NEGAR: Network Embedding Guided Architecture Recovery for Software Systems

Jiayi Chen*, Zhixing Wang*, Yuchen Jiang*, Jun Pang†§, Tian Zhang*§, Minxue Pan*§, and Jianwen Sun‡
*State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China
†Department of Computer Science, University of Luxembourg, Esch-sur-Alzette, Luxembourg
‡Software Engineering Application Technology Lab, Huawei Technologies, Shenzhen, China
§Corresponding Authors: Jun.pang@uni.lu, ztluck@nju.edu.cn, mxp@nju.edu.cn

***Abstract*—With their rapid development, the scale and complexity of software systems are rapidly growing. Identifying and organizing files of similar functionality into the same module, called architecture recovery, contributes to the maintainability of a software system. However, manual architecture recovery on large-sized software requires unbearable costs. Hence a lot of automatic algorithms have been proposed in recent years. However, current algorithms' accuracy is still insufficient to support practical applications. To improve the accuracy of architecture recovery, this work proposes a novel algorithm, NEGAR, which leverages random walks to extract latent graphic information from the dependency graph of files in the software system and learn the node representation for clustering. The proposed algorithm NEGAR has been comprehensively evaluated on three medium-sized and two large-sized software systems, as well as a super large-sized software system, in terms of four widely-used metrics. The experimental results demonstrate the outstanding accuracy and excellent scalability of NEGAR.**

***Index Terms*—Architecture recovery, software maintenance, network embedding**

I. Introduction

As software grows in scale and complexity, developers need to build a software architecture to enhance its comprehensibility and maintainability [1]. A software architecture consists of several modules, each of which contains a number of basic components that are close in functionality [2]. Hence, the recovery of software architecture is to identify the essential components with similar functions and divide them into the same modules. However, it is not easy to build and maintain good architecture for software due to the rapid evolution in terms of scale and complexity.

For instance, even medium-sized software ranging from 70 KSLOC to 280 KSLOC takes an experienced expert hundreds of hours to recover an accurate ground-truth architecture [3]. Scaling to hyperscale software, even with the aid of semi-automated approaches, it would take an entire team of professionals two years to recover a sound software architecture [4].

Many automated or semi-automated algorithms have been proposed to facilitate the recovery efficiency of the software architecture. According to the input data, the existing architecture recovery algorithms are generally divided into two categories: dependency-based and knowledge-based. Knowledge-based algorithms leverage textual information, such as comments, domain knowledge, design documents, source code, etc., to identify entities with similar functionalities. Although knowledge-based algorithms have been proven to achieve remarkable results on some projects [5], [6], their application scope is limited due to the lack of useful textual information in some projects. Unlike knowledge-based algorithms, dependency-based algorithms use the dependency information extracted from the software. The core inspiration of the dependency-based algorithm is that components with similar functionality tend to have denser interconnections with each other on the dependency graph. Such algorithms first construct dependency graphs from software and then perform specific clustering methods on the dependency graphs. Dependency graphs can describe the associations between software components [7] and have various implementation levels, such as class inheritance, file inclusion, method invocation, etc. Therefore, dependency graphs of suitable granularity can enable dependency-based algorithms to perform better than knowledge-based algorithms [4], [8].

Although the dependency-based algorithm is commonly better than the knowledge-based algorithm in architectural recovery, its overall effectiveness is still far from practical. The main reason for this problem is that existing dependency-based algorithms tend to use a rather traditional clustering idea, which does not fully use the latent structural information in the dependency graph. For example, ACDC [9] uses a simple graph structure pattern recognition method to identify software modules; FCA [10] clusters nodes by simple matrix operations; Bunch [11] takes the cohesive and coupling property of the architecture as the optimization goal to find the clustering results that satisfy the maximization of the objective function. Unfortunately, the clustering process of these algorithms is often restricted to a small range of local structures, and the modules obtained by clustering are connected subgraphs. However, this goes against the properties of dependency graphs constructed from real software: identical nodes are not always directly connected. Instead, they may be separated by other nodes or distributed in different subgraphs at a slight distance.

To better utilize the structural information in the dependency graph, we use a graph node representation learning technique from deep learning called *network embedding*. The network embedding technique aims to mine the latent structural information from non-Euclidean graphs to learn the vector representation of graph nodes in multidimensional vector

space [12].

Different network embedding techniques exist, and one widely-used approach is based on random-walk statistics, such as DeepWalk [13] and node2vec [14]. These random-walk-based methods use a stochastic and flexible node similarity metric that allows a larger exploration of the graph's local structure, enabling nodes with similar local neighborhood graphs to get closer vector representations. Such techniques have achieved wide applications in social network analysis and have shown superior performance on downstream tasks such as classification and clustering [15]. However, as far as we know, there exists no related work that applies random walk-based network embedding techniques to the recovery of software architectures.

The primary contributions of this work can be summarized as follows.

- A dependency-based architecture recovery algorithm NEGAR is proposed, which utilizes a controllable random-walk method to sample the statistical information of the dependency graph structure and learns a node vector representation for clustering using the language model SkipGram.
- A comprehensive experimental evaluation is conducted to assess the performance of NEGAR. The results show that the proposed algorithm outperforms the chosen state-of-the-art algorithms in terms of accuracy and quality of the recovered architecture.
- We further evaluate the scalability of NEGAR on a super large-scale software, and the results show that NEGAR can still achieve excellent performance in an acceptable time cost.

The rest of the paper is organized as follows. Section 2 introduces several state-of-the-art architecture recovery algorithms. Section 3 describes the algorithm design and implementation of NEGAR. Section 4 details the experimental evaluation. Section 5 concludes the paper with future work.

II. Related Work

Existing architecture recovery algorithms can generally be divided into two major categories according to the source of input data: dependency-based [7], [10], [16]–[33] and knowledge-based [5], [6] algorithms.

