Monolith to Microservices: Representing Application Software through Heterogeneous Graph Neural Network

Alex Mathai[1] , **Sambaran Bandyopadhyay**[2*] , **Utkarsh Desai**[1†] and **Srikanth Tamilselvam**[1]
[1]IBM Research
[2]Amazon
{alexmathai98, samb.bandyo, utk.is.here, srikanthtamilselvam}@gmail.com

Abstract

Monolithic software encapsulates all functional capabilities into a single deployable unit. But managing it becomes harder as the demand for new functionalities grow. Microservice architecture is seen as an alternative as it advocates building an application through a set of loosely coupled small services wherein each service owns a single functional responsibility. But the challenges associated with the separation of functional modules, slows down the migration of a monolithic code into microservices. In this work, we propose a representation learning based solution to tackle this problem. We use a heterogeneous graph to jointly represent software artifacts (like programs and resources) and the different relationships they share (function calls, inheritance, etc.), and perform a constraint-based clustering through a novel heterogeneous graph neural network. Experimental studies show that our approach is effective on monoliths of different types.

1 Introduction

Monolith architecture is the traditional unified model for designing software applications. It encapsulates multiple business functions into a single deployable unit. But such applications become difficult to understand and hard to maintain as they age, as developers find it difficult to predict the change impact [Kuryazov *et al.*, 2020]. Therefore, microservice [Thönes, 2015] architectures are seen as an alternative. It aims to represent the application as a set of small services where each service is responsible for a single functionality. It brings multiple benefits like efficient team structuring, independence in development & deployment, enables flexible scaling and less restriction on technology or programming language preference. But migrating from monolith to microservices is a labour intensive task. It often involves domain experts, microservices architects and monolith developers working in tandem to analyze the application from multiple views and identify the components of monolith applications that can be turned into a cohesive, granular service. They also need to work with constraints like the exact number of microservices to be exposed and components that should definitely be part of a particular microservice.

The software engineering community refers to this migration process as a software decomposition task. Many works [Tzerpos and Holt, 2000; Harman *et al.*, 2002; Mazlami *et al.*, 2017; Mahouachi, 2018; Mancoridis *et al.*, 1999] leverage the syntactical relationships between the programs and treated this decomposition as an optimization problem to improve different quality metrics like cohesion, coupling, number of modules, amount of changes etc. While the accuracy of these approaches has been evolving over time, they have their drawbacks such as 1) reliance on external artifacts like logs, commit history etc. 2) focus on only a subset of the programs 3) less attention to non program artifacts like the tables, files 4) minimal consideration for transactional data. Recently [Jin *et al.*, 2019; Kalia *et al.*, 2020] executed test cases to extract runtime traces. Each execution is considered as a business function and they try to cluster business functions. But this work relies on access to runtime traces and complete coverage of test cases which cannot always be guaranteed. Also, these works did not consider data entities for decomposition.

Graphs are a natural choice to represent the application's structural and behavioral information [Mancoridis *et al.*, 1998; Desai *et al.*, 2021]. The structural information consisting of different application entities such as programs, files, database tables can be represented as nodes and their different relationships such as *calls, extends, implements* between program to program and different CRUD operations that happen from program to data resources (table, file) can be represented as edges in the graph. Figure 1 captures the construction of a heterogeneous graph from a sample java code. The behavioral information of the application identified through the sequence of programs and data resources that come together to support a business function can be captured as node/edge attributes. The monolith to microservices task can thus be viewed as a graph based clustering task which involves 1) Representation learning of application implementation from the graph structure and 2) Using this learnt representation for clustering. Graph neural networks have achieved state of the art results for multiple graph-based downstream tasks such as node classification and graph classification [Kipf and Welling, 2017; Xu *et al.*, 2019; Veličković *et al.*, 2018]. Most graph neural networks follow

*The work was done when Sambaran was affiliated to IBM Research prior to joining Amazon

†Utkarsh is currently at Google

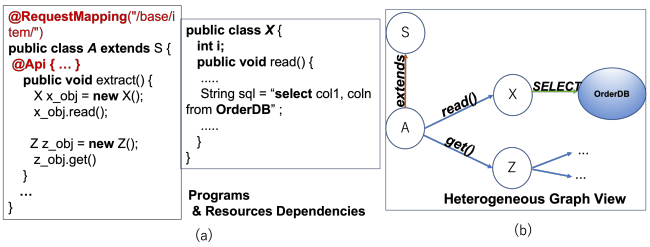


Figure 1: Representing software as a heterogeneous graph: (a) Shows a class *A* that `extends` class *S*. *A* implements the `extract()` method where it invokes class *X* `read()` method. The `extract()` method also has the `@Api` annotation indicating that it exposes a service. We also notice class *X* performing a `READ` operation on table *OrderDB*. (b) Shows the graph view with classes *A*, *S*, *X* as *program* nodes and table *OrderDB* as a *resource* node. Additionally, the different kind of dependencies are represented as edges $e(A, S)$, $e(A, X)$, $e(A, Z)$ and $e(X, OrderDB)$

