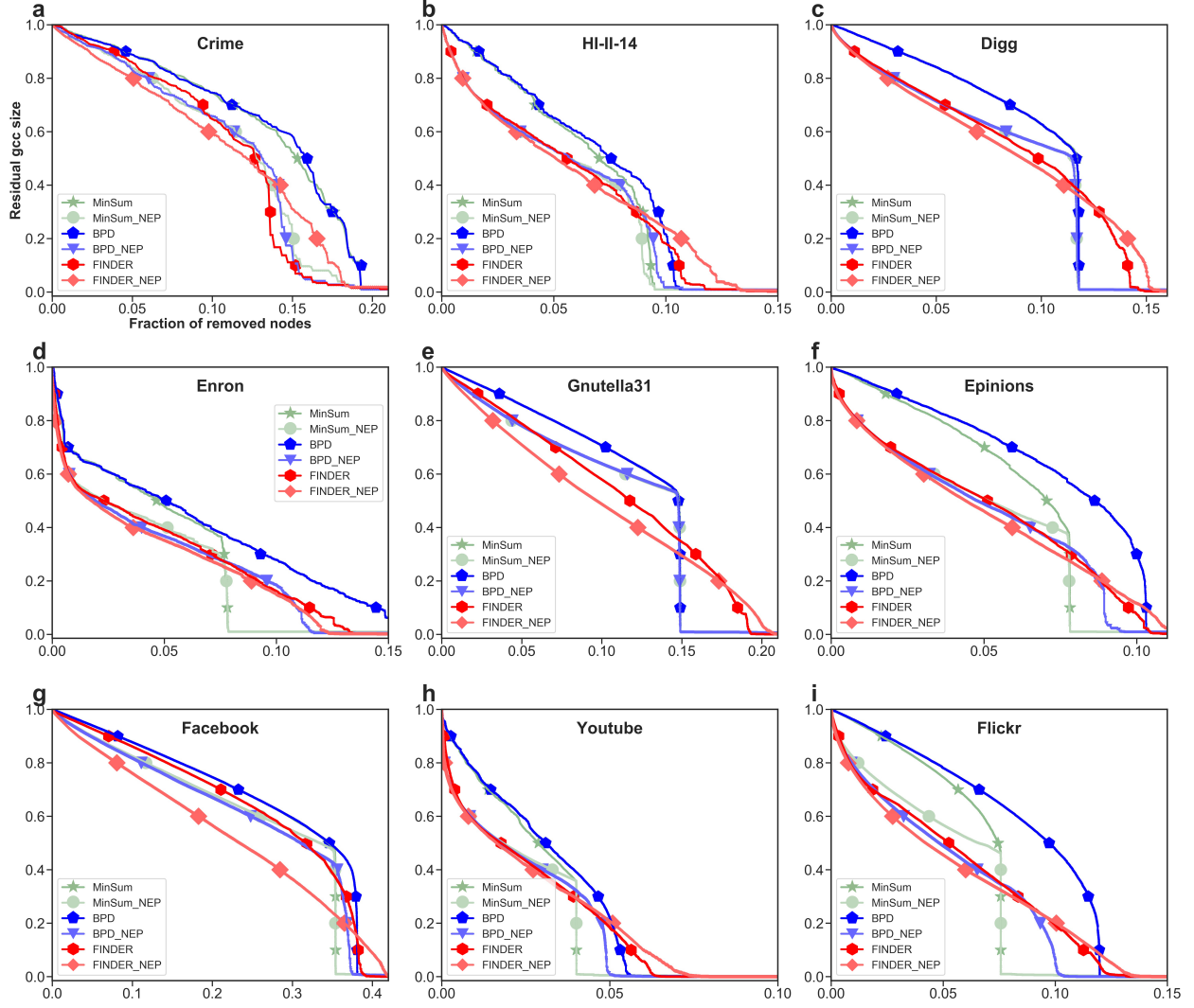


Figure S12: ANC curves for MinSum, BPD, FINDER and their NEP-enhanced ones. The curves of the enhanced methods are overall lower than the vanilla ones, indicating smaller ANC score values.

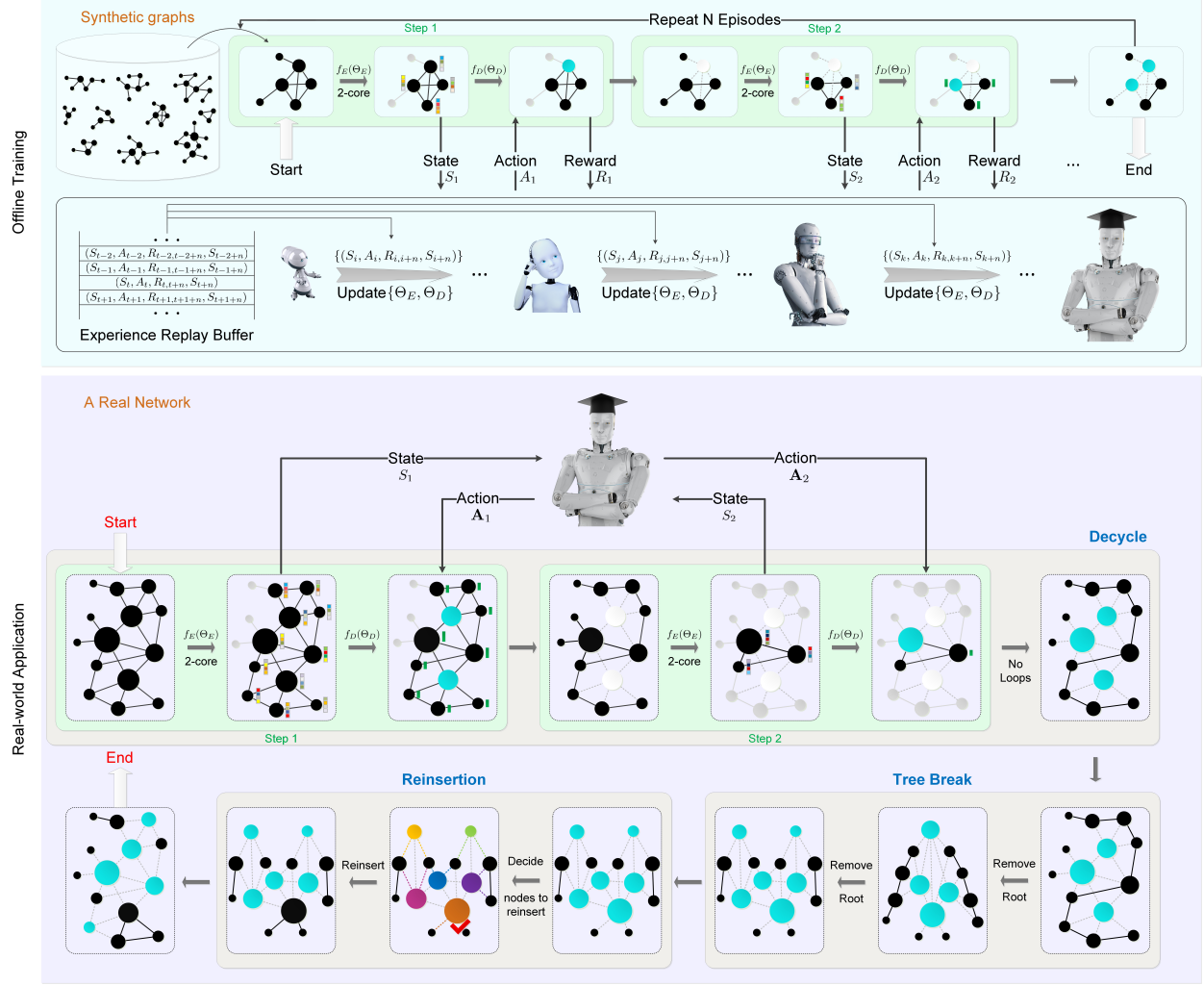


Figure S13: Illustration of FINDER that is designed to the minimal percolation threshold problem. **(top)** Offline Training. During training phase, we sample a graph and let the agent ‘play the game’ on it, i.e., complete a whole key player finding process (denoted as an episode). Previously for each episode, we represent the current observed graph as the model state, and no edges left after nodes’ removal as the terminal sign, and the change of ANC score each step as the reward. Currently, we represent the current graph 2-core as the network state, and the residual graph with no 2-core as the terminal sign, and set -1 as each step’s reward, so as to minimize the nodes’ removal to break all loops in a graph. The training procedure is basically the same as the previous one. **(bottom)** Real-world Application. Once finishes training, the model is supposed to learn how to remove the minimum nodes to break all loops in a graph. When the residual giants are reduced to be tree-like topology, it can be considered as completely fragmented, since any tree can destroyed with a sub-extensive number of node removals. Here we use the same small