Dependency-based algorithms leverage structural information for modularization, which can be further classified into hierarchical and search-based categories. Hierarchical algorithms utilize a greedy strategy to merge similar clusters. WCA (Weighted Combined Algorithm) [34] measures the inter-cluster distance between different clusters and merges close clusters into a new larger cluster. It starts with each node as an individual cluster associated with a feature vector. Then two most similar clusters are merged into a larger cluster based on a specific inter-cluster distance measurement and the feature vector of the new cluster is recalculated. This process repeats until the clusters reaches the number set in advance. There are two inter-cluster distance measurements for WCA: Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM (UENM). LIMBO [35] scales the Information Bottleneck algorithm to large data sets. It regards each node as an individual cluster and summarizes all clusters into a Distributational Cluster Feature (DCF) tree. Then leaves of the DCF tree are merged using the Information Bottleneck algorithm to produce a specific number of clusters. At last, the original nodes are assigned to a cluster. In search-based algorithms, the architecture recovery is transformed into an optimization problem [36] with the objective for maximizing a quality function. Bunch is a well-known search-based algorithm, whose objective function is called Modularization Quality (MQ), that measures the extent of cohesion and coupling of clusters. Different searching strategies are employed to reduce the algorithmic complexity, such as genetic algorithm (GA), nearest and steepest ascent hill climbing algorithms (NAHC and SAHC) and so on. Although some heuristic methods are leveraged to reduce the time complexity, the time cost of search-based algorithms is still unbearable in terms of large software systems [4], [10]. Some other algorithms are special and do not belong to the above two kinds of algorithms. FCA [10] is proposed to quickly recover the architecture while the clustering quality is guaranteed. The core idea of FCA is to start clustering from the border nodes, which are prioritized through specific mathematical operations on special matrices derived from the dependence matrix. ACDC (Algorithm for Comprehension-Driven Clustering) [9] is a comprehension-driven algorithm recovering architectures based on identify of specific sub-system patterns that are closer to the view of real software development, which are more suitable for human comprehension.

Different from dependency-based algorithms, knowledge-based algorithms utilize textual information to implement architecture recovery. Architecture Recovery Using Concerns (ARC) [6] leverages information retrieval and machine learning methods. ARC regards the program as textual documents and leverages a statistical language model called Latent Dirichlet Allocation (LDA) [37] to extract concerns from identifiers and comments of the program code. Concerns represent concepts of the software system and can be used to identify different clusters. Zone Based Recovery (ZBR) [5] utilizes natural language semantics of textual data in the program. Each file is regarded as a textual document and divided into several zones. ZBR evaluates the term frequency-inverse document frequency (TF-IDF) score for each word in a zone and uses the Expectation-Maximization algorithm to weigh each zone. ZBR leverages group average agglomerative algorithm for clustering. There are two variants of ZBR [4]. One variant uses uniform weights for each zone, and the other uses the ratio of the number of tokens in the entire software program as the weights.

III. Proposed Algorithm

A. Overview of Framework

Dependency graph for file granularity can express potential relationship between different files that is useful for identifying nodes of the same functionality. Nodes belonging to the same cluster tend to be densely clustered in local subgraph. But

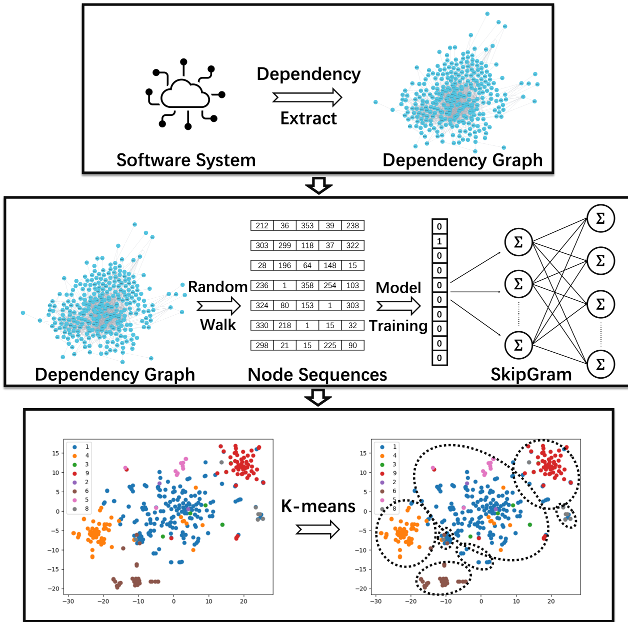

Fig. 1. Illustration of the framework of NEGAR.

these nodes are not necessarily highly interconnected, which may be separated by nodes belonging to other clusters. To exploit this latent dependency graphic properties, we have proposed a novel algorithm NEGAR in this work, whose framework is presented in Fig. 1. The input is an dependency graph extracted from the software system. Then the algorithm utilizes a controllable random walk to sample node sequences, which are used to train a natural language model SkipGram. SkipGram aims to learning a embedding function which embeds nodes into a low-dimensional vector space based on their context in the sampled node sequences. After obtaining the vectors for all nodes, we leverage k-means algorithm for clustering.

The input to our algorithm is a dependency graph extracted from the software, which is independent with the exact dependency extraction process. There are a lot of related work and tools to extract dependency graphs of different granularity. For example, Understand toolset[1] can extract call and include dependencies, the proposed technique proposed in previous work [38] can generate precise dependencies based on analysis on LLVM bitcode, etc. In this work, we directly utilize the public dependencies provided by the authors of [4], hence we skip the details of dependency graph extraction.

In the clustering process, we apply the classical k-means algorithm. Some related works [4], [39] use direct neighbor vectors as the input to k-means and have achieved good results on some dependency graphs. However, the direct neighbor vectors contain only the first-order neighborhood information of nodes, which cannot perceive a larger range of graph structure information. In addition, the vector length can be too large for clustering on large-scale graphs. These factors limit the effectiveness of clustering performance. In this work, we expect to use a different node representation as input to k-means. The following subsections concentrate on description of our network embedding method, which consists of two parts: random walk sampling and representation learning.

[1]https://scitools.com/

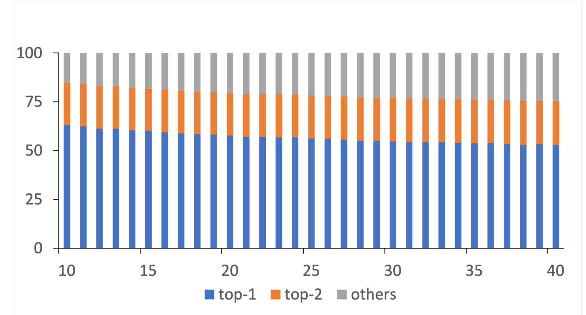

Fig. 2. The proportion of homogeneous nodes in the sequences sampled from Bash's Symbol Dependency graph. The vertical axis represents the proportion, and the horizontal axis represents the length of the node sequence.