.

message passing mechanisms where the vector representation of a node is updated by combining its own features and aggregated features from its neighborhood. Recently [Desai *et al.*, 2021] showed how the programs and its relationships in the application can be represented as a graph and proposed a multi-objective graph convolution network that combined node representation & node clustering by diluting outliers. But since the framework did not consider application's data resources like database tables, files and the different relationships that exists between programs & resources in the graph construction, the functional independence property of microservices is not completely satisfied. In addition, application architects have a *functional view* of the application, so they decide on the target number of microservices and identify the core representative programs or tables from the monolith for each microservice. The solution should therefore accept the architect inputs as constraints and form clusters to maximize functional alignment.

In this work, we propose a novel graph neural network based solution to refactor monolith applications into a desired number of microservices. The main contributions of our paper are listed below.

1. We translate the application software's structural and behavioral properties into a heterogeneous graph through nodes, edges and node/edge attributes.
2. We introduce CHGNN - a novel heterogeneous graph neural network (GNN), that enables the representation of both *data resources* and *programs*. For the first time in literature, we perform a constraints-based clustering jointly in the framework of heterogeneous GNN.
3. We show that inclusion of heterogeneous information generates better quality microservices recommendations through four publicly available monolith applications.

2 Methodology

Given a monolith application and constraints S_{list} - the list of K seed sets from the subject matter experts (SMEs), we want to partition the monolith into K fairly distributed clusters where each cluster is a group of programs and resources that perform a well-defined functionality.

2.1 Converting Applications to Graph

We now describe our approach to represent an application as a graph. The primary programming construct in different languages is different - a class in Java and a program in COBOL. Hence, in the rest of this work, we refer to classes or programs as simply programs for consistency. Consider a simple Java application as shown in Figure 1. Each class or program in the application can be represented as a *Program* node in the graph. Certain programs might also access resources such as database tables, files or other data structures. These can be represented as *Resource* nodes in the graph. We denote the combined set of *Program* and *Resource* nodes as V, the set of all nodes. We establish an undirected `CALLS` edge from *Program* node A to *Program* node B if there is a method in the program A that calls a method from program B. We also identify resource usage and create a `CRUD` edge from *Program* node X to *Resource* node R, if the program X accesses resource R. Static analysis tools can analyze the application code and identify the call chains and resource usage. E denotes the combined set of all edges between the various nodes in the graph. Multiple method calls or resource usages between two nodes are still represented by a single unweighted edge.

We now generate the node attribute matrix, corresponding to the *Program* and *Resource* nodes of the graph. APIs exposed by applications are referred as EntryPoint Specifications [Dietrich *et al.*, 2018], or simply, *Entrypoints* (EPs). The methods invoked through these APIs are annotated with tags such as `@Api` as shown in Figure 1. We refer to such methods as entrypoint methods and the corresponding programs as *entrypoint programs*. Each entrypoint program can thus be associated with multiple entrypoints due to different entrypoint methods. From an entrypoint method, we can obtain a sequence of invoked methods and their corresponding programs using the execution traces of that Entrypoint. If EP is the set of Entrypoints in an application and V_P, is the set of *Program* nodes, we can define a matrix $A^{|V_P|\times|EP|}$, such that $A(i, p) = 1$ if program i is present in the execution trace of entrypoint p, else 0. Additionally, we define another matrix $C^{|V_P|\times|V_P|}$ such that $C(i, j)$ is the number of Entrypoint execution traces that contain both programs i and j. If a program is not invoked in an execution trace for any Entrypoint, we remove the corresponding non-reachable *Program* node from the graph. Finally, classes or programs may also inherit from other classes or programs or implement Interfaces. In Figure 1, class *A* inherits from class *S*. Although this establishes a dependency between the programs, it is not a direct method invocation. Hence, this dependency is not included as an edge in the graph, but as a *Program* node attribute. Therefore, we define a matrix $I^{|V_P|\times|V_P|}$ and set $I(i, j) = I(j, i) = 1$ if programs i and j are related via an inheritance relationship and 0 otherwise. The attribute matrix for *Program* nodes is the concatenation of A, C and I, and denoted as X_P.