real-world network as the example, our goal is to remove the minimal set of nodes to break the network into GCC with no more than 3 nodes. Firstly, we apply the well-trained model to decycle the network, i.e., repeating the iterative compute-and select of the embedding vectors and Q values on the 2-core of the remaining graph until there is no 2-core left. Then we adopt a greedy tree-breaking algorithm described in [9], i.e., iteratively finding and removing the root nodes for the residual tree components, to break them into small pieces. Finally, greedily reinsert some nodes that would close cycles without increasing too much the size of the largest component. As is shown, we finally remove 5 nodes to achieve the goal, the residual GCC contains no more than 3 nodes.

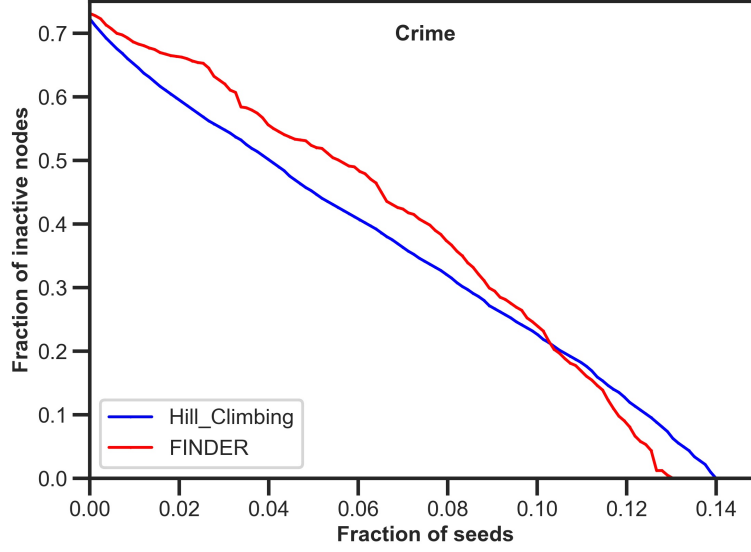


Figure S14: **Comparison of the spreading power of nodes selected by FINDER with that of nodes selected by Hill-Climbing, a simple greedy algorithm with lower bound guarantee for the influence maximization problems.** The curve is plotted with x axis being the fractions of seeds or influencers, y axis being the fraction of nodes that remain inactive after the dynamical process has converged. The spreading dynamic here is a special case of linear threshold model, where each node's threshold is $\theta_i = d_i - 1$, and each edge has a constant weight 1. Note that a node remains inactive only if it has 2 or more inactive neighbors. The solution of FINDER here is obtained by using the model that is designed to minimize the number of node removals, rather than minimizing the ANC score.

Table S1: **Summary of the typical node embedding models.** Here $E(\cdot)$ denotes the encoding function, $D(\cdot)$ denotes the decoding function. $\mathcal{V}$ represents the node set, $\mathcal{E}$ represents the edge set.

Model	$E(v_i)$	$D(v_i, v_j)$	Loss
DeepWalk	Zv_i	$\frac{\exp(z_j^T \cdot z_i)}{\sum_{k \in \mathcal{V}} \exp(z_k^T \cdot z_i)}$	$\sum_{(v_i, v_j) \in \mathcal{E}} -\log(D(z_i, z_j))$
node2vec	Zv_i	$\frac{\exp(z_j^T \cdot z_i)}{\sum_{k \in \mathcal{V}} \exp(z_k^T \cdot z_i)}$	$\sum_{(v_i, v_j) \in \mathcal{E}} -\log(D(v_i, v_j))$
LINE	Zv_i	$\frac{1}{1 + \exp(-z_i^T \cdot z_j)} \text{ or } \frac{\exp(z_j^T \cdot z_i)}{\sum_{k \in \mathcal{V}} \exp(z_k^T \cdot z_i)}$	$\sum_{(v_i, v_j) \in \mathcal{E}} -w_{ij} \cdot \log(D(v_i, v_j))$

Table S2: **Ablation study between FINDER and S2V-DQN.** FINDER differs from S2V-DQN in many aspects. We perform ablation studies to quantify the contributions of each different component. We replace one component each time and keep the remaining ones the same. **(A)** Ablation results on the critical node (CN) problem; **(B)** Ablation results on the network dismantling (ND) problem. All ANC results have been multiplied by 100. The bold blue result indicates the best one (corresponding the lowest ANC score) for that network. S2V-DQN cannot obtain results on Youtube and Flickr for the CN problem within an acceptable time.

(A) Ablation results on the critical node problem									
ANC ($\times 0.01$)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
FINDER	8.19	3.34	5.90	2.03	7.69	3.05	19.37	1.21	3.60
No-GraphSAGE	8.87	11.69	12.52	33.04	9.53	33.18	33.30	9.45	20.97
No-VirtualNode	9.66	4.71	7.66	3.00	8.03	4.89	18.02	2.76	4.87
No-CrossProduct	7.98	3.57	6.23	7.13	6.47	5.72	17.17	4.92	3.94
No-ReconsLoss	8.06	9.58	7.44	21.12	7.21	19.33	28.11	1.92	3.76
S2V-DQN	8.48	30.88	31.37	33.28	24.33	32.75	32.90	NA	NA

(B) Ablation results on the network dismantling problem									
ANC ($\times 0.01$)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
FINDER	10.99	5.54	8.66	4.30	11.10	4.99	26.84	2.37	5.46
No-GraphSAGE	11.39	5.93	9.23	34.15	11.36	6.31	43.33	3.04	5.88
No-VirtualNode	12.18	9.91	13.31	12.33	11.64	9.90	26.71	7.45	9.05
No-CrossProduct	11.00	6.49	9.56	8.85	10.50	8.17	31.55	3.95	7.77
No-ReconsLoss	10.97	5.80	8.70	48.41	10.55	5.29	47.89	2.53	6.08
S2V-DQN	12.70	6.43	11.27	13.64	11.69	24.74	29.95	18.92	38.34

Table S3: **Basic statistics of real-world networks analyzed in this work.**

Statistics	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
Nodes num (N)	829	4,165	29,652	33,696	62,561	75,877	63,392	1,134,890	1,624,991
Edges num (E)	1,473	13,087	84,781	180,811	147,878	405,739	816,831	2,987,624	15,473,043
maximum degree	25	286	310	1,383	95	3,044	1,098	28,754	27,224
average degree	3.55	6.28	5.72	10.73	4.73	10.69	25.77	5.27	19.04
diameter	10	11	5.72	11	11	15	15	24	24
mean shortest path length	5.04	4.16	4.68	4.03	5.96	4.40	4.31	5.55	5.19
clustering coefficient	0.0058	0.0444	0.0054	0.5092	0.0055	0.1378	0.2218	0.0808	0.1892
assortativity	-0.1645	-0.2016	0.0027	-0.1165	-0.0927	-0.0406	0.1768	-0.0369	-0.0167

Table S4: **List of hyper-parameters and their values.** Note: All parameters are obtained by an informal search. Due to the high computational cost, we didn't perform a systematic grid search for the optimal set of hyper-parameters. We believe that our experimental results will be further improved by systematically tuning the hyper-parameter values.