B. Random Walk Sampling

Random walk is a stochastic and flexible sampling algorithm to extract structural statistics from a graph and measure similarity between nodes, which is used in various research fields such as content recommendation and community detection. To observe the characteristics of random walk, we sampled a set of node sequences at different lengths on Symbol Dependency graph of Bash using uniform random walk. We calculated the percentage of nodes belonging to the same cluster in each sequence and take the average value. As shown in Fig. 2, the top-1 proportion of nodes belonging to a same cluster is bigger than 50% and much larger than the others. This statistical feature can be used to measure the similarity between nodes.

The sampled sequence by random walk started from node v_i can be denoted as:

$$\mathcal{W}_{v_i} = \mathcal{W}_{v_i}^1, \mathcal{W}_{v_i}^2, \ldots, \mathcal{W}_{v_i}^\ell,$$

where $\mathcal{W}_{v_i}^1$ is the root node v_i and ℓ represents the walk length. When $\mathcal{W}_{v_i}^j$ is the currently visiting node, the next node to be visited is chosen among the direct neighbors of $\mathcal{W}_{v_i}^j$ based on specific sampling strategy.

There are two kinds of random walk strategy utilized in DeepWalk [13] and node2vec [14]. The random walk in DeepWalk uses a uniform random strategy, i.e., the next selected node is chosen with medium probability from the neighbors of the current node. node2vec leverages a controllable random walk that pursues a tradeoff between BFS and DFS. Assuming the root node is u, BFS and DFS are explained as following [14]:

1) **Breadth-first Strategy** (BFS): The breadth-first strategy tends to sample the direct neighbors of the root node u. BFS concentrates on the microscopic view of partial structures, as it restricts sampling scope to the direct neighborhoods of the root nodes. The sampled statistics

of BFS reflect the homophily [40] of node similarity, where the nodes that are highly inter-connected and belong to the same cluster have higher probability of co-occurrence in a traverse sequence.

2) **Depth-first Sampling** (DFS): The depth-first strategy tends to sample nodes with increasing distance from the root node u. DFS can explore further away from the root node and obtain a macroscopic view of the sub-graph. The sampled statistics of DFS focus on the structural equivalence [41] . In contrast with homophily, structural equivalence does not stress connectivity, where nodes that are far apart in the graph can still belong to the same cluster.

In this work, we utilize the controllable random walk in node2vec [14] based on the observation that nodes in real dependency graph may exhibit both homophily and structural equivalence and the uniform random walk in DeepWalk is actually a specific case of the controllable random walk. Under controllable random walk, given the root node u and the currently visiting node t, the probability of node x being sampled next follows the distribution:

$$P(c_i = x | c_{i-1} = t) = \begin{cases} \frac{\pi_{tx}}{Z} & if\ (t, x) \in \mathcal{E} \\ 0 & otherwise \end{cases} \tag{1}$$

where π_{tx} is an unnormalized transition probability between nodes t and x, Z is the normalizing factor and $\mathcal{E}$ is the set of dependency edges.

The controllable random walk leverages a 2nd order strategy to guide the random process, which is controlled by two hyper-parameters p and q. Assuming the random walk resides in the node t by previous traversing through edge (v, t), the unnormalized transition probability to node x is defined as $\pi_{tx} = \alpha_{pq}(v, x) * \omega(t, x)$, where

$$\alpha_{pq}(v, x) = \begin{cases} \frac{1}{p} & if\ d_{vx} = 0 \\ 1 & if\ d_{vx} = 1 \\ \frac{1}{q} & if\ d_{vx} = 2 \end{cases} \tag{2}$$

$\omega(t, x)$ represents the weight of edge (t, x) and d_{vx} is the minimun distance from node t to x. If the graph is unweighted, we have $\pi_{tx} = \alpha_{pq}(v, x)$.

Hyper-parameters *p* and *q* control the behavior of random walk. *p* is *return parameter* controlling likelihood of immediately revisiting the previously visited node. If $1/p$ is less than $\min(1, 1/q)$, the random walk is less likely to choose the node just visited. In contrast, if $1/p$ is higher than $\max(1, 1/q)$, the random walk tends to backtrack the traversing sequence, which indicates the explored sub-graph is more compact with the root. *q* is *in-out parameter* controlling the depth of random walk. If $1/q$ is higher than $\max(1, 1/p)$, the random walk will be more inclined to explore further away from the root node. In contrast, if $1/q$ is less than $\min(1, 1/p)$, the random walk tends to obtain a microscopic view of partial graph. If $1/p$ and $1/q$ are both much less than 1, the random walk will have higher probability on visiting direct neighborhoods of the root node, which is a BFS-like exploration.

C. Representation Learning

SkipGram [42] is a widely-used language model for word representation learning, whose input is a corpus consisting of a large number of sentences. SkipGram mines the corpus for potential statistical information, so that words with similar contextual content can learn to get closer vector representations. Homogeneous nodes occupy the majority of sampled node sequences by random walk as shown in Fig. 2, thus can be used as the corpus instead. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ represents the unweighted and undirected graph, the embedding function is simply an mapping process:

$$\mathbf{z}_i = \phi(v_i) = \mathbf{Z}\mathbf{v}_i \tag{3}$$

where $\mathbf{Z} \in \mathbb{R}^{d \times |\mathcal{V}|}$ is a matrix containing $|\mathcal{V}|$ embedding vectors, d is the number of vector dimensions and $\mathbf{v}_i \in \mathbb{I}_{\mathcal{V}}$ is the one-hot vector indicating which column of matrix $\mathbf{Z}$ (i.e., $\mathbf{z}_i$) is the corresponding vector of v_i. The trainable parameters of embedding function is $\theta_\phi = \{\mathbf{Z}\}$, i.e., the embedding vectors for all nodes are optimized directly during the training process.

The corpus of *SkipGram* is constructed from the node sequence sampled by random walk. Given a node sequence $\{\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_\ell\}$ from the corpus and assigned a node $v_i \in \{\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_\ell\}$, the context of v_i is defined as the nodes appearing in a window of length w on both sides of v_i in the sequence. The goal of *SkipGram* is to maximize the co-occurrence probability of pairs of node appearing in the same context based on an independence assumption

$$Pr(\mathcal{N}_\mathcal{S}(v_i, w) | v_i) = \prod_{\substack{v_j \in \mathcal{N}_\mathcal{S} \\ j \neq i}} Pr(v_j | v_i) \tag{4}$$

where $\mathcal{N}_\mathcal{S}(v_i, w) = \{v_{i-w}, \ldots, v_{i+w}\} \backslash v_i$ is the context of v_i.