For *Resource* nodes, the Inheritance features are not applicable. The A and P matrices are obtained by summing

up the corresponding rows from the respective X_P matrices. The relationship between *Program* nodes and *Resource* nodes is many-to-many and this formulation simply aggregates features from all related programs into the resource to form the resource attribute matrix X_R. Each constituent matrix of X_P and X_R is row-normalized individually. The final set of node attributes are denoted as $X_V = \{X_P, X_R\}$. The Edge attributes for `CALLS` edges is simply the vector $[1, 0]$ and there are no additional features. For `CRUD` edges, the attribute vector represents the type of resource access performed. Since a program can access a resource in more than one fashion, this is a 1×4 vector, where each attribute represents the associated access type - [Create, Read, Update, Delete]. Hence a program that reads and updates a *Resource* node will have $[0, 1, 1, 0]$ as the edge feature. The edge attribute matrix is represented as X_E.

Thus, an application can be represented by a heterogeneous graph as $G = (V, E, X_V, X_E)$. Let us assume that $\phi(v)$ and $\psi(e)$ denote the node-type of v and edge-type of e respectively. Let us use $\mathbf{x}_v \in \mathbb{R}^{D_{\phi(v)}}$ to denote the attribute vector for the node v which belongs to $D_{\phi(v)}$ dimensional space. Similarly, $\mathbf{x}_e \in \mathbb{R}^{D_{\psi(e)}}$ is the edge attribute of the edge e.

2.2 Proposed Heterogeneous Graph Neural Network

In this subsection, we aim to propose a graph neural network (GNN) which can (i) handle different node and edge types in the graph, (ii) obtain vector representation of both nodes and edges by jointly capturing both the structure and attribute information, (iii) output community membership of all the nodes in the graph in a unified framework. We refer the proposed architecture as *CHGNN* (Community aware Heterogeneous Graph Neural Network). There are different steps in the design of CHGNN as described below.

Mapping Entities to a Common Vector Space

Due to heterogeneity from different software artifacts, attributes associated with nodes and edges of the input graph are not of same types and they can have different dimensions. Such heterogeneity can be addressed in the framework of message passing heterogeneous graph neural networks in two ways: (i) Map the initial attributes to a common vector space using trainable parameter matrices at the beginning [Wang *et al.*, 2019]; (ii) Use different dimensional parameter matrices while aggregating and combine information at each step of message passing [Vashishth *et al.*, 2020]. We choose the first strategy since that makes the subsequent design of the GNN simpler and helps to add more layers in the GNN. So, we introduce type specific trainable matrices $W_{\phi(v)} \in \mathbb{R}^{F^{(0)} \times D_{\phi(v)}}$ for nodes and $W_{\psi(e)} \in \mathbb{R}^{F^{(0)} \times D_{\psi(e)}}$ for edges $\forall v \in V$ and $\forall e \in E$.

$$\mathbf{h}_v^{(0)} = \sigma(W_{\phi(v)}\mathbf{x}_v);\ \mathbf{h}_e^{(0)} = \sigma(W_{\psi(e)}\mathbf{x}_e) \tag{1}$$

where σ is a nonlinear activation function. $\mathbf{x}_v$ and $\mathbf{x}_e$ are the initial attribute vectors of node v and edge e respectively. $\mathbf{h}_v^{(0)}$ and $\mathbf{h}_e^{(0)}$ are considered as 0th layer embeddings for v and e respectively. They are fed to the message passing framework as discussed below.

Message Passing Layers for Nodes and Edges

Message passing graph neural networks have achieved significant success for multiple node and graph level downstream tasks. In this framework, we obtain vector representation for both nodes and edges of the heterogeneous graph. There are $L \geq 1$ message passing layers. We define the lth layer $(1 \leq l \leq L)$ of this network as follows.

As the first step of a message passing layer, features from the neighborhood are aggregated for each node. In recent literature, it has been shown that obtaining both node and edge representation improves the downstream performance for multiple applications [Jiang *et al.*, 2019; Bandyopadhyay *et al.*, 2019]. Following that, we also design the GNN to exchange features between nodes and edges, and update the vector representation for both. In each layer, we use two parameter matrices $W_1^{(l)} \in \mathbb{R}^{F^{(l)} \times F^{(l-1)}}$ and $W_2^{(l)} \in \mathbb{R}^{F^{(l)} \times F^{(l-1)}}$ to handle node and edge embeddings respectively. For a node $v \in V$, its neighborhood information is aggregated as:

$$\mathbf{z}_v^{(l)} = \sum_{u \in \mathcal{N}(v)} \frac{1}{\sqrt{d_u}\sqrt{d_v}} W_1^{(l)} \mathbf{h}_u^{(l-1)} * \sigma\big(W_2^{(l)} \mathbf{h}_{uv}^{(l-1)}\big) \tag{2}$$

where d_u and d_v are the degrees of the nodes u and v respectively. $\frac{1}{\sqrt{d_u}\sqrt{d_v}}$ is used to symmetrically normalize the degrees of the nodes in the graph [Kipf and Welling, 2017]. $\sigma()$ is a nonlinear activation function and $*$ is Hadamard (element-wise) product. Next, the aggregated information $\mathbf{z}_v^{(l)}$ is combined with the embedding of node v to update it as follows.