Hyperparameter	Value	Description
replay memory size	5×10^5	Capacity of the experience replay buffer
learning rate	1×10^{-4}	The learning rate used by Adam optimizer
embedding dimension	32,64,128	dimensions of node embedding vector
update time	10^3	the frequency with which target network is updated
layer iterations	3,4	number of neighbor-aggregation iterations
maximum episodes	10^6	maximum episodes for the training process
discount factor	0.99	discount factor gamma used in Q-learning update
Q-learning steps	3,4,5	number of steps for multi-step Q-learning algorithm
mini-batch size	16,32,64	number of mini-batch training samples
initial exploration	1	initial value of ϵ in ϵ -greedy exploration
final exploration	5×10^{-2}	final value of ϵ in ϵ -greedy exploration
exploration steps	10^4	the number of steps over which the initial value of ϵ is linearly annealed to its final value
no_op_max	10^3	number of episodes to be played by the agent without updating parameters, so as to collect initial training samples

Table S5: **Time complexity analysis.**

Method	Time complexity	Description
HDA	$O(n * N \log N)$	N is the number of nodes, n is the total number of greedy steps;
CI	$O(N^2 \log N)$	N is the number of nodes;
MinSum	$O(ET + N \log N + N)$	E is the number of edges, T is the number of message passing steps, N is the number of nodes;
BPD	$O(ET + N \log N + N)$	E is the number of edges, T is the number of message passing steps, N is the number of nodes;
CoreHD	$O(n * N \log N + N)$	N is number of nodes, n is the total number of greedy steps;
GND	$O(t * E)$	E is the number of edges, t is the number of power iteration steps;
FINDER	$O(E + N \log N + N)$	E is the number of edges, N is the number of nodes;

Table S6: **Effects of batch nodes selection used in FINDER to solve the CN problem.** We remove different fractions (StepRatio) of nodes at each adaptive step, and report the ANC (**A**) and running time (**B**) for each corresponding strategy. The bold blue result indicates the best one (corresponding the lowest ANC score) for that network.

(A) ANC ($\times 0.01$)									
ANC ($\times 0.01$)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
StepRatio_0.1%	8.17	3.38	5.87	2.02	7.67	3.03	19.25	1.21	3.54
StepRatio_0.5%	8.17	3.36	5.88	2.02	7.68	3.03	19.29	1.20	3.55
StepRatio_1%	8.19	3.34	5.90	2.03	7.69	3.05	19.37	1.21	3.60
StepRatio_10%	9.21	3.90	6.27	2.21	8.29	3.16	21.25	1.65	4.35
All-At-Once	10.63	4.55	7.14	2.27	8.35	3.20	23.55	1.76	4.49

(B) Running time (s)									
Time (s)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
StepRatio_0.1%	5.2	11.0	164.9	240.6	257.6	359.4	1,183.5	3,260.9	10,388.9
StepRatio_0.5%	1.5	2.5	33.8	52.5	59.3	84.8	287.4	790.6	2,575.4
StepRatio_1%	0.5	2.4	9.2	13.5	14.6	22.7	77.1	262.8	914.5
StepRatio_10%	0.2	0.2	2.7	4.5	5.1	8.0	18.0	93.2	281.3
All-At-Once	0.2	0.1	1.4	2.3	3.0	4.0	7.9	56.4	213.5

Table S7: **Effects of batch nodes selection used in FINDER to solve the ND problem.** We remove different fractions (StepRatio) of nodes at each adaptive step, and report the ANC (**A**) and running time (**B**) for each corresponding strategy. The bold blue result indicates the best one (corresponding the lowest ANC score) for that network.

(A) ANC ($\times 0.01$)									
ANC ($\times 0.01$)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
StepRatio_0.1%	10.92	5.54	8.62	4.30	11.07	4.98	26.78	2.36	5.45
StepRatio_0.5%	10.98	5.55	8.64	4.30	11.08	4.99	26.81	2.36	5.45
StepRatio_1%	10.99	5.54	8.66	4.30	11.10	4.99	26.84	2.37	5.46
StepRatio_10%	11.23	6.19	8.88	4.22	11.50	5.26	27.45	2.75	5.91
All-At-Once	12.14	6.46	9.48	4.19	11.85	5.32	30.95	2.75	5.95

(B) Running time (s)									
Time (s)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
StepRatio_0.1%	5.7	11.2	179.4	244.4	257.8	436.3	1,368.4	3,993.2	13,147.9
StepRatio_0.5%	1.7	2.2	39.7	56.5	58.7	98.6	290.2	907.5	2,831.6
StepRatio_1%	0.3	1.2	8.2	12.7	13.3	22.6	73.4	281.2	989.7
StepRatio_10%	0.2	0.2	2.7	4.4	4.4	7.3	16.5	73.0	281.9
All-At-Once	0.1	0.1	1.3	1.6	2.1	4.6	5.9	45.5	168.2

Table S8: **Effects of batch nodes selection for HDA and FINDER.** Note that it is very time-consuming for the one-by-one removal FINDER on Youtube and Flickr, here we just show its 0.1% batch-by-batch removal FINDER instead. (A) ANC scores; (B) Wall-clock running time (s).

(A) ANC ($\times 0.01$)									
ANC ($\times 0.01$)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
HDA	11.62	5.76	8.79	4.52	11.47	5.20	27.20	2.41	5.98
HDA_1%	34.55	5.80	40.00	4.56	29.23	37.54	45.43	2.45	6.03
FINDER_0.1%	10.92	5.54	8.62	4.30	11.07	4.98	26.78	2.36	5.45
FINDER_1%	10.99	5.54	8.66	4.30	11.10	4.99	26.84	2.37	5.46

(B) Running time (s)									
Time (s)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
HDA	0.0	0.1	1.5	2.5	4.5	7.9	12.8	1963.8	5098.1
HDA_1%	0.0	0.1	1.4	1.4	2.2	3.6	5.2	56.1	126.7
FINDER_0.1%	5.7	11.2	179.4	244.4	257.8	436.3	1368.4	3993.2	13147.9
FINDER_1%	0.3	1.2	8.2	12.7	13.3	22.6	73.4	281.2	989.7

Table S9: **NEP-enhanced results on nine real-world networks (node-unweighted ND problem)**. All results have multiplied 100 for illustration purpose. The bold blue result indicates the best one (corresponding the lowest ANC score) of that network.

ANC ($\times 0.01$)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
MinSum	13.83	6.52	9.52	4.77	11.73	6.04	27.25	2.57	6.10
MinSum_NEP	12.14	5.79	8.56	4.45	11.26	4.97	26.03	2.18	5.34
BPD	13.95	6.96	9.52	6.18	11.72	7.48	28.35	2.99	8.41
BPD_NEP	11.78	5.87	8.55	4.64	11.25	5.25	26.50	2.38	5.89
GND	13.81	6.94	10.66	4.56	12.57	5.37	27.42	2.32	5.83
GND_NEP	12.38	6.22	10.00	4.75	12.36	5.38	27.99	2.41	6.10
FINDER	10.99	5.54	8.66	4.30	11.10	4.99	26.84	2.37	5.46
FINDER_NEP	9.95	5.04	8.09	3.54	10.44	4.45	23.99	2.00	4.66

Table S10: **Results on synthetic networks (CN, node-unweighted).** We test each method on synthetic networks of different scales. For each scale, the result is averaged over 100 randomly generated graphs from the BA model. We show the mean and variance values for the ANC (**A**) and running time (**B**). For FINDER, the results are achieved by removing nodes one by one at each adaptive step. The bold blue result indicates the best one (corresponding the lowest ANC score and execution time) for that scale.