The co-occurrence probability between two nodes can be calculated using *softmax* algorithm:

$$Pr(v_j | v_i) = \frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{v_k \in \mathcal{V}} e^{\mathbf{z}_i^\top \mathbf{z}_k}} \tag{5}$$

Formally, the objective function of SkipGram can be concluded as the following cross-entropy loss:

$$\min_\phi \sum_{\substack{v_i \in \mathcal{V} \\ v_j \in \mathcal{N}_\mathcal{S}(v_i) \\ j \neq i}} -\log Pr(v_j | v_i) \tag{6}$$

Calculating the denominator of Softmax formula over a large graph is a time-consuming operation as it requires $|\mathcal{V}|$ times of calculation. There are two primary methods to reduce the time complexity: Hierarchical Softmax and Negative Sampling, which are used in DeepWalk [13] and node2vec [14] respectively. Hierarchical Softmax leverages Huffman-tree to reduce the time complexity to $\mathcal{O}(\log |\mathcal{V}|)$; Negative sampling approximates the normalizing factor to avoid calculation over all nodes. In this work, we tried both of these two approaches.

D. Algorithm: NEGAR

This section presents the algorithm implementation of *NEGAR*. The pseudocode is represented in Algorithm 1. Lines 1-2 initialize the embedding function ϕ randomly and the corpus *walks* to store traverse sequences sampled by random walk. Lines 3-9 contain two layers of loops, where the outer loop controls the number of walks per node, and the inner loop performs a random walk sampling started from each node. The choice of start node has an implicit bias on the context of nodes. To offset the bias, the order of node set $\mathcal{V}$ will be shuffled before sampling, and the random walk will be simulated γ times per node. We leverages the controllable random walk strategy that is controlled by hyper-parameters p and q. The controllable probability is implemented using alias sampling algorithm in $\mathcal{O}(1)$ time complexity. Line 10 employs the node sequence sampled by random walk as the corpus to train the SkipGram model. For each sequence $\{\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_\ell\}$ in corpus, SkipGram extracts the context centered on each node $\mathcal{W}_i$ ($1 \leq i - 2\omega$ && $i + 2\omega \leq \ell$) and uses the loss function in Eq. 6 to optimize the embedding function. Lines 11-12 obtain embedding vectors for all nodes in $\mathcal{V}$ and use k-means algorithm to cluster the vectors into k clusters.

Algorithm 1 NEGAR

Input: Unweighted Dependency Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
- Return parameter p
- In-out parameter q
- Walks per node γ
- Walk length ℓ
- Window size ω
- Embedding Dimensions d
- Epochs over the corpus η
- Cluster number k
- Optimization method opt

Output: Recovered architecture **A**

1: Encoder initialization: $\phi \in \mathcal{R}^{|\mathcal{V}| \times d}$
2: $walks = \varnothing$
3: **for** $i = 0$ to γ **do**
4:     $\mathcal{O} =$ Shuffle($\mathcal{V}$)
5:     **for all** $v_i \in \mathcal{O}$ **do**
6:         $\mathcal{W} =$ RandomWalk($\mathcal{G}, v_i, \ell, p, q$)
7:         Append $\mathcal{W}$ to $walks$
8:     **end for**
9: **end for**
10: SkipGram($\phi, walks, \omega, \eta, opt$)
11: Obtain embeddings: $embeds = \{\phi(v_i) | v_i \in \mathcal{V}\}$
12: Clustering: $arch =$ k-means($embeds, k$)
13: **return** $arch$

IV. Experiments

In this section, we introduce our experimental setup and the configuration of algorithmic parameters. Then we propose three research questions and answer each based on an analysis of the experimental results.

TABLE I
Evaluated Software Systems

Project	Version	Description	Language	SLOC
Bash	4.2	Unix Shell	C	115K
Hadoop	0.19.0	Data Processing	Java	87K
ArchStudio	4	Architecture Development	Java	55K
ITK	4.5.2	Image Segmentation Toolkit	C++	1M
Chromium	svn-171054	Web Browser	C++	10M

A. Experimental Setup

To ensure the validity of the experimental conclusions, we adopted the experimental setup from a previously published work [4]. The tested projects include three medium-sized, one large-sized, and one super large-sized software, whose details are listed in Table I. We used four metrics to evaluate the architecture recovery algorithm's effectiveness: a2a, MoJoFM, c2c with a threshold of 0.33, and Normalized TurboMQ [4]. The a2a, MoJoFM, and c2c quantify the similarity of the recovered architecture recovered by the algorithms to the ground truth architecture, where the credible ground truth architectures of selected projects are provided by previous work [4]. Normalized TurboMQ measures the architecture's degree of cohesion and coupling, which does not rely on ground truth. Higher results of the above metrics indicate better performance of the algorithms. We choose 12 variants of state-of-the-art algorithms for comparison, where 8 are dependency-based and 4 are knowledge-based.

TABLE II
Experimental hyper-parameters configuration of NEGAR in terms of different projects.

Project	ℓ	ω	d	p	q	γ	k	η	opt
Bash	[10,40]	[5,20]	128	0.25,0.5,1,2,4	0.25,0.5,1,2,4	200	14	5	hs,ns
ArchStudio	[10,40]	[5,20]	128	0.25,0.5,1,2,4	0.25,0.5,1,2,4	200	57	5	hs,ns
Hadoop	[10,40]	[5,20]	128	0.25,0.5,1,2,4	0.25,0.5,1,2,4	200	67	5	hs,ns
ITK	40	20	128	1	1	100	50	1	hs
Chromium	40	20	128	1	1	100	67	1	hs

B. Algorithmic Parameters

To explore the theoretical optimal performance for NEGAR, we leverage the grid-searching approach for hyper-parameters tuning and chosen the best value for specific metric as the result. The tuned range of hyper-parameters is listed in Table II.