$$\mathbf{h}_v^{(l)} = \sigma\big(\mathbf{z}_v^{(l)} + \frac{1}{d_v}\mathbf{h}_v^{(l-1)}\big) \tag{3}$$

$\mathbf{h}_v^{(l)}$ is considered as the node embedding of node v at lth layer. To update the embedding of an edge (u, v), we use the updated embeddings of two end point nodes and the existing embedding of the edge as follows ($||$: concatenation of vectors):

$$\mathbf{h}_{uv}^{(l)} = \sigma\Big(W_3^{(l)}\big(\frac{\mathbf{h}_u^{(l)} + \mathbf{h}_v^{(l)}}{2} \,||\, \mathbf{h}_{uv}^{(l-1)}\big)\Big) \tag{4}$$

where $W_3^{(l)} \in \mathbb{R}^{F^{(l)} \times (F^{(l)} + F^{(l-1)})}$ is a parameter matrix. This completes the definition of the layer l of the message passing network. Please note that the dimensions of the trainable matrices $W_1^{(l)}$, $W_2^{(l)}$ and $W_3^{(l)}$ determine the dimension of the embedding space of the nodes and edges.

To build the complete network, we first map the heterogeneous nodes and edges to a common space using Equation 1. Subsequently, we use 2 message passing layers ($l = 1, 2$) as encoders (compressing the feature space) and next 2 message passing layers ($l = 3, 4$; $L = 4$) as decoders (decompressing the feature space), with $F^{(0)} > F^{(1)} > F^{(2)} < F^{(3)} = F^{(1)} < F^{(4)} = F^{(0)}$. To map the node and edge features to their respective input attribute space $\mathbb{R}^{D_{\phi}(v)}$ and $\mathbb{R}^{D_{\psi}(e)}$, we again use linear transformations followed by activation functions as shown below.

$$\hat{\mathbf{x}}_v = \sigma(\hat{W}_{\phi(v)}\mathbf{h}_v^{(L)})\ ;\ \ \hat{\mathbf{x}}_e = \sigma(\hat{W}_{\psi(e)}\mathbf{h}_e^{(L)}) \tag{5}$$

These reconstructed node and edge attributes are used to design the loss functions as discussed next.

Design of the Loss Functions and Joint Clustering of Heterogeneous Nodes

We use three types of unsupervised reconstruction losses.

Node Attribute Reconstruction: We try to bring the initial node features $\mathbf{x}_v$ and the reconstructed node features $\hat{\mathbf{x}}_v$ close to each other by minimizing $\sum_{v \in V} ||\mathbf{x}_v - \hat{\mathbf{x}}_v||_2^2$.

Edge Attribute Reconstruction: With similar motivation as above, we minimize $\sum_{e \in E} ||\mathbf{x}_e - \hat{\mathbf{x}}_e||_2^2$.

Link Reconstruction: Above two loss components do not capture anything about the link structure of the heterogeneous graph. Let us introduce the binary variables $a_{u,v}\ \forall u, v \in V$ such that $a_{u,v} = 1$ if $(u, v) \in E$ and $a_{u,v} = 0$ otherwise. We want to ensure that embeddings of two nodes are close to each other if there is an edge between them by minimizing $\sum_{u,v \in V} \left(a_{u,v} - \mathbf{x}_u \cdot \mathbf{x}_v\right)_2^2$.

Unifying Node Clustering: After we map the monolith to a heterogeneous graph, we cluster the nodes of the graph to form microservices. As the nodes are represented in the form of vectors through the heterogeneous GNN encoder as discussed in Section 2.2, we unify the clustering objective with the heterogeneous GNN as follows.

The node embeddings at the end of encoding layers (i.e., $L/2$ layers) are $\mathbf{h}_v^{(L/2)}$, $\forall v \in V$. We design a k-means++ objective [Arthur and Vassilvitskii, 2006] by introducing two parameter matrices $M \in \{0, 1\}^{|V| \times K}$ and $C \in \mathbb{R}^{K \times F^{(L/2)}}$. M is the binary cluster assignment matrix where each row sums up to 1. We assume to know the number of clusters K. $M_{vk} = 1$ if node v belongs to kth cluster and $M_{vk} = 0$ otherwise[1]. kth row of C, denoted as C_k, is the center of kth cluster in the embedding space. Node clusters and the corresponding cluster centers can be obtained by minimizing clustering loss (CL) which is $\sum_{v \in V} \sum_{k=1}^{K} M_{vk} ||\mathbf{h}_v^{(L/2)} - C_k||_2^2$.