(A) ANC ($\times 0.01$)						
ANC ($\times 0.01$)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	21.65 $\pm$ 1.69	21.59 $\pm$ 1.51	20.74 $\pm$ 1.16	20.08 $\pm$ 0.81	19.80 $\pm$ 0.80	19.61 $\pm$ 0.71
HBA	21.22 $\pm$ 1.75	21.22 $\pm$ 1.64	20.60 $\pm$ 1.31	20.26 $\pm$ 1.13	19.91 $\pm$ 0.99	19.91 $\pm$ 0.96
HCA	23.37 $\pm$ 2.15	23.63 $\pm$ 1.79	23.47 $\pm$ 1.47	23.29 $\pm$ 1.06	23.09 $\pm$ 0.92	23.12 $\pm$ 0.92
HPRA	21.98 $\pm$ 1.87	21.82 $\pm$ 1.46	20.92 $\pm$ 1.11	20.26 $\pm$ 0.88	19.97 $\pm$ 0.77	19.81 $\pm$ 0.69
CI	31.53 $\pm$ 0.37	32.09 $\pm$ 0.25	32.46 $\pm$ 0.26	32.71 $\pm$ 0.20	32.86 $\pm$ 0.14	32.92 $\pm$ 0.11
RatioCut	21.43 $\pm$ 1.63	21.43 $\pm$ 1.42	20.84 $\pm$ 0.99	20.47 $\pm$ 1.05	20.18 $\pm$ 0.96	20.02 $\pm$ 0.88
FINDER	19.34$\pm$1.37	19.12$\pm$1.08	18.31$\pm$0.82	17.85$\pm$0.62	17.59$\pm$0.54	17.48$\pm$0.52

(B) Running time (s)						
Time (s)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	0.00±0.00	0.00±0.00	0.01±0.00	0.03±0.01	0.06±0.01	0.10±0.01
HBA	0.06±0.02	0.35±0.20	2.64±1.45	11.20±3.90	33.76±9.54	69.59±14.53
HCA	0.07±0.03	0.38±0.22	2.78±1.53	11.96±4.18	31.86±8.31	67.35±13.53
HPRA	0.18±0.06	0.56±0.21	2.03±0.74	5.26±1.27	10.40±1.97	17.19±2.81
CI	0.01±0.00	0.03±0.01	0.19±0.11	0.79±0.28	2.00±0.53	4.37±0.99
RatioCut	0.19±0.13	0.42±0.15	1.09±0.35	2.21±0.59	3.48±0.69	5.89±25.05
FINDER	0.07±0.03	0.13±0.03	0.29±0.07	0.56±0.10	0.89±0.12	1.30±0.13

Table S11: **Results on synthetic networks (CN, node-degree-weighted).** We assign each node a normalized weight that is proportional to its degree. We test each method on synthetic networks of different scales. For each scale, the result is averaged over 100 randomly generated graphs from the BA model. We show the mean and variance values for the ANC (A) and running time (B). For FINDER, the results are achieved by removing nodes one by one at each adaptive step. The bold blue result indicates the best one (corresponding the lowest ANC score and execution time) for that scale.

(A) Weighted ANC ($\times 0.01$)						
ANC ($\times 0.01$)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	37.11 $\pm$ 1.47	39.56 $\pm$ 1.16	41.29 $\pm$ 0.85	42.04 $\pm$ 0.54	42.35 $\pm$ 0.43	42.53 $\pm$ 0.37
HBA	35.54 $\pm$ 1.52	37.82 $\pm$ 1.19	39.59 $\pm$ 0.86	40.57 $\pm$ 0.68	40.86 $\pm$ 0.47	41.15 $\pm$ 0.36
HCA	36.87 $\pm$ 1.32	39.10 $\pm$ 1.10	40.68 $\pm$ 0.68	41.60 $\pm$ 0.48	41.88 $\pm$ 0.42	42.09 $\pm$ 0.35
HPRA	36.55 $\pm$ 1.52	38.98 $\pm$ 1.32	40.71 $\pm$ 0.78	41.48 $\pm$ 0.59	41.84 $\pm$ 0.43	41.99 $\pm$ 0.37
CI	23.13 $\pm$ 1.78	24.81 $\pm$ 1.90	26.05 $\pm$ 1.04	25.27 $\pm$ 0.80	24.99 $\pm$ 0.55	25.04 $\pm$ 0.51
RatioCut	25.06 $\pm$ 1.95	23.79 $\pm$ 1.33	23.25 $\pm$ 0.87	23.35 $\pm$ 0.71	23.53 $\pm$ 0.73	23.66 $\pm$ 0.72
FINDER	19.84$\pm$0.71	19.61$\pm$0.47	19.20$\pm$0.29	19.01$\pm$0.19	18.93$\pm$0.15	18.89$\pm$0.13

(B) Running time (s)						
Time (s)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	0.00±0.00	0.01±0.00	0.02±0.01	0.04±0.01	0.07±0.02	0.11±0.02
HBA	0.09±0.04	0.49±0.27	3.44±1.92	14.77±5.26	39.57±10.46	83.30±16.60
HCA	0.09±0.04	0.49±0.28	3.54±1.95	15.02±5.26	39.91±10.38	83.74±16.92
HPRA	0.23±0.07	0.70±0.27	2.50±0.91	6.41±1.52	12.04±1.97	19.27±2.62
CI	0.01±0.00	0.03±0.01	0.19±0.11	0.79±0.28	2.00±0.53	4.37±0.99
RatioCut	0.35±0.34	0.70±0.29	1.67±0.70	3.66±0.77	5.24±1.03	6.41±1.06
FINDER	0.11±0.03	0.21±0.06	0.53±0.15	1.05±0.19	1.72±0.23	2.55±0.30

Table S12: **Results on synthetic networks (CN, node-random-weighted).** We assign each node a weight according to Eq .S28. We test each method on synthetic networks of different scales. For each scale, the result is averaged over 100 randomly generated graphs from the BA model. We show the mean and variance values for the ANC (**A**) and running time (**B**). For FINDER, the results are achieved by removing nodes one by one at each adaptive step. The bold blue result indicates the best one (corresponding the lowest ANC score and execution time) for that scale.

(A) Weighted ANC ($\times 0.01$)						
ANC ($\times 0.01$)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	33.17 $\pm$ 3.89	35.82 $\pm$ 2.64	37.26 $\pm$ 2.28	38.33 $\pm$ 1.51	38.70 $\pm$ 1.13	39.06 $\pm$ 0.98
HBA	32.09 $\pm$ 4.02	34.81 $\pm$ 2.56	36.01 $\pm$ 2.12	37.21 $\pm$ 1.70	37.64 $\pm$ 1.24	38.11 $\pm$ 0.99
HCA	33.25 $\pm$ 4.06	36.08 $\pm$ 2.57	37.08 $\pm$ 2.11	38.25 $\pm$ 1.47	38.67 $\pm$ 1.27	39.02 $\pm$ 0.99
HPRA	33.02 $\pm$ 3.63	35.58 $\pm$ 2.70	37.25 $\pm$ 1.92	38.17 $\pm$ 1.51	38.62 $\pm$ 1.18	38.94 $\pm$ 1.05
CI	25.43 $\pm$ 2.94	26.32 $\pm$ 2.30	27.54 $\pm$ 1.47	26.64 $\pm$ 1.23	26.14 $\pm$ 0.93	26.23 $\pm$ 0.91
RatioCut	16.39 $\pm$ 2.31	16.23 $\pm$ 1.59	16.32 $\pm$ 1.17	16.51 $\pm$ 0.83	16.49 $\pm$ 0.84	16.38 $\pm$ 0.76
FINDER	13.38$\pm$1.84	13.47$\pm$1.10	13.53$\pm$0.82	13.51$\pm$0.54	13.46$\pm$0.41	13.46$\pm$0.41

(B) Running time (s)						
Time (s)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	1.00±0.00	0.01±0.00	0.02±0.01	0.05±0.02	0.09±0.03	0.14±0.04
HBA	0.11±0.06	0.68±0.37	4.79±2.72	21.47±8.06	47.21±12.79	93.09±19.54
HCA	0.12±0.06	0.59±0.32	4.15±2.23	16.41±5.84	43.86±11.78	101.23±24.29
HPRA	0.29±0.11	0.76±0.31	2.97±1.09	7.31±1.86	13.36±2.45	21.44±3.23
CI	0.01±0.00	0.03±0.01	0.19±0.11	0.79±0.28	2.00±0.53	4.37±0.99
RatioCut	0.27±0.09	0.62±0.22	4.04±25.08	1.62±0.97	4.70±0.91	7.03±1.16
FINDER	0.16±0.06	0.33±0.09	0.77±0.20	1.57±0.27	2.56±0.33	3.78±0.41

Table S13: **Running time comparison on real-world networks (CN problem).** The results of FINDER are achieved by removing 1% node at each adaptive step. HBA, HCA, and HPRA are not compared due to their high computational costs on these networks. The bold blue result indicates the best one (corresponding the lowest execution time) of that network. (A) Node-unweightd. Each node has uniform weights. (B) Each node is assigned a normalized weight that is proportional to its degree. (C) Each node is assigned a weight according to Eq. S28. Note that RatioCut cannot obtain results for Youtube within acceptable time in this case.