For Bash, ArchStudio and Hadoop, because of their medium size, we tuned in a large parameter space. The walk length ℓ ranged from 10 to 40, and the window size ω was set as the half of ℓ. Besides, the return parameter p and in-out parameter q were tuned on $\{0.25, 0.5, 1, 2, 4\}$ to investigate whether different random walk strategies will improve the effect of architecture recovery. In addition, we both tried Hierarchical Softmax (hs) and Negative Sampling (hs) approaches to optimize the calculation of softmax.

For ITK and Chromium, it consumed NEGAR a long time to recover the architecture because of the large scale of the dependency graph. Hence we obtained the results under a fixed configuration of hyper-parameters and decreased the value of γ and η.

C. Research Questions

This work mainly focuses on the following three research questions:

RQ1. Can the proposed algorithm NEGAR achieve superior effectiveness in terms of different metrics than the existing algorithms?

RQ2. Can the proposed algorithm NEGAR scale to a super large project comprising 10MSLOC?

RQ3. Is there a remarkable statistical improvement in the results obtained by the proposed algorithm NEGAR and the selected algorithms?

All experiments were conducted on a server with Intel(R) Xeon(R) W-2245 CPU @ 3.90GHz and 32GB of memory.

D. Analysis of Results

The results are presented in Tables III, IV, V, VI, and VII. The first column represents different algorithms, and the other columns represent results on different implementation levels of dependency in terms of specific metrics. Each granularity's dependency has four values corresponding to a specific metric for dependency-based algorithms. **As knowledge-based algorithms [4] are independent of dependency, they have unique result corresponding to each metric in a project.** Because Normalized TurboMQ requires the corresponding dependency graph of the recovered architecture, knowledge-based algorithms, and Dir.Struc method cannot calculate the value for Normalized TurboMQ and is marked as not applicable (NA). Besides, in large-sized and super-large-sized projects, some algorithms failed to obtain results due to time out (TO) and memory overflow (MEM) problems. We also tried LINE [43] as the network embedding method, another widely-used method using first and second order neighborhood to measure node similarity. However, the exploration of LINE is restrictive and is unable to explore nodes at further depths than DeepWalk and node2vec. Its performance is less satisfactory, thus we didn't present its results.

RQ1 is answered based on Table III, IV, V, VI, VII from three aspects:

Comparison with dependency-based algorithms. Comparison with dependency-based algorithms. Dependencies of different implementation levels have different effects on the architecture recovery. For example, some algorithms may prefer particular kinds of dependencies while other algorithms can achieve better performance on other dependencies. Hence, an algorithm can be considered superior to others only if it can achieve consistently better results in most dependencies in the same project. **We marked the top-1 result with gray color for each level of dependency.** As shown in the results, NEGAR achieves top-1 scores on the majority of dependencies, except that ACDC performs a little bit better than NEGAR in ArchStudio. **This indicates that NEGAR has a comprehensively better effect than the selected dependency-based algorithms.**

Comparison with knowledge-based algorithms. Since knowledge-based algorithms (ARC and ZBR) have only one group of experimental results on each project, while dependency-based algorithms have multiple groups of results, we compare the results obtained on each level of dependency between NEGAR and knowledge-based algorithms. For Symbol dependency (Sym, S-CHA, S-Int, No DyB), NEGAR achieves consistently better results on all five projects. For Transitive dependency (Trans), obtained by calculating the closure of Symbol dependency, NEGAR is better than the knowledge-based algorithms on all projects other than Hadoop. For Include dependency (Inc), NEGAR can achieve better results on three C/C++ projects (Bash, ITK, Chromium). NEGAR's results on ArchStudio are better on MoJoFM and c2c, but the value of a2a is worse than the average value of the knowledge-based algorithms, and the results of all the metrics on Hadoop are worse than the knowledge-based algorithms. For Function-Call (Funct) and Function-Call-and-Global-Variable-Use (F-GV) dependencies, the results of NEGAR on ArchStudio, Hadoop, and Chromium are better than knowledge-based algorithms, but it is in contrast on Bash and ITK. **In conclusion, utilizing favorable implementation-level dependencies, NEGAR can achieve a better architectural recovery effect than knowledge-based algorithms.**

Comparison with baselines. The primary difference between NEGAR and DN+K-means is the node vector used for input. DN+K-means leverages the direct neighborhood (DN) as the node vector, while NEGAR utilizes random walk to sample the graphic statistics and leveraged SkipGram for representation learning. **As shown in the results, NEGAR significantly improves almost all results more than DN+K-means, which indicates NEGAR can extract better latent graph structure information.** Dir.Struc. utilizes the directory structure as the approximate architecture. Overall, the results of NEGAR are closer to Dir.Struc. than other algorithms, where NEGAR achieved a better effect on C/C++ projects but Dir.Struc. was better on Java projects. **This indicates that the recovered architecture by NEGAR may be more similar to the actual directory structure of the software.**

RQ2 is answered by analysis of the algorithmic performance of Chromium. Because part of the experimental data is referenced from previous work [4] and the exact results of running time are not provided, we discuss it based on our experimental results and publicly available data. Different levels of dependency consume different amounts of running time and memory. The largest graph of Chromium is the Include dependency graph, whose number of edges exceeds one million and the number of nodes reaches more than twenty thousand. The single running time of NEGAR on the Include dependency graph is around 14 hours. Running on other levels of dependency consumes 5-6 hours. Other algorithms are diverse in running time. Previous work [10] merged multiple dependencies and measured performance of different algorithms on this converged dependency graph. According to its results, FCA and ACDC were the top-2 scalable algorithms, which consumed 8 and 10 hours. In our

TABLE III
RESULTS OF MOJOFM (M)(%), A2A (A)(%), C2C (C)(%) AND NORMALIZED TURBOMQ (T)(%) FOR BASH

Algorithm	Bash																			
	Inc				Sym				Trans				Funct				F-GV			
	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T
ACDC	52	65	47	9	57	80	77	22	38	80	40	6	49	41	7	29	50	41	7	29
Bunch-NAHC	53	68	20	25	43	84	20	31	34	83	10	20	49	41	10	33	46	41	9	28
Bunch-SAHC	57	69	23	30	52	85	37	30	34	83	13	20	43	40	6	28	49	41	11	28
WCA-UE	29	64	10	0	24	80	8	4	27	80	10	4	34	39	4	5	33	39	6	5
WCA-UENM	29	64	10	0	23	80	8	4	25	80	7	4	32	39	10	5	31	39	6	6
LIMBO	32	61	4	3	27	77	5	9	28	77	4	4	17	37	0	4	17	37	0	4
DN+K-means	59	67	19	0	55	84	28	17	49	84	26	6	47	41	5	14	46	40	2	16
FCA	60	67	29	15	47	77	43	19	44	82	14	9	40	38	7	15	39	39	7	14
NEGAR	72	72	37	35	71	91	57	59	53	87	29	26	62	43	23	43	62	43	21	43
ARC					43					67					20					NA
ZBR-tok					41					71					7					NA
ZBR-uni					29					70					0					NA
Dir.Struc.					57					64					36					NA