Allowing seed constraints : As motivated in Section 1, real-world applications generally have constraints provided by SMEs in the form of clustering seeds. To incorporate such constraints, we take a list of seed sets $S_{list} = [S_1, \cdots, S_K]$ as input, where each S_i is a set of seed nodes that must belong to the corresponding cluster $\mathcal{C}_i$.

$$\mathbf{SC}_k = \sum_{i \in S_k} \mathbf{h}_i^{(L/2)} / |S_k|, \forall k \in [1, \cdots, K] \tag{6}$$

$$Dist = -\sum_{k=1}^{K} \sum_{k'=1}^{K} ||\mathbf{SC}_k - \mathbf{SC}_{k'}||_2^2 \tag{7}$$

With S_{list} as input, we need to ensure two requirements - 1) In the final output, every seed set (S_i) must belong to a predetermined cluster ($\mathcal{C}_i$) and 2) the embeddings of the seed sets in different clusters must be as far apart as possible. To address the first requirement we add hard constraints to our cluster assignment algorithm . This can be seen in Algorithm 1 (Lines 15-17), where for each seed artifact, we assign the cluster manually. To address the second requirement we add soft constraints to our clustering loss function. This is captured by measuring the distance between the seed set centers ($\mathbf{SC}_k$) as shown in Equation 7. As the distance should be maximised, we negate this distance to maintain a minimization objective.

[1]To avoid cluttering of notations, we use M_{vk} instead of $M_{\text{index}(v)k}$, where $1 \leq \text{index}(v) \leq |V|$

Algorithm 1 CHGNN

Input: Class dependencies and S_{list} (list of K seed sets)
1: Convert the application to a graph representation as defined in Section 2.1 and obtain the V, E and X_V and X_E.
2: Let S_{list} be $[S_1, \cdots, S_K]$. Hence each cluster ($\mathcal{C}_k$) has a mandatory set of seeds (S_k).
3: Let the list of clusters be $[\mathcal{C}_1, \cdots, \mathcal{C}_K]$ and the list of centers for each cluster be $[C_1, \cdots, C_K]$.
4: Pre-train the heterogeneous GNN encoder and decoder (refer **Training Procedure** in our extended paper [2])
5: % Initialize the cluster centers as follows %
6: $C_k = \sum_{k \in S_k} \mathbf{h}_k^{L/2} / |S_k|, \forall k \in [1, \cdots, K]$
7: **for** T iterations **do**
8: % Assign a cluster to each node %
9: **for** $v \in V$ **do**
10: **if** v not in S_{list} **then**
11: % Find closest centres for non-seeds %
12: Find the closest center (C_k) with Equation 9
13: Add v to the corresponding cluster ($\mathcal{C}_k$)
14: **else**
15: % Fix the cluster for seeds %
16: Find S_k such that $v \in S_k$
17: Add v to the corresponding cluster ($\mathcal{C}_k$)
18: **end if**
19: **end for**
20: Update cluster centers with Equation 10.
21: Update the parameters of the heterogeneous GNN encoder and decoder by minimizing Eq. 8 using ADAM.
22: **end for**
Output: Clusters and Cluster Centers

Hence, the total loss to be minimized by CHGNN is:

$$\min_{\mathcal{W}, M, C} \mathcal{L} = \alpha_1 \sum_{v \in V} ||\mathbf{x}_v - \hat{\mathbf{x}}_v||_2^2 + \alpha_2 \sum_{e \in E} ||\mathbf{x}_e - \hat{\mathbf{x}}_e||_2^2 + \alpha_3 \sum_{u,v \in V} \left(a_{u,v} - \mathbf{x}_u \cdot \mathbf{x}_v\right)_2^2 + \alpha_4 \left(CL + Dist\right) \tag{8}$$

where $\mathcal{W}$ contains the trainable parameters of the GNN described in Section 2.2. α_1, α_2, α_3 and α_4 are non-negative weights. In our algorithm, we set them such that the sum of these non-negative weights always sum up to one.

2.3 Training and Analysis

First, we pre-train the parameters of the GNN without including the clustering loss component, i.e., setting $\alpha_4 = 0$ in Equation 8. We use ADAM optimization technique to update the parameters of the GNN. Once the pre-training is completed, we use alternating optimization techniques to update each of clustering parameters M and C, and parameters of GNN $\mathcal{W}$, while keeping others fixed. Using Lloyd's update

rule for k-means, we update M and C as:

$$M(v,k) = \begin{cases} 1, \text{ if } k = \underset{k' \in \{1,\cdots,K\}}{\text{argmin}} ||\mathbf{h}_v^{(L/2)} - C_{k'}||_2^2 \\ 0, \text{ Otherwise} \end{cases} \tag{9}$$

$$C_k = \frac{1}{N_k} \sum_{v \in \mathcal{C}_k} \mathbf{h}_v^{(L/2)} \tag{10}$$

where $N_k = \sum_{v \in V} M_{vk}$. Due to the presence of clustering loss component in Equation 8, updating the parameters $\mathcal{W}$ of GNN can pull the node embeddings close to their respective cluster centers further, along with reconstructing initial node and edge attributes and the link structure.

