Expert Systems With Applications 231 (2023) 120597

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

A community detection approach based on network representation learning for repository mining

Marco De Luca, Anna Rita Fasolino, Antonino Ferraro, Vincenzo Moscato, Giancarlo Sperlí *, Porfirio Tramontana

Department of Electrical Engineering and Information Technology (DIETI), University of Naples Federico II, Via Claudio 21, 80125, Naples (NA), Italy

ARTICLE INFO

Keywords:
Developer social network
Social network analysis
Graph-embedding
Community detection
Repository mining

ABSTRACT

In this paper, we propose a novel heterogeneous graph-based model for capturing and handling all the complex and strongly-correlated information of a software *Developer Social Network* (DSN) to support several analytic tasks. In particular, we challenge the problem of automatically discovering *communities* of software developers sharing interests for similar projects by relying on *Social Network Analysis* (SNA) findings. To overcome the huge graph-size issue, we leverage different graph embedding techniques. Eventually, we evaluate the proposed approach with respect to state-of-the-art approaches from an efficiency and an effectiveness point of view by carrying out an experiment involving the GitHub dataset.

1. Introduction

In the last two decades, the git control version system has been adopted by millions of software developers for efficiently managing their projects and easily disseminating their work. GitHub[1] is one of the most diffused version control systems that further provides online open-access to git repositories. It has emerged as a leading choice for developers and researchers seeking to collaborate and share projects using git. GitHub provides query mechanisms and APIs for browsing, searching, and extracting relevant information from its repositories. Based on such mechanisms, several techniques and approaches have been proposed for efficiently and effectively mining data from git repositories, to satisfy different information needs and support several advanced analytics.

In this context, *Developer Social Networks* (DSNs) have recently emerged as an effective tool for the analysis of community structures and collaborations among developers in software projects and software ecosystems (Herbold et al., 2021). A DSN can be considered a Social Network in all respects and all the potentialities, and *Social Network Analysis (SNA)* facilities can be used to infer useful knowledge from these environments for different aims. The classical SNA techniques can be easily adopted on these information networks – usually modeled as graphs – to support a plethora of interesting tasks (leveraging the related topological properties, or based on statistic or bio-inspired approaches): *community detection* to find the community of developers that share interests in similar projects, *influence analysis* to discover the most important developers within a given community, *team formation* to detect the group of developers w.r.t. a given technology within several communities in order to start more quickly a new project in according to agile paradigm, *recommendation* to automatically suggest the most appropriate developers to solve an issue, etc. Thus, community detection in DSNs represents a very challenging issue and the first preliminary step in order to realize the other SNA tasks.

At the best of our knowledge, most approaches proposed in the literature for community detection in DSNs employ *homogeneous* graphs that are characterized by single relevant entities as nodes (i.e., developers) and by a single type of edge and they are often used to model social networks. One example is represented by the homogeneous graph exploited in Hou et al. (2021) that includes a single type of node (representing the developers) and a single type of edge (representing the *commit* relationship). In turn, nodes of an *heterogeneous graph* can have different types (i.e., developers, repositories, issues) and be connected by different types of edges. Due to the multitude of data types represented, they turn out to have a more complex topological structure, which makes them both more information-rich, but also computationally more onerous. However, according to the most recent literature (Ji et al., 2021; Wang et al., 2022a; Yang et al., 2020) *heterogeneous graphs* seem to be the best candidates for capturing and managing the intrinsic complexity and wide range of relations that can be established among modern information networks such as DSNs.

* Corresponding author.
E-mail addresses: marco.deluca2@unina.it (M. De Luca), annarita.fasolino@unina.it (A.R. Fasolino), antonino.ferraro@unina.it (A. Ferraro), vincenzo.moscato@unina.it (V. Moscato), giancarlo.sperli@unina.it (G. Sperlí), porfirio.tramontana@unina.it (P. Tramontana).