(A) Node-unweighted									
Time (s)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
HDA	0.0	0.0	1.5	2.5	4.5	7.9	12.8	1,963.8	5,098.1
CI	0.0	0.5	11.0	70.7	9.4	668.4	2,723.2	7,056.4	>5d
RatioCut	0.1	1.3	55.0	35.7	386.3	233.1	4,614.5	52,246.2	>5d
FINDER	0.5	2.4	9.2	13.5	14.6	22.7	77.1	262.8	914.5
(B) Node-degree-weighted.									
Time (s)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
HDA	0.0	0.0	1.5	2.5	4.5	7.9	12.8	1,963.8	5,098.1
CI	0.0	0.5	11.0	70.7	9.4	668.4	2,723.2	7,056.4	>5d
RatioCut	0.1	1.4	42.9	62.1	144.8	382.1	355.0	61,288.2	>5d
FINDER	1.6	6.5	45.8	66.8	94.1	164.8	274.0	2,051.8	8,579.2
(C) Node-random-weighted.									
Time (s)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
HDA	0.0	0.0	1.5	2.5	4.5	7.9	12.8	1,963.8	5,098.1
CI	0.0	0.5	11.0	70.7	9.4	668.4	2,723.2	7,056.4	>5d
RatioCut	0.2	3.1	52.6	77.7	154.8	441.6	391.8	NA	>5d
FINDER	1.3	7.9	57.7	88.6	125.3	188.6	303.3	2,130.4	10,373.8

Table S14: **Results on synthetic networks (ND, node-unweighted)**. We test each method on synthetic networks of different scales. For each scale, the result is averaged over 100 randomly generated graphs from the BA model. We show the mean and variance values for the ANC (**A**) and running time (**B**). For FINDER, the results are achieved by removing nodes one by one at each adaptive step. The bold blue result indicates the best one (corresponding the lowest ANC score and execution time) for that scale. Note that HBA has long been considered as the most effective strategy for the network dismantling problem (node-unweighted scenario)[5], which can be validated in this Table. However, it is computationally prohibitive in medium and large scale networks, while FINDER is only slightly worse than HBA, and is much faster.

(A) ANC ($\times 0.01$)						
ANC ($\times 0.01$)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	29.10 $\pm$ 2.03	27.74 $\pm$ 1.87	25.86 $\pm$ 1.35	24.74 $\pm$ 0.94	24.23 $\pm$ 0.91	24.01 $\pm$ 0.80
HBA	28.28$\pm$1.94	26.51$\pm$1.76	24.30$\pm$1.32	23.13$\pm$0.96	22.57$\pm$0.80	22.35$\pm$0.75
HCA	29.22 $\pm$ 2.18	27.50 $\pm$ 1.92	25.37 $\pm$ 1.44	24.36 $\pm$ 0.95	23.76 $\pm$ 0.82	23.47 $\pm$ 0.82
HPRA	30.29 $\pm$ 2.31	29.07 $\pm$ 2.03	27.27 $\pm$ 1.49	26.29 $\pm$ 1.10	25.87 $\pm$ 0.96	25.65 $\pm$ 0.91
CI	48.34 $\pm$ 0.64	48.15 $\pm$ 0.75	48.14 $\pm$ 0.73	48.30 $\pm$ 0.60	48.56 $\pm$ 0.48	48.58 $\pm$ 0.38
MinSum	29.63 $\pm$ 1.91	28.00 $\pm$ 1.79	25.55 $\pm$ 1.33	24.19 $\pm$ 0.92	27.17 $\pm$ 1.31	26.71 $\pm$ 1.19
BPD	31.90 $\pm$ 2.15	30.97 $\pm$ 1.91	29.22 $\pm$ 1.47	28.18 $\pm$ 1.08	27.63 $\pm$ 1.02	27.57 $\pm$ 0.91
CoreHD	29.60 $\pm$ 2.06	27.98 $\pm$ 1.87	25.60 $\pm$ 1.39	24.32 $\pm$ 0.96	23.81 $\pm$ 0.78	23.54 $\pm$ 0.76
GND	30.20 $\pm$ 2.26	28.69 $\pm$ 1.91	26.89 $\pm$ 1.45	25.94 $\pm$ 1.54	25.35 $\pm$ 1.41	25.02 $\pm$ 1.26
FINDER	29.09 $\pm$ 2.05	27.47 $\pm$ 1.82	25.32 $\pm$ 1.28	24.17 $\pm$ 0.93	23.70 $\pm$ 0.85	23.45 $\pm$ 0.74

(B) Running time (s)						
Time (s)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	0.0±0.0	0.0±0.0	0.01±0.0	0.03±0.01	0.05±0.01	0.08±0.01
HBA	0.07±0.03	0.39±0.23	2.92±1.62	13.73±5.12	40.49±11.53	78.73±15.77
HCA	0.07±0.03	0.38±0.22	2.95±1.64	13.19±4.75	40.07±10.98	77.17±15.7
HPRA	0.17±0.05	0.55±0.21	2.12±0.78	5.36±1.26	11.94±2.44	20.74±2.91
CI	0.01±0.00	0.03±0.01	0.19±0.11	0.79±0.28	2.00±0.53	4.37±0.99
MinSum	0.70±0.20	1.15±0.43	1.80±0.56	2.82±1.10	3.65±1.43	4.52±1.88
BPD	0.73±0.51	1.48±0.71	4.08±1.68	8.70±1.59	11.21±1.73	15.34±1.72
CoreHD	0.01±0.00	0.01±0.00	0.03±0.01	0.05±0.01	0.08±0.01	0.10±0.01
GND	0.19±0.13	0.42±0.15	1.09±0.35	2.21±0.59	3.48±0.69	5.89±25.05
FINDER	0.10±0.24	0.14±0.04	0.30±0.08	0.56±0.09	0.89±0.12	1.30±0.15

Table S15: **Results on synthetic networks (ND, node-degree-weighted)**. Each node is assigned a normalized weight that is proportional to its degree. We test each method on synthetic networks of different scales. For each scale, the result is averaged over 100 randomly generated graphs from the BA model. We show the mean and variance values for the ANC (**A**) and running time (**B**). For FINDER, the results are achieved by removing nodes one by one at each adaptive step. The bold blue result indicates the best one (corresponding the lowest ANC score and execution time) for that scale.

(A) Weighted ANC ($\times 0.01$)						
ANC ($\times 0.01$)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	46.65 $\pm$ 1.37	47.80 $\pm$ 1.02	48.39 $\pm$ 0.80	48.56 $\pm$ 0.54	48.57 $\pm$ 0.50	48.63 $\pm$ 0.43
HBA	44.80 $\pm$ 1.23	45.45 $\pm$ 0.89	45.85 $\pm$ 0.68	46.09 $\pm$ 0.59	46.09 $\pm$ 0.43	46.18 $\pm$ 0.38
HCA	45.81 $\pm$ 1.20	46.56 $\pm$ 0.99	46.78 $\pm$ 0.63	47.05 $\pm$ 0.50	47.03 $\pm$ 0.47	47.08 $\pm$ 0.41
HPRA	46.49 $\pm$ 1.34	47.67 $\pm$ 1.16	48.44 $\pm$ 0.72	48.68 $\pm$ 0.59	48.82 $\pm$ 0.50	48.85 $\pm$ 0.45
CI	38.75 $\pm$ 1.93	40.90 $\pm$ 2.19	42.28 $\pm$ 1.12	41.27 $\pm$ 0.94	40.81 $\pm$ 0.76	40.62 $\pm$ 0.66
MinSum	45.91 $\pm$ 1.14	46.66 $\pm$ 0.74	46.93 $\pm$ 0.61	46.87 $\pm$ 0.45	49.97 $\pm$ 0.95	49.89 $\pm$ 0.93
BPD	48.95 $\pm$ 1.41	50.94 $\pm$ 1.20	52.12 $\pm$ 0.74	52.39 $\pm$ 0.70	52.55 $\pm$ 0.62	52.68 $\pm$ 0.50
CoreHD	46.42 $\pm$ 1.14	47.10 $\pm$ 0.82	47.32 $\pm$ 0.62	47.29 $\pm$ 0.47	47.39 $\pm$ 0.33	47.36 $\pm$ 0.34
GND	41.52 $\pm$ 3.04	40.36 $\pm$ 1.78	39.33 $\pm$ 1.67	38.94 $\pm$ 1.22	38.78 $\pm$ 0.92	38.85 $\pm$ 0.86
FINDER	35.01$\pm$0.94	34.04$\pm$0.73	33.04$\pm$0.51	32.59$\pm$0.34	32.47$\pm$0.28	32.46$\pm$0.26