3 Experimental Evaluation

To study the efficacy of our approach, we chose four publicly-available monoliths namely Daytrader, PlantsbyWebsphere (PBW), Acme-Air and GenApp. Together, these applications show a good diversity in terms of the programming paradigms, languages and technologies used. Details of each monolith are provided in Table 1. We did not include DietApp (used in baseline [Desai *et al.*, 2021]), as the public repository does not expose the code used for interfacing between programs and tables. Hence DietApp reduces to a homogeneous graph (only programs). As expected, upon experimentation, we observe that CHGNN gets the same output as the baseline.

3.1 Constraints to Clustering

For each application, we get the target number of microservices (*K clusters*) and constraints of what entities each microservice (*K seed sets*) should contain as inputs from the SMEs. Naturally, the seed sets cannot be shared/overlap with each other. Typically, the constraints include tables and program entities that are central to the cluster. They act as a means to guarantee functional alignment.

3.2 Quantitative Metrics

To quantitatively evaluate the clusters, we use four established graph metrics. We briefly touch upon these below.

1. Modularity (Mod) : The Modularity metric [Newman and Girvan, 2004] is widely used to evaluate the quality of generated graph partitions. It computes the difference between the *actual* intra-edges of a cluster and the *expected* intra-edges of the cluster in a randomly re-wired graph. Higher the modularity - the better the partitions.

2. Non-Extreme Distribution (NED) : The NED metric [Wu *et al.*, 2005] examines if the graph partitions are tiny, large or of an acceptable size. Rather than fixing the low and high limits to 5 and 20 [Wu *et al.*, 2005], we take the average cluster size as input and check if the cluster size lies within the 50% tolerance bandwidth of the average. This change helps NED generalise well to small and large sized applications.

$$NED = \frac{1}{N} \sum_{k=1}^{K} n_k, \forall n_k \in [(1-\epsilon) * \frac{N}{K}, (1+\epsilon) * \frac{N}{K}] \tag{11}$$

where N and K are the number of graph nodes and the number of clusters respectively. Here, we set ϵ to 0.5.

3. Coverage : Coverage [Fortunato, 2010] tries to measure *cohesion* by computing the ratio of the number of intra-cluster edges to the total number of edges in the graph. Higher coverage implies higher cohesion which results in better clusters.

4. S-Mod : S-Mod [Jin *et al.*, 2019] is another quantitative approach that measures the quality of generated partitions. It is computed by subtracting *coupling* from *cohesion*. More cohesion implies more intra-cluster edges and more coupling implies more inter-cluster edges. For microservices, it is ideal to have high cohesion and low coupling. Hence, higher S-Mod values are favourable.

3.3 Combined Metric

However in our experiments, we observe that a few *boulder* (very large) clusters and many *dust* (very small) clusters can increase intra-cluster edges and decrease inter-cluster edges. This improves S-Mod and Coverage considerably, but leads to a poor NED score - which penalises for size imbalance. Hence, instead of relying on a single metric, we calculate $Metric_{Sum}$ (summing all metrics) and rank our approaches accordingly. $Metric_{Sum} \in [-2, 4]$ as Mod $\in [-1, 1]$, NED $\in [0, 1]$, S-Mod $\in [-1, 1]$ and Coverage $\in [0, 1]$. In all our experiments, we observe that every metric is positive for all the algorithms considered.

3.4 Baseline Algorithms and Experimental Setup

As converting monoliths to heterogeneous GNNs and applying them for clustering is a novel direction, we have designed most of the baselines with motivations from existing works on graph representation learning. They are discussed below.