[1] https://github.com

https://doi.org/10.1016/j.eswa.2023.120597
Received 17 September 2022; Received in revised form 11 March 2023; Accepted 27 May 2023
Available online 8 June 2023

Unfortunately, DSN-related graphs can reach enormous sizes and thus *graph-embedding* techniques may support in a more efficient manner SNA tasks (Wang et al., 2022b; Xie et al., 2021). In particular, thanks to graph embedding, community detection problems can be effectively faced by leveraging classical clustering algorithms.

In this paper, we decided to explore the possibility of modeling the interactions that developers have within a project repository by a heterogeneous graph and to use graph embedding and community detection techniques on the embedded graph to find relevant types of communities in the global software engineering world. To this aim, we defined a novel DSN model that extracts relevant entities from the GitHub website and relevant relationships that occur among these entities. Therefore, we designed a framework that supports the population of the DSN and implements the analysis tasks by exploiting graph embedding techniques. Actually, we are interested in *non-overlapped* communities because we are focused on identifying developers who can work on the same project on the basis of similar experiences and interests and not on the basis of possessing common skills that can obviously be shared within different projects. To validate the proposed approach, we carried out an experiment where we studied the effectiveness and the performance of the proposed community detection techniques by considering different instances of DSNs and different types of graph embedding techniques.

The remainder of the paper is structured as follows. Section 2 reports the Related Work. Section 3 illustrates the framework we defined to instantiate a DSN and to implement the community detection task, relying on the social model of GitHub. Section 4 shows the experiment we performed to validate the proposed approach with the related results while Section 5 eventually reports conclusive remarks and future work.

2. Related work

In this subsection the state of the art of the literature regarding existing GitHub datasets, approaches to model Developer Social Networks, and proposed solutions for the problem of Community Detection in the context of Developer Social Networks will be presented.

2.1. GitHub information models

Several works in the literature have proposed information meta-models and data models for information mining from platforms such as Github.

GHTorrent represents the Github dataset that has been most frequently used by the scientific community. It was originally proposed in 2012 by Gousios and Spinellis (2012) and Gousios (2013). They extracted a dataset from GitHub making it available both in form of csv files and as a relational database. GHTorrent contains a broad spectrum of information about Developers and Repositories, including information about commits, issues and pull requests. This dataset and its subsequent updates have been the basis for many scientific works. In particular, the use of GHTorrent has been promoted by the community at the Mining Challenge at MSR 2014[2] from which many examples of works based on this dataset were presented (e.g. Matragkas et al. 2014, Rahman and Roy 2014 and Sheoran et al. 2014). Unfortunately, the execution of community detection algorithms on very large relational databases or csv files is very onerous, thus different database models should be considered.

Software Heritage[3] is a graph database including information extracted both from Github and from other similar infrastructures such as Gitlab and Bitbucket. Software Heritage provides a web interface and a set of APIs to browse and queries its data. Software Heritage has been used as a dataset by many recent works, thanks to its efficiency and scalability (e.g. Bhattacharjee et al. (2020), Boldi et al. (2020) and Pietri et al. (2020)). Software Heritage's information meta-model focuses on the history of projects making it suitable for studies about their temporal evolution.

Many other studies in the scientific literature proposed graph databases to represent the information extracted from GitHub. For example, in Luo et al. (2015) a graph database was proposed to model Developers, Repositories and a limited set of relationships including Commits, Comments and Watch (between a Developer and a Repository) and Forks (between Repositories). Other examples of graph based models of GitHub information have been successively proposed (Cai et al. 2016, Di Rocco et al. 2020, Geiger et al. 2018, Hu et al. 2016, Leibzon 2016, Nguyen et al. 2020, Seifer et al. 2019, Sülün et al. 2021, 2019, and Zhang et al. 2017). Unfortunately, most of these graph databases are not updated or are not more available, thus in this work we propose another model including a subset of information allowing the execution of effective and efficient Community Detection algorithms.

2.2. Developer social networks

A large set of GitHub mining activities aim at studying developers and the relationships between them.