(B) Running time (s)						
Time (s)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	0.00±0.00	0.01±0.00	0.02±0.01	0.04±0.03	0.08±0.06	0.12±0.04
HBA	0.08±0.03	0.53±0.37	3.85±2.15	15.37±5.58	37.74±9.76	76.22±15.97
HCA	0.09±0.04	0.52±0.29	3.77±2.09	14.87±5.23	36.65±9.49	73.69±15.25
HPRA	0.17±0.05	0.53±0.2	1.9±0.69	4.94±1.14	9.18±1.48	14.64±1.91
CI	0.01±0.00	0.03±0.01	0.19±0.11	0.79±0.28	2.00±0.53	4.37±0.99
MinSum	0.70±0.20	1.15±0.43	1.80±0.56	2.82±1.10	3.65±1.43	4.52±1.88
BPD	0.34±0.47	1.00±0.69	2.97±1.31	5.52±0.95	6.23±2.87	7.76±2.85
CoreHD	0.01±0.00	0.01±0.00	0.03±0.01	0.05±0.01	0.08±0.01	0.10±0.01
GND	0.35±0.34	0.70±0.29	1.67±0.70	3.66±0.77	5.24±1.03	6.41±1.06
FINDER	0.12±0.13	0.23±0.06	0.53±0.14	1.08±0.20	1.82±0.30	2.54±0.27

Table S16: **Results on synthetic networks (ND, node-random-weighted).** Each node’s removal cost is according to Eq. S28. We test each method on synthetic networks of different scales. For each scale, the result is averaged over 100 randomly generated graphs from the BA model. We show the mean and variance values for the ANC (**A**) and running time (**B**). For FINDER, the results are achieved by removing nodes one by one at each adaptive step. The bold blue result indicates the best one (corresponding the lowest ANC score and execution time) for that scale. Note that GND cannot obtain results for Youtube within acceptable time.

(A) Weighted ANC ($\times 0.01$)						
ANC ($\times 0.01$)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	42.60 $\pm$ 3.93	43.88 $\pm$ 2.65	44.24 $\pm$ 2.43	44.73 $\pm$ 1.53	44.76 $\pm$ 1.32	45.11 $\pm$ 1.02
HBA	41.03 $\pm$ 4.23	42.17 $\pm$ 2.63	41.95 $\pm$ 2.15	42.44 $\pm$ 1.80	42.59 $\pm$ 1.34	42.90 $\pm$ 1.07
HCA	41.03 $\pm$ 4.23	42.17 $\pm$ 2.63	41.95 $\pm$ 2.15	42.44 $\pm$ 1.80	42.59 $\pm$ 1.34	42.90 $\pm$ 1.07
HPRA	42.59 $\pm$ 3.80	3.91 $\pm$ 2.85	44.71 $\pm$ 1.97	45.06 $\pm$ 1.59	45.29 $\pm$ 1.33	45.54 $\pm$ 1.19
CI	41.14 $\pm$ 2.97	42.25 $\pm$ 2.53	43.62 $\pm$ 1.45	42.56 $\pm$ 1.25	41.86 $\pm$ 1.07	41.79 $\pm$ 1.01
MinSum	42.01 $\pm$ 4.12	43.30 $\pm$ 2.62	43.10 $\pm$ 2.06	43.20 $\pm$ 1.55	46.45 $\pm$ 1.55	46.49 $\pm$ 1.35
BPD	45.29 $\pm$ 4.02	47.40 $\pm$ 2.89	48.29 $\pm$ 2.18	48.74 $\pm$ 1.46	48.86 $\pm$ 1.31	49.25 $\pm$ 1.05
CoreHD	42.47 $\pm$ 4.02	43.52 $\pm$ 2.72	43.38 $\pm$ 2.35	43.57 $\pm$ 1.68	43.73 $\pm$ 1.22	43.96 $\pm$ 1.13
GND	29.76 $\pm$ 2.98	29.99 $\pm$ 2.21	30.11 $\pm$ 1.80	30.16 $\pm$ 1.00	30.12 $\pm$ 1.00	29.86 $\pm$ 0.92
FINDER	28.24$\pm$2.57	28.08$\pm$1.47	27.96$\pm$1.07	28.01$\pm$0.76	27.92$\pm$0.67	27.94$\pm$0.68

(B) Running time (s)						
Time (s)	30-50	50-100	100-200	200-300	300-400	400-500
HDA	0.00±0.00	0.01±0.00	0.02±0.01	0.05±0.02	0.09±0.03	0.14±0.03
HBA	0.12±0.05	0.60±0.38	4.18±2.28	16.52±6.38	42.87±12.28	89.11±18.68
HCA	0.14±0.06	0.73±0.42	5.10±3.02	22.00±7.64	48.48±13.36	93.11±18.93
HPRA	0.28±0.10	0.83±0.35	2.86±1.11	7.52±1.91	13.69±2.65	22.11±3.34
CI	0.01±0.00	0.03±0.01	0.19±0.11	0.79±0.28	2.00±0.53	4.37±0.99
MinSum	0.70±0.20	1.15±0.43	1.80±0.56	2.82±1.10	3.65±1.43	4.52±1.88
BPD	0.38±0.49	1.04±0.58	3.02±1.12	5.66±1.03	37.01±1.54	7.80±2.98
CoreHD	0.01±0.00	0.01±0.00	0.03±0.01	0.05±0.01	0.08±0.01	0.10±0.01
GND	0.27±0.09	0.62±0.22	4.04±0.25	1.62±0.97	4.70±0.91	7.03±1.16
FINDER	0.10±0.03	0.20±0.05	0.48±0.13	0.95±0.16	1.55±0.20	2.28±0.25

Table S17: **Running time comparison on real-world networks (ND problem).** The results of FINDER are achieved by removing 1% node at each adaptive step. HBA, HCA, and HPRA are not compared due to their high computational costs on these networks. The bold blue result indicates the best one (corresponding the lowest execution time) of that network. **(A)** Node-unweightd, each node has uniform weight; **(B)** Node-degree-weighted, each node is assigned a normalized weight that is proportional to its degree; **(C)** Node-random-weighted, each node’s removal cost is according to Eq. S28. Note that GND cannot obtain results for Youtube within acceptable time in this case.