TABLE IV
RESULTS OF MOJOFM (M)(%), A2A (A)(%), C2C (C)(%) AND NORMALIZED TURBOMQ (T)(%) FOR ARCHSTUDIO

Algorithm	ArchStudio																											
	Inc				S-CHA				S-Int				No DyB				Trans				Funct				F-GV			
	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T
ACDC	60	71	21	66	60	86	54	41	77	88	77	76	78	83	75	84	71	92	72	71	75	87	64	72	74	88	62	74
Bunch-NAHC	48	69	5	72	40	80	6	42	49	81	11	74	47	75	10	85	40	80	5	35	53	81	13	74	46	81	9	75
Bunch-SAHC	54	70	10	71	39	80	7	41	53	82	13	76	40	74	8	85	38	80	7	50	53	81	11	72	54	82	19	74
WCA-UE	30	70	5	1	30	83	5	11	32	84	14	22	45	81	35	65	32	83	13	15	31	82	7	10	31	83	8	19
WCA-UENM	30	70	5	1	30	83	5	11	32	84	14	22	45	81	35	65	33	84	13	15	31	82	7	10	31	83	8	19
LIMBO	23	67	0	2	23	79	0	12	24	79	0	31	25	74	0	38	24	78	0	7	24	77	0	24	23	78	0	27
DN+K-means	44	70	25	13	37	81	18	21	39	82	24	38	41	77	23	51	43	83	33	29	39	81	22	35	38	82	24	39
FCA	48	69	35	34	39	77	46	33	50	78	68	57	53	73	70	66	46	78	67	37	49	76	58	52	49	77	67	56
NEGAR	66	72	37	64	59	85	42	62	72	87	63	86	75	83	66	93	68	88	69	82	72	85	54	85	70	87	57	87
ARC							56							84							29							NA
ZBR-tok							48							85							16							NA
ZBR-uni							48							86							23							NA
Dir.Struc.							88							87							91							NA

TABLE V
RESULTS OF MOJOFM (M)(%), A2A (A)(%), C2C (C)(%) AND NORMALIZED TURBOMQ (T)(%) FOR HADOOP

Algorithm	Hadoop																											
	Inc				S-CHA				S-Int				No DyB				Trans				Funct				F-GV			
	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T
ACDC	24	68	3	48	29	81	13	28	41	84	18	59	41	79	18	65	28	80	10	29	41	84	16	57	41	84	16	58
Bunch-NAHC	23	67	3	40	21	79	1	26	24	80	4	53	24	76	4	61	17	78	3	17	26	80	5	52	26	80	5	48
Bunch-SAHC	24	67	3	40	26	80	7	31	28	81	6	53	26	76	6	61	20	79	3	18	29	81	8	54	28	81	5	56
WCA-UE	13	68	7	1	12	80	12	5	15	80	12	8	28	78	9	34	17	81	12	7	17	81	15	6	17	81	12	8
WCA-UENM	13	68	7	1	12	80	12	5	15	80	12	8	28	78	9	33	17	81	12	7	17	81	15	6	17	81	12	8
LIMBO	15	67	0	2	13	79	0	7	14	79	0	19	14	75	0	25	13	79	0	2	13	78	0	17	14	79	0	17
DN+K-means	30	70	22	11	25	81	15	13	29	81	19	29	28	77	16	34	29	82	14	9	29	82	18	26	29	82	19	27
FCA	28	68	36	26	27	76	46	18	34	77	49	39	37	73	49	50	22	81	16	23	35	77	46	38	36	77	49	39
NEGAR	37	71	15	48	49	83	28	72	53	85	32	65	52	81	30	72	35	84	18	37	53	85	32	62	53	85	33	64
ARC							35							82							24							NA
ZBR-tok							29							81							16							NA
ZBR-uni							38							83							23							NA
Dir.Struc.							63							88							45							NA

experiments, we tested the fastest-of-the-art algorithm (FCA) and took around 10 hours on the Include dependency graph and around 3-6 hours on other dependencies, consistent with its results. Other algorithms would consume a much longer time. For example, WCA and DN+K-means consumed more than 30 hours. On the other hand, bunch-SAHC and LIMBO timed out after 24 and 8 days. Moreover, ZBR ran out of 40 GB of RAM in processing for knowledge-based algorithms, and ARC had a similar execution time as WCA [4]. **In summary, although NEGAR's execution is not as efficient as the fastest algorithms, it can still be applied to large-scale software architecture recovery tasks.**

RQ3 is answered using Cliff's δ effect size metric, a statistic tool to quantify the difference between two groups of values (NEGAR versus other algorithms). The value of Cliff's δ ranges from -1 to 1. A positive value indicates that the results of NEGAR are better than those of another algorithm. A larger absolute value indicates a more pronounced difference in the distribution of the results. We utilize the following representative magnitudes [10] to interpret the difference: negligible (N) ($|\delta| < 0.147$), small (S) ($|\delta| < 0.33$), medium (M) ($|\delta| < 0.474$) and large (L) ($|\delta| \geq 0.474$). The results are listed in Table VIII, which show that **NEGAR is significantly better than almost all the other algorithms in terms of**