Many studies are focused on aspects related to the developers popularity (e.g. Batista et al. 2017, Dabbish et al. 2012, Hou et al. 2021, Jarczyk et al. 2014, Jiang et al. 2013, Marlow et al. 2013 and Zihayat et al. 2014). Another problem often approached by studies based on GitHub information is the Expert Finding, in which techniques for the effective search of developers with specified skills are based on the analysis of their activity on GitHub (e.g. Batista et al. 2017, Dabbish et al. 2012, Hou et al. 2021, Jarczyk et al. 2014, Jiang et al. 2013, Marlow et al. 2013 and Zihayat et al. 2014). The characterization of working groups of Developers sharing the participation to the same GitHub repositories is the object of several other relevant works (e.g. Avelino et al. 2016, Matragkas et al. 2014, Pinto et al. 2016, Wang and Perry 2015 and Yamashita et al. 2015).

The studies that are most similar to ours are those related to the modeling of Developer Social Networks and the consequent detection of community of developers. Recently, the systematic study by Herbold et al. (2021) has classified 255 scientific papers in the broader field of Developer Social Networks. In most of them, these networks have been modeled as oriented graphs in which the nodes represent the developers and the edges represent the relationships between them that can be inferred by the analysis of infrastructures such as GitHub. In particular, the authors highlight that in this specific area, there is a lack of shared datasets and that most of the works were based on the analysis of few projects (only 50 studies analyzed at least 100 projects)

2.3. Community detection on developer social networks

The specific task of Community Detection in the context of GitHub developers has been faced by several recent works in the literature. In all these works a graph is built on the basis of the information extracted from GitHub, in which the nodes represent the developers and the edges between two developers represent the existence of shared repositories to which both developers have contributed in terms of commits (e.g. in Batista et al. 2017 and Hajiakhoond Bidoki and Sukthankar 2018). In addition, some other information such as the comments to the same issues or pull requests are considered in Schettino et al. (2019) and Wang et al. (2019). All these works apply different metrics to weigh the edges representing the collaboration between two developers and use novel or well-known clustering algorithms to detect Developer Communities. Unfortunately, most of these works remain at a proposal level, without a wide experimentation or available material for replication purposes, thus they are not able to provide evidence of the existence of a better performing weighting model.

[2] http://2014.msrconf.org/challenge.php
[3] https://www.softwareheritage.org/

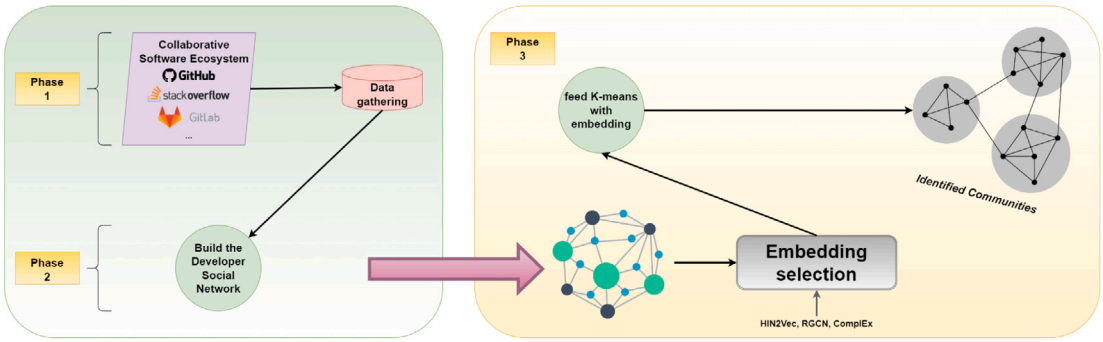

Fig. 1. Framework overview.

The work that is most similar to ours and that provides more details about its replication is the one presented in 2021 by Hou et al. (2021). They propose and validate Community Detection techniques applied to a graph whose nodes are represented by the developers and whose edges represent the existence of a shared repository on which both the developers commit changes. The edges are weighted on the basis of a metric called Developer Intensity Cooperation that takes into account the quantity of commits of the two Developers on shared repositories. Different clustering algorithms were proposed and compared in order to detect communities of developers. The cohesion of such clustering was evaluated by means of topological metrics. The authors validated their approach on the basis of information extracted from GitHub regarding the activity of several thousand of Developers over a two-year period, between 2017 and 2019 and on popular projects (characterized by a large quantity of 'stars' given by GitHub users).