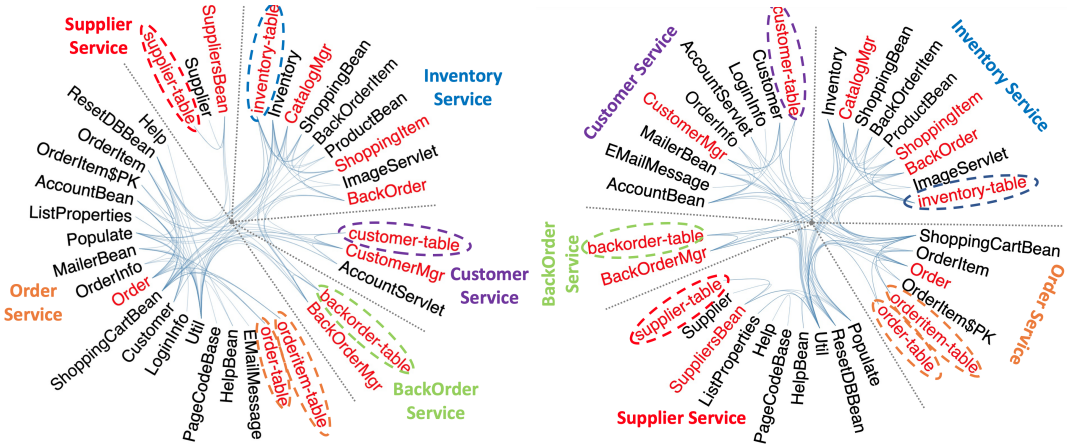
COGCN++ : [Desai *et al.*, 2021] introduced COGCN, wherein, the monolith application (having programs and resources) is converted to a homogeneous graph. Subsequently, COGCN, which has GCN layers trained on reconstruction and clustering loss, is applied on the homogeneous graph to obtain the micro-services. When running vanilla COGCN, we noticed that it does not ensure the mutual exclusivity of seed sets. Hence we add the seed constraint loss (Equation 7) to fulfill this requirement. We refer to this model as COGCN++. Note : COGCN++ does not consider edge embeddings/attributes - it only considers node embeddings.

HetGCNConv: Here, we create the heterogeneous graph as discussed in Section 2.1, but consider all the edges as of similar types. We map the heterogeneous nodes to a common vector space as done in Section 2.2 by using node-type specific parameter matrices $W_{\phi(v)}$'s. We then use the GCN convolution model and train it by setting α_2 to 0.

CHGNN-EL : A variant of our proposed CHGNN where we drop the edge feature re-construction loss from the optimization procedure by setting $\alpha_2 = 0$ in Equation 8.

CHGNN : This is our final model proposed in this work.

Please note that COGCN++ is a homogeneous graph based approach. HetGCNConv and CHGNN-EL are two variants of our proposed heterogeneous model CHGNN. HetGCNConv only uses heterogeneous nodes but homogeneous edges. CHGNN-EL uses both heterogeneous nodes and edges, but sets α_2 to 0.



(a) Results for COGCN++ (b) Results for CHGNN

Figure 2: Candidate microservices for PBW application : The clusters capture the five business functions of PBW. The bounded circles indicate the different *resources* (*tables*) owned by the clusters.The nodes in *red* fonts are seeds provided by the SME.

3.5 Qualitative and Quantitative Results

For our qualitative analysis, we study our predictions for the PBW application and compare our results with COGCN++. As seen in Figure. 2b, we observe that CHGNN using the seed inputs (S_{list}) has identified five functional clusters - Customer, Order, Inventory, BackOrder and Supplier. The programs in each cluster have very close dependencies within the cluster and contribute majorly to a common business function. This is also evident from the similar names of artifacts in a cluster (like *OrderItem*, *Order-table*, *Orderitem-table* in the *Order* service). Associated with each cluster is a group of data nodes (*dashed circled*) that interact closely with the programs in their cluster. As seen, the *Supplier* cluster has *supplier-table*, *Supplier*, *SupplierBean* and *Populate* nodes. In Figure 2a, we observe that COGCN++ is successful at separating seed sets, as it leverages the seed constraint loss. However, we notice the following differences.

1. Unlike CHGNN that creates evenly sized clusters, COGCN++ creates a few *boulder* clusters like the *Order* cluster (having 19 artifacts) and many *dust* clusters like *Supplier*, *BackOrder* and *Customer* (having atmost 3 artifacts). As explained in Section 3.3, this strategy helps COGCN++ outperform on S-Mod and Coverage but dramatically underperform on NED. For the task of microservices partitioning, having skewed cluster sizes may result into few over-utilized services and many under-utilized services - which is not desired.

2. We also find that COGCN++ wrongly associates many artifacts that actually belong to the *Customer* service (like *AccountBean*, *LoginInfo* and *EmailMessage*) with the *Order* and *Inventory* service. This is not the case for CHGNN.

All of the above observations have been validated by one of our SMEs (R1) - "CHGNN is better. Reason : The clusters are more evenly distributed. Customer service came out well with account management (AccountBean, LoginInfo, EmailMessage) contained in it which is desired."

We depict our quantitative results in terms of $Metric_{Sum}$ in Table 1 and explain two trends that we have observed.

(i) COGCN++ scores lower values for $Metric_{Sum}$ consistently in every application. Hence, on an average, the heterogeneous graph formulations outperform COGCN++.