TABLE VI
RESULTS OF MOJOFM (M)(%), A2A (A)(%), C2C (C)(%) AND NORMALIZED TURBOMQ (T)(%) FOR ITK

Algorithm	ITK																							
	Inc				S-CHA				S-Int				No DyB				Funct				F-GV			
	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T
ACDC	59	67	0	33	55	74	0	24	52	63	0	18	48	58	0	32	60	48	8	40	60	48	8	40
Bunch-NAHC	37	71	0	15	36	78	0	23	35	68	0	23	35	58	0	22	45	47	0	34	47	47	0	37
Bunch-SAHC	32	69	0	10	46	78	2	29	43	66	0	23	41	57	0	21	54	48	0	44	53	47	0	37
WCA-UE	30	73	0	2	31	82	0	5	44	47	8	1	45	38	0	1	35	48	0	5	35	48	0	5
WCA-UENM	30	73	0	2	31	82	0	5	44	47	8	1	45	38	0	1	35	48	0	5	35	48	0	5
LIMBO	30	69	0	4	31	76	0	6	44	43	0	3	36	36	0	1	35	45	0	4	35	45	0	4
DN+K-means	38	74	0	13	42	82	8	24	39	71	3	15	43	61	9	13	60	51	6	31	61	51	8	25
FCA	67	66	0	34	63	71	0	23	48	63	0	14	44	55	0	16	52	45	0	22	52	45	0	22
NEGAR	69	73	15	82	57	79	15	82	60	70	8	75	54	59	0	79	62	49	15	82	61	48	8	81
ARC						24						54						0						NA
ZBR-tok						MEM						MEM						MEM						NA
ZBR-uni						MEM						MEM						MEM						NA
Dir.Struc.						59						61						0						NA

TABLE VII
RESULTS OF MOJOFM (M)(%), A2A (A)(%), C2C (C)(%) AND NORMALIZED TURBOMQ (T)(%) FOR CHROMIUM

Algorithm	Chromium																							
	Inc				S-CHA				S-Int				No DyB				Funct				F-GV			
	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T	M	A	C	T
ACDC	64	71	30	15	70	73	45	19	73	74	37	18	71	64	23	20	71	62	17	24	71	62	17	24
Bunch-NAHC	28	69	0	4	31	73	0	24	24	76	0	9	29	66	0	26	29	63	0	16	35	63	3	19
Bunch-SAHC	12	60	6	2	71	71	33	30	43	66	12	11	42	66	10	23	39	64	1	29	29	62	0	11
WCA-UE	23	70	0	0	23	75	0	2	23	78	0	2	27	68	0	2	29	66	0	2	29	66	0	2
WCA-UENM	23	70	0	0	23	75	0	2	23	78	0	2	27	68	0	2	29	66	0	2	29	66	0	3
LIMBO	TO	TO	TO	TO	23	70	0	2	3	73	0	2	26	64	0	2	27	61	0	2	27	61	0	2
DN+K-means	40	71	7	0	42	74	6	17	43	77	8	13	43	67	5	19	45	65	5	22	45	65	4	22
FCA	44	67	10	6	53	66	6	11	52	70	6	10	51	62	9	12	53	60	9	15	52	60	9	15
NEGAR	68	77	30	79	66	79	35	75	73	82	39	68	71	71	38	82	71	70	36	85	72	70	36	86
ARC						54						54						3						NA
ZBR-tok						MEM						MEM						MEM						NA
ZBR-uni						MEM						MEM						MEM						NA
Dir.Struc.						69						60						7						NA

TABLE VIII
RESULTS OF CLIFF'S δ EFFECT SIZE TEST (NEGAR VERSUS OTHERS) IN TERMS OF MOJOFM (M)(%), A2A (A)(%), C2C (C)(%) AND NORMALIZED TURBOMQ (T)(%).

Algorithm	M	A	C	T
ACDC	0.2 (S)	0.16 (S)	0.19 (S)	0.7 (L)
Bunch-NAHC	0.92 (L)	0.33 (S)	0.89 (L)	0.74 (L)
Bunch-SAHC	0.79 (L)	0.34 (M)	0.8 (L)	0.71 (L)
WCA-UE	0.99 (L)	0.29 (S)	0.86 (L)	0.97 (L)
WCA-UENM	0.99 (L)	0.29 (S)	0.86 (L)	0.97 (L)
LIMBO	0.99 (L)	0.44 (M)	0.97 (L)	0.99 (L)
DN+K-means	0.83 (L)	0.25 (S)	0.65 (L)	0.96 (L)
FCA	0.78 (L)	0.45 (M)	0.17 (S)	0.89 (L)
ARC	0.74 (L)	0.34 (M)	0.6 (L)	NA
ZBR-tok	0.95 (L)	0.37 (M)	0.79 (L)	NA
ZBR-uni	0.5 (L)	0.34 (M)	0.74 (L)	NA
Dir.Struc.	-0.13 (N)	0.07 (N)	0.01 (N)	NA

MoJoFM, a2a, c2c and Normalized TurboMQ. Besides, the gap between ACDC and NEGAR is small in terms of MoJoFM, a2a and c2c, and the gap between FCA and NEGAR is small in terms of c2c. This shows that ACDC and FCA also have good effects in some aspects. Regarding baseline algorithms, NEGAR is superior to K-means, which proves the effectiveness of the random walk-based node representation method. The gap between NEGAR and Dir.Struc. is negligible, which indicates that the recovered architecture is close to the directory structure of the objective projects.

V. CONCLUSION

Organizing similar files into the same cluster, called architecture recovery, benefits the understandability and maintainability of software. This paper proposed a novel algorithm, NEGAR, which leverages the random-walks-based network embedding method to learn the node representation for clustering. According to the experimental results, NEGAR surpasses the chosen state-of-the-art algorithms in terms of multiple metrics. In addition, NEGAR can scale to super large-sized software and is faster than most other algorithms.

A few points deserve further study. On the one hand, the presence of redundant dependencies in the graph can interfere with the architecture recovery, but no research currently concentrates on evaluating these negative influences. On the other hand, non-existing algorithms can use both structural and non-structural information. Although structural information has shown higher value, non-structural information, such as file name, comments, code semantics, etc., can also contribute to architecture recovery. Thus, we plan to carry out the following two research directions in the future.

1) *Optimizing the dependency graph structure:* We aim to optimize the redundant dependency in the graph and evaluate the impact on the architecture recovery.
2) *Integrate structural and non-structural information:* We plan to utilize graph neural networks (GNN) to integrate structural and non-structural information (e.g., using

code semantics as initial node attributes in the graphs) for architecture recovery.

REFERENCES

[1] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, "Documenting software architectures: Views and beyond," in *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, IEEE, 2003, pp. 740–741.

[2] D. Garlan, "Software architecture: A roadmap," in *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, ACM Press, 2000, 91–101.

[3] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, IEEE, 2013, pp. 901–910.