(A) Node-unweighted									
Time (s)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
HDA	0.0	0.0	0.3	2.5	4.5	7.9	12.8	1,963.8	5,098.2
CI	0.0	0.5	4.9	70.7	9.4	668.4	2,723.2	7,056.4	>5d
MinSum	2.0	20.0	108.2	323.6	225.4	681.1	1,408.0	5,478.2	35,215.2
BPD	0.3	9.0	39.0	86.0	41.0	254.0	839.0	462.0	17,942.0
CoreHD	0.1	2.0	8.5	36.2	11.9	142.9	214.8	2,018.9	36,837.3
GND	0.1	1.3	12.7	35.7	386.3	233.1	4,614.5	52,246.2	>5d
FINDER	0.3	1.2	7.1	12.7	13.3	22.6	73.4	281.22	989.66

(B) Node-degree-weighted

Time (s)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
HDA	0.0	0.0	0.3	2.5	4.5	7.9	12.8	1,963.8	5,098.1
CI	0.0	0.5	4.9	70.7	9.4	668.4	2,723.2	7,056.4	>5d
MinSum	2.0	20.0	108.2	323.5	225.4	681.1	1,408.0	5,478.2	35,215.2
BPD	1.0	9.0	24.0	54.0	41.0	166.0	553.0	507.0	20,110.0
CoreHD	0.1	2.0	8.5	36.2	11.9	142.9	214.8	2,018.9	36,837.3
GND	0.1	1.4	9.0	62.1	144.8	382.1	355.0	61,288.2	174,363.0
FINDER	0.6	7.3	42.9	40.4	50.8	143.4	555.8	2,044.9	7,734.6

(C) Node-random-weighted

Time (s)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
HDA	0.0	0.0	0.3	2.5	4.5	7.9	12.8	1,963.8	5,098.1
CI	0.0	0.5	4.9	70.7	9.4	668.4	2,723.2	7,056.4	>5d
MinSum	2.0	20.0	108.2	323.5	225.4	681.1	1,408.0	5,478.2	35,215.2
BPD	1.0	7.0	23.0	64.0	66.0	187.0	631.0	572.0	21,800.0
CoreHD	0.1	2.0	8.5	36.2	11.9	142.9	214.8	2,018.9	36,837.3
GND	0.2	3.1	10.1	77.7	154.8	441.6	391.8	NA	202,476.0
FINDER	1.7	6.9	43.2	93.4	134.7	220.7	345.1	2,348.9	10,618.4

Table S18: **FINDER’s performance for the minimal percolation threshold problem.** (A) Percolation threshold, the fraction of nodes of which removal break the network into residual component with 1% GCC size; (B) Running time (s).

(A) Percolation threshold ($\times 0.01$)									
	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
HDA	23.88	10.95	15.25	13.00	19.61	10.70	39.63	6.38	12.89
CI	55.73	9.96	13.50	10.94	16.98	9.08	37.46	4.90	11.26
MinSum	19.30	9.34	11.77	7.86	14.92	7.82	35.42	4.01	7.57
BPD	19.42	10.32	11.80	17.58	14.90	10.32	38.11	5.56	11.98
CoreHD	19.54	9.65	12.01	13.60	15.35	10.32	38.38	4.94	10.31
GND	25.57	13.42	18.41	11.57	20.73	10.79	47.89	5.62	12.04
FINDER	19.66	9.34	11.93	7.95	15.28	7.81	36.61	4.05	8.28

(B) Running time (s)									
Time (s)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
HDA	0.0	0.0	1.5	2.5	4.5	7.9	12.8	1963.8	5098.1
CI	0.0	0.5	11.0	70.7	9.4	668.4	2723.2	7056.4	>5d
MinSum	2.0	20.0	134.5	323.6	225.4	681.1	1408.0	5478.2	35215.2
BPD	0.3	9.0	31.0	86.0	41.0	254.0	839.0	462.0	17942.0
CoreHD	0.1	2.0	11.5	32.0	9.9	142.9	214.8	1846.1	36837.3
GND	0.1	1.3	55.0	35.7	386.3	233.1	4614.5	52246.2	>5d
FINDER	0.2	1.8	10.3	22.5	31.6	38.9	200.4	410.8	2914.5

Table S19: **FINDER** performance on two power-grid networks.

(A) Vanilla results			(B) NEP-enhanced results		
ANC ($\times 0.01$)	PG-WestState	PG-Texas	ANC ($\times 0.01$)	PG-WestState	PG-Texas
HDA	5.21	5.12	HDA_NEP	1.42	2.09
CI	5.23	4.87	CI_NEP	5.36	5.63
MinSum	6.62	7.61	MinSum_NEP	6.89	8.01
BPD	4.74	4.81	BPD_NEP	5.62	4.86
CoreHD	4.68	4.16	CoreHD_NEP	3.25	4.63
GND	2.20	3.11	GND_NEP	4.53	5.49
FINDER	5.01	4.73	FINDER_NEP	1.09	1.94
FINDER_CM	4.34	4.35	FINDER_CM_NEP	1.06	1.70

Table S20: **Training FINDER using the same type of graphs as the test graphs yields the best performance.**) The node-set sizes of both training graphs and test graphs ranges between 30 to 50. Each result is averaged over 100 test instances, and we report the mean and variance values. The bold ones denote the best of the test graphs. All results here have been multiplied by 100.

(A) Critical node (CN) problem				
Train type	Test type			
		Erdős-Rényi (ER)	Watts-Strogatz (WS)	Barabási-Albert (BA)
Erdős-Rényi (ER)		33.24±0.01	26.03±0.20	18.25±0.32
Watts-Strogatz (WS)		33.25±0.01	24.35±0.19	25.02±0.55
Barabási-Albert (BA)		33.24±0.01	26.55±0.22	17.20±0.27
(B) Network dismantling (ND) problem				
Train type	Test type			
		Erdős-Rényi (ER)	Watts-Strogatz (WS)	Barabási-Albert (BA)
Erdős-Rényi (ER)		47.33±0.22	36.80±0.27	23.28±0.45
Watts-Strogatz (WS)		49.82±0.02	34.98±0.24	30.61±2.33
Barabási-Albert (BA)		47.42±0.21	36.87±0.24	22.84±0.46

Table S21: **Performance of FINDER (trained with different types of synthetic graphs.)** All results here have been multiplied by 100. The bold ones denote the best among the three types of training graphs.

(A) Critical node (CN) problem										
Train type	Real networks	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
Erdős-Rényi (ER)		11.70	5.18	7.42	4.75	8.77	5.62	27.68	1.79	7.06
Watts-Strogatz (WS)		13.62	15.25	10.10	14.47	10.45	11.97	24.82	7.99	14.72
Barabási-Albert (BA)		8.19	3.34	5.90	2.03	7.69	3.05	19.37	1.21	3.60

(B) Network dismantling (ND) problem										
Train type	Real networks	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
Erdős-Rényi (ER)		11.38	7.00	10.48	5.04	10.75	6.87	25.93	4.51	6.76
Watts-Strogatz (WS)		11.71	31.76	27.52	43.63	20.03	33.74	41.30	27.52	30.93
Barabási-Albert (BA)		10.99	5.54	8.66	4.30	11.10	4.99	26.84	2.37	5.46
ER + WS + BA		11.15	7.17	8.88	5.43	10.81	737	44.46	4.32	6.66

Table S22: **ANC scores of various methods, including Configuration Model-trained FINDER: FINDER_CM.** We consider the node-unweighted network dismantling problem here. All results are in accordance with Figure 5(g) in the main text, except that of FINDER_CM , which is trained and selected separately for each networkssh. All results here have been multiplied by 100. The bold blue ones denote the best one (corresponding the lowest ANC score).

ANC ($\times 0.01$)	Crime	HI-II-14	Digg	Enron	Gnutella31	Epinions	Facebook	Youtube	Flickr
HDA	11.62	5.76	8.79	4.52	11.47	5.20	27.20	2.41	5.98
CI	12.43	6.16	8.66	4.45	11.74	5.06	26.95	2.25	5.78
MinSum	13.83	6.52	9.52	4.77	11.73	6.04	27.25	2.57	6.10
BPD	13.95	6.96	9.52	6.18	11.72	7.48	28.35	2.99	8.41
CoreHD	11.33	6.06	8.68	5.14	11.97	5.46	27.47	2.44	6.22
GND	13.81	6.94	10.66	4.56	12.57	5.37	27.42	2.32	5.83
FINDER	10.99	5.54	8.66	4.30	11.10	4.99	26.84	2.37	5.46
FINDER_CM	10.81	5.43	8.52	4.33	10.64	4.82	26.56	2.40	5.34

-
- [1] Pastor-Satorras, R. & Vespignani, A. Epidemic spreading in scale-free networks. *Physical Review Letters* **86**, 3200 (2001).
- [2] Albert, R., Jeong, H. & Barabási, A.-L. Error and attack tolerance of complex networks. *Nature* **406**, 378 (2000).
- [3] Freeman, L. C. A set of measures of centrality based on betweenness. *Sociometry* 35–41 (1977).
- [4] Leskovec, J., Lang, K. J., Dasgupta, A. & Mahoney, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* **6**, 29–123 (2009).
- [5] Holme, P., Kim, B. J., Yoon, C. N. & Han, S. K. Attack vulnerability of complex networks.