Dataset	Details	COGCN++	HET-GNN Variations		
			HetGCNConv	CHGNN-EL	CHGNN
ACME (Airline App, Lang: Java)	K=4 #Class=30 #Resource=6	1.329	1.712	1.775	**1.784**
DayTrader (Trading App, Lang:Java)	K=6 #Class=111 #Resource=11	1.156	1.096	1.310	**1.336**
PBW (Plant Store, Lang:Java)	K=5 #Class=30 #Resource=6	1.583	1.577	**1.805**	1.762
Genapp (Insurance App, Lang:Cobol)	K=4 #Class=30 #Resource=10	1.870	**2.016**	2.010	2.000

Table 1: Performance of partitioning Monoliths (average of 30 runs)

(ii) In all the applications, either CHGNN or CHGNN-EL appear in the top two results for $Metric_{Sum}$. Hence, on an average, both models are relatively consistent across the varying application topologies.

Summary: From this, it is evident that (1) Heterogeneous graph formulations always guarantee better performance as can be seen in the three HET-GNN variations in Table 1. (2) Clusters in CHGNN are evenly distributed and more meaningful in nature when compared to COGCN++.

For extensive qualitative and metric-specific quantitative analysis of each application, architecture specifications and hardware requirements, please refer to our extended paper [2] .

4 Conclusion

We proposed a novel heterogeneous GNN that enables representation of application data resources and programs jointly for recommending microservices. Both quantitative and qualitative studies show the effectiveness of heterogeneous graph formulations. In the future, we aim to study the decomposition task at a more granular level from *programs* to *functions* and *tables* to *columns*.

[2] https://arxiv.org/abs/2112.01317

References

[Arthur and Vassilvitskii, 2006] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.

[Bandyopadhyay *et al.*, 2019] Sambaran Bandyopadhyay, Anirban Biswas, M Narasimha Murty, and Ramasuri Narayanam. Beyond node embedding: a direct unsupervised edge representation framework for homogeneous networks. *arXiv preprint arXiv:1912.05140*, 2019.

[Desai *et al.*, 2021] Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth Tamilselvam. Graph neural network to dilute outliers for refactoring monolith application. In *Proceedings of 35th AAAI Conference on Artificial Intelligence (AAAI'21)*, 2021.

[Dietrich *et al.*, 2018] Jens Dietrich, François Gauthier, and Padmanabhan Krishnan. Driver generation for java ee web applications. In *2018 25th Australasian Software Engineering Conference (ASWEC)*, pages 121–125. IEEE, 2018.

[Fortunato, 2010] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, Feb 2010.

[Harman *et al.*, 2002] Mark Harman, Robert M Hierons, and Mark Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO*, volume 2, pages 1351–1358, 2002.

[Jiang *et al.*, 2019] Xiaodong Jiang, Pengsheng Ji, and Sheng Li. Censnet: Convolution with edge-node switching in graph neural networks. In *IJCAI*, pages 2656–2662, 2019.

[Jin *et al.*, 2019] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, 2019.

[Kalia *et al.*, 2020] Anup K Kalia, Jin Xiao, Chen Lin, Saurabh Sinha, John Rofrano, Maja Vukovic, and Debasish Banerjee. Mono2micro: an ai-based toolchain for evolving monolithic enterprise applications to a microservice architecture. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1606–1610, 2020.

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

[Kuryazov *et al.*, 2020] Dilshodbek Kuryazov, Dilshod Jabborov, and Bekmurod Khujamuratov. Towards decomposing monolithic applications into microservices. In *2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT)*, pages 1–4. IEEE, 2020.

[Mahouachi, 2018] Rim Mahouachi. Search-based cost-effective software remodularization. *Journal of Computer Science and Technology*, 33(6):1320–1336, 2018.

[Mancoridis *et al.*, 1998] Spiros Mancoridis, Brian S Mitchell, Chris Rorres, Y Chen, and Emden R Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*, pages 45–52. IEEE, 1998.

[Mancoridis *et al.*, 1999] Spiros Mancoridis, Brian S Mitchell, Yihfarn Chen, and Emden R Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 50–59. IEEE, 1999.

[Mazlami *et al.*, 2017] G. Mazlami, J. Cito, and P. Leitner. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531, 2017.

[Newman and Girvan, 2004] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[Thönes, 2015] Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.

[Tzerpos and Holt, 2000] Vassilios Tzerpos and Richard C Holt. Accd: an algorithm for comprehension-driven clustering. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 258–267. IEEE, 2000.

[Vashishth *et al.*, 2020] Shikhar Vashishth, Soumya Sanyal, Vikram Nitin, and Partha Talukdar. Composition-based multi-relational graph convolutional networks. In *International Conference on Learning Representations*, 2020.

[Veličković *et al.*, 2018] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[Wang *et al.*, 2019] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. Heterogeneous graph attention network. In *The World Wide Web Conference*, pages 2022–2032, 2019.

[Wu *et al.*, 2005] Jingwei Wu, Ahmed E Hassan, and Richard C Holt. Comparison of clustering algorithms in the context of software evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 525–535. IEEE, 2005.

[Xu *et al.*, 2019] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.