[4] T. Lutellier, D. Chollak, J. Garcia, *et al.*, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 159–181, 2017.

[5] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*, IEEE, 2011, pp. 35–44.

[6] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, IEEE Computer Society, 2011, pp. 552–555.

[7] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 150–165, 2005.

[8] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, IEEE, 2013, pp. 486–496.

[9] V. Tzerpos and R. C. Holt, "Accd: An algorithm for comprehension-driven clustering," in *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, IEEE, 2000, pp. 258–267.

[10] N. Teymourian, H. Izadkhah, and A. Isazadeh, "A fast clustering algorithm for modularization of large-scale software systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1451–1462, 2022.

[11] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Proceedings of the 6th IEEE International Conference on Software Maintenance (ICSM'99)*, IEEE, 1999, pp. 50–59.

[12] W. H. L, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *ArXiv*, vol. abs/1709.05584, 2017.

[13] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'14)*, ACM, 2014, pp. 701–710.

[14] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*, ACM, 2016, pp. 855–864.

[15] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: A survey," *Knowledge-Based Systems*, vol. 151, pp. 78–94, 2018.

[16] N. S. Jalali, H. Izadkhah, and S. Lotfi, "Multi-objective search-based software modularization: Structural and non-structural features," *Soft Computing*, vol. 23, no. 21, pp. 11 141–11 165, 2019.

[17] C. Lindig and G. Snelting, "Assessing modular structure of legacy code based on mathematical concept analysis," in *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, IEEE, 1997, 349–359.

[18] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, 2007.

[19] J. K. Chhabra *et al.*, "Harmony search based remodularization for object-oriented software systems," *Computer Languages, Systems & Structures*, vol. 47, pp. 153–169, 2017.

[20] J. Huang and J. Liu, "A similarity-based modularization quality measure for software module clustering problems," *Information Sciences*, vol. 342, pp. 96–110, 2016.

[21] J. Huang, J. Liu, and X. Yao, "A multi-agent evolutionary algorithm for software module clustering problems," *Soft Computing*, vol. 21, no. 12, pp. 3415–3428, 2017.

[22] A. Isazadeh, H. Izadkhah, and I. Elgedawy, *Source code modularization: theory and techniques*. Springer Cham, 2017.

[23] H. Izadkhah, I. Elgedawy, and A. Isazadeh, "E-cdgm: An evolutionary call-dependency graph modularization approach for software systems," *Cybernetics and Information Technologies*, vol. 16, no. 3, pp. 70–90, 2016.

[24] K. Jeet and R. Dhir, "Software clustering using hybrid multi-objective black hole algorithm," in *Proceedings of the 28th International Conference on Software Engineering and Knowledge Engineering (SEKE'16)*, JVLC, 2016, pp. 650–653.

[25] A. C. Kumari and K. Srinivas, "Hyper-heuristic approach for multi-objective software module clustering," *Journal of Systems and Software*, vol. 117, pp. 384–401, 2016.

[26] B. S. Mitchell and S. Mancoridis, "On the evaluation of the bunch search-based software modularization algorithm," *Soft Computing*, vol. 12, no. 1, pp. 77–93, 2008.

[27] M. C. Monçores, A. C. Alvim, and M. O. Barros, "Large neighborhood search applied to the software module clustering problem," *Computers & Operations Research*, vol. 91, pp. 92–111, 2018.

[28] M. Tajgardan, H. Izadkhah, and S. Lotfi, "Software systems clustering using estimation of distribution approach," *Journal of Applied Computer Science Methods*, vol. 8, no. 2, pp. 99–113, 2016.

[29] A. Corazza, S. Di Martino, and G. Scanniello, "A probabilistic based approach towards software system clustering," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR'10)*, IEEE, 2010, pp. 88–96.

[30] R. Naseem, O. Maqbool, and S. Muhammad, "Cooperative clustering for software modularization," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2045–2062, 2013.

[31] A. Rathee and J. K. Chhabra, "Software remodularization by estimating structural and conceptual relations among classes and using hierarchical clustering," in *Proceedings of the 1st International Conference on Advanced Informatics for Computing Research (ICAICR'17)*, vol. 712, Springer Singapore, 2017, pp. 94–106.

[32] M. Saeed, O. Maqbool, H. A. Babri, S. Z. Hassan, and S. M. Sarwar, "Software clustering techniques and the use of combined algorithm," in *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, IEEE, 2003, pp. 301–306.

[33] C. Srinivas, V. Radhakrishna, and C. G. Rao, "Clustering software components for program restructuring and component reuse using hybrid xnor similarity function," *Procedia Technology*, vol. 12, pp. 246–254, 2014.

[34] O. Maqbool and H. A. Babri, "The weighted combined algorithm: A linkage algorithm for software clustering," in *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, IEEE, 2004, pp. 15–24.

[35] P. Andritsos, P. Tsaparas, R. J. Miller, and K. C. Sevcik, "Limbo: Scalable clustering of categorical data," in *Proceedings of the 9th International Conference on Extending Database Technology (EDBT'04)*, Springer-Verlag, 2004, pp. 123–146.

[36] B. S. Mitchell, "A heuristic search approach to solving the software clustering problem," Ph.D. dissertation, 2002.

[37] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, 993–1022, 2003.

[38] P. Wang, J. Yang, L. Tan, R. Kroeger, and J. D. Morgenthaler, "Generating precise dependencies for large software," in *Proceedings of the 4th International Workshop on Managing Technical Debt (MTD'13)*, IEEE, 2013, pp. 47–50.

[39] T. Lutellier, D. Chollak, J. Garcia, *et al.*, "Comparing software architecture recovery techniques using accurate dependencies," in *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15)*, vol. 2, IEEE, 2015, pp. 69–78.

[40] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3-5, pp. 75–174, 2010.

[41] K. Henderson, B. Gallagher, T. Eliassi-Rad, *et al.*, "Rolx: Structural role extraction & mining in large graphs," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'12)*, ACM, 2012, pp. 1231–1239.

[42] D. Guthrie, B. Allison, W. Liu, L. Guthrie, and Y. Wilks, "A closer look at skip-gram modelling," in *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC'06)*, vol. 6, ELRA, 2006, pp. 1222–1225.

[43] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th International Conference on World Wide Web (WWW'15)*, International World Wide Web Conferences Steering Committee, 2015, 1067–1077.