- Physical Review E* **65**, 056109 (2002).
- [6] Bavelas, A. Communication patterns in task-oriented groups. *The Journal of the Acoustical Society of America* **22**, 725–730 (1950).
 - [7] Morone, F. & Makse, H. A. Influence maximization in complex networks through optimal percolation. *Nature* **524**, 65 (2015).
 - [8] Brin, S. & Page, L. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems* **30**, 107–117 (1998).
 - [9] Braunstein, A., Dall’Asta, L., Semerjian, G. & Zdeborová, L. Network dismantling. *Proceedings of the National Academy of Sciences* **113**, 12368–12373 (2016).
 - [10] Mugisha, S. & Zhou, H.-J. Identifying optimal targets of network attack by belief propagation. *Physical Review E* **94**, 012305 (2016).
 - [11] Zdeborová, L., Zhang, P. & Zhou, H.-J. Fast and simple decycling and dismantling of networks. *Scientific Reports* **6**, 37954 (2016).
 - [12] Bayati, M. *et al.* Statistical mechanics of steiner trees. *Physical Review Letters* **101**, 037208 (2008).
 - [13] Zhou, H.-J. Spin glass approach to the feedback vertex set problem. *The European Physical Journal B* **86**, 455 (2013).
 - [14] Hagen, L. & Kahng, A. B. New spectral methods for ratio cut partitioning and clustering. *IEEE transactions on computer-aided design of integrated circuits and systems* **11**, 1074–1085 (1992).
 - [15] Ren, X.-L., Gleinig, N., Helbing, D. & Antulov-Fantulin, N. Generalized network dismantling. *Proceedings of the National Academy of Sciences* **116**, 6554–6559 (2019).
 - [16] Chen, Y., Paul, G., Havlin, S., Liljeros, F. & Stanley, H. E. Finding a better immunization strategy. *Physical Review Letters* **101**, 058701 (2008).
 - [17] Von Luxburg, U. A tutorial on spectral clustering. *Statistics and Computing* **17**, 395–416 (2007).
 - [18] Bar-Yehuda, R. & Even, S. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms* **2**, 198–203 (1981).
 - [19] Mises, R. & Pollaczek-Geiringer, H. Praktische verfahren der gleichungsauflösung. *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik* **9**, 152–164 (1929).

- [20] Morone, F., Min, B., Bo, L., Mari, R. & Makse, H. A. Collective influence algorithm to find influencers via optimal percolation in massively large social media. *Scientific Reports* **6**, 30062 (2016).
- [21] Perozzi, B., Al-Rfou, R. & Skiena, S. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 701–710 (ACM, 2014).
- [22] Ying, Z. *et al.* Hierarchical graph representation learning with differentiable pooling. In *Advances in Neural Information Processing Systems*, 4800–4810 (2018).
- [23] Wang, X. *et al.* Community preserving network embedding. In *Thirty-First AAAI Conference on Artificial Intelligence* (2017).
- [24] Grover, A. & Leskovec, J. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 855–864 (ACM, 2016).
- [25] Hamilton, W. L., Ying, R. & Leskovec, J. Representation learning on graphs: Methods and applications. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2017).
- [26] Tang, J. *et al.* Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, 1067–1077 (ACM, 2015).
- [27] Cao, S., Lu, W. & Xu, Q. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, 891–900 (ACM, 2015).
- [28] Defferrard, M., Bresson, X. & Vandergheynst, P. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, 3844–3852 (2016).
- [29] Kipf, T. N. & Welling, M. Semi-supervised classification with graph convolutional networks. In *Int. Conf. on Learn. Represent.(ICLR)* (2017).
- [30] Hamilton, W., Ying, Z. & Leskovec, J. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, 1024–1034 (2017).
- [31] Dai, H., Dai, B. & Song, L. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*, 2702–2711 (2016).
- [32] Chen, T. & Sun, Y. Task-guided and path-augmented heterogeneous network embedding for

- author identification. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, 295–304 (2017).
- [33] Li, Y., Tarlow, D., Brockschmidt, M. & Zemel, R. Gated graph sequence neural networks. In *Int. Conf. on Learn. Represent.(ICLR)* (2016).
 - [34] Xu, K. *et al.* Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*, 5453–5462 (2018).
 - [35] Sutton, R. S., McAllester, D. A., Singh, S. P. & Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, 1057–1063 (2000).
 - [36] Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518**, 529 (2015).
 - [37] Khalil, E., Dai, H., Zhang, Y., Dilkina, B. & Song, L. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems 30*, 6348–6358 (2017).
 - [38] Hessel, M. *et al.* Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).
 - [39] Szepesvári, C. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning* **4**, 1–103 (2010).
 - [40] Sutton, R. S. & Barto, A. G. *Reinforcement learning: An introduction* (MIT press, 2018).
 - [41] Tesauro, G. Temporal difference learning and td-gammon. *Communications of the ACM* **38**, 58–68 (1995).
 - [42] Van Hasselt, H., Guez, A. & Silver, D. Deep reinforcement learning with double q-learning. In *Thirty AAAI Conference on Artificial Intelligence*, vol. 2, 5 (Phoenix, AZ, 2016).
 - [43] Schaul, T., Quan, J., Antonoglou, I. & Silver, D. Prioritized experience replay. In *Int. Conf. on Learn. Represent.(ICLR)* (2016).
 - [44] Sutton, R. S. Learning to predict by the methods of temporal differences. *Machine Learning* **3**, 9–44 (1988).
 - [45] Wang, Z. *et al.* Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning*, 1995–2003 (2016).
 - [46] Bellemare, M. G., Dabney, W. & Munos, R. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, 449–458 (2017).

- [47] Riedmiller, M. Neural fitted q iteration—first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, 317–328 (Springer, 2005).
- [48] Watts, D. J. & Strogatz, S. H. Collective dynamics of ‘small-world’ networks. *Nature* **393**, 440 (1998).
- [49] Wang, D., Cui, P. & Zhu, W. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1225–1234 (ACM, 2016).
- [50] Barabási, A.-L. & Albert, R. Emergence of scaling in random networks. *Science* **286**, 509–512 (1999).
- [51] Kunegis, J. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, 1343–1350 (ACM, 2013).
- [52] Leskovec, J., Kleinberg, J. & Faloutsos, C. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)* **1**, 2 (2007).
- [53] Arulselvan, A., Commander, C. W., Elefteriadou, L. & Pardalos, P. M. Detecting critical nodes in sparse graphs. *Computers & Operations Research* **36**, 2193–2200 (2009).
- [54] Molloy, M. & Reed, B. A critical point for random graphs with a given degree sequence. *Random Structures & Algorithms* **6**, 161–180 (1995).
- [55] Clusella, P., Grassberger, P., Pérez-Reche, F. J. & Politi, A. Immunization and targeted destruction of networks using explosive percolation. *Physical Review Letters* **117**, 208301 (2016).
- [56] Schneider, C. M., Mihaljev, T. & Herrmann, H. J. Inverse targeting—an effective immunization strategy. *EPL (Europhysics Letters)* **98**, 46002 (2012).
- [57] Qin, S.-M., Ren, X.-L. & Lü, L.-Y. Efficient network dismantling via node explosive percolation. *Communications in Theoretical Physics* **71**, 764 (2019).
- [58] Schmidt, C., Pfister, H. D. & Zdeborová, L. Minimal sets to destroy the k-core in random networks. *Physical Review E* **99**, 022310 (2019).
- [59] Kempe, D., Kleinberg, J. & Tardos, É. Maximizing the spread of influence through a social network. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 137–146 (ACM, 2003).