3. Framework

Our framework aims to support community detection task within a DSN environment, which has been treated by three main phases: *Data Ingestion*, *Data Modeling* and *Community Detection*. In the first phase, we crawl data from a collaborative software repository. Successively, we process the extracted data to build the DSN graph structure, where the nodes consist of developers, repositories and issues connected by different kinds of edges. The last phase involves the community discovery process, where we first embed the DSN in a lower dimensional space and then divide the multidimensional points into a set of clusters (communities). Fig. 1 provides an overview of the three phases.

3.1. Task definition

Understanding user relationships within a developer social networks can be useful for different analytic tasks (e.g., team formation or expert finding). Despite different effort in finding overlapped communities of developers sharing common skills, we are more interested in analyzing the problem of identifying developers who can work on the same project on the basis of similar interests, that is typically addressed as a non-overlapped task (see Hou et al. 2021, Mockus et al. 2020 and Moradi-Jamei et al. 2022). For this reason, we are interested in identifying developers' communities, composed of users sharing the same interests in terms of similar projects and/or applications. In more clear terms, the goal of our task is to identify a sub-set of developers in the DSN who share the same interests in terms of their interactions on given repositories (commits, stars, forks, etc.).

Definition 1 (*Task Definition*)**.** Our task can be modeled as a particular function f associating to each developer d_i belonging to the developers' set D ($d_i \in D$) one of the K communities $C_j \in \mathcal{P}(D)$ that can be seen as non overlapped subsets of D ($\cup_{j=1..K} C_j = D \wedge \nexists z \neq j : C_j \cap C_z \neq \emptyset$):

$$f : d_i \rightarrow C_j \tag{1}$$

The number of obtained communities depends on the number of developers and on the adopted features for their characterization.

3.2. Modeling GitHub as a DSN

Collaborative software repositories contain heterogeneous information about developers, repositories and issues, which, in our opinion, can be summarized as a list of actions that a developer can perform on an object (issue or repositories).

Fig. 2 shows a conceptual model describing collaborative software repositories (e.g., Github) and the main interactions in that context. The model is depicted as a UML class diagram having three classes, namely *Developers*, *Repositories*, and *Issues*, and 13 associations among them.

Developers are registered users of a collaborative software repository system, who can create and become *owners* of new *Repositories*, i.e. collection of source code and artifacts about a given project. A new repository can also be created by means of a *Fork* operation on an existing one, creating a clone of the original repository owned by another developer. The owner of a repository can invite and qualify other developers as *Collaborators*. A collaborator may manage the repository with almost the same rights of the owner. In particular, both the owner and every collaborator can *commit changes*, i.e. submit updates of any repository files. On the other hand, a developer, even if not directly involved in a project, may propose a contribution (for example the implementation of a new feature or the correction of a bug) via a *Pull Request*: the owner of the repository and his collaborators can approve it and apply these changes or refuse it. When a developer wants to express his interest in all the activities of another Developer he can decide to *Follow* him: in this case he will receive notifications of all the public activities made by the developer on Github repositories. This is a common feature of many social networks and represents the unique direct connection between the two developers. If a developer is interested in the evolution of a repository (owned by another developer), he can *Watch* it: in this case he will receive notifications of the changes of the project on his dashboard. A simpler way a developer has to express his appreciation for an existing Repository is by flagging it with a *Star*, which is a mechanism equivalent to those of many social networks (i.e. "like").

Another common interaction to be considered regards *Issues*. An issue is a message written by a developer (involved or not in the project) used to communicate the existence of a problem or a bug in a project, or the request for a new functionality, or simply a comment about the project. Unlike pull requests, no changes are associated to Issues. An issue has its own life cycle, which starts with its opening by a Developer and continues with a possible dialogue between this Developer, the Repository owner and his collaborators and possibly other external developers. During these interactions, the issue can be approved by Owners and *Assigned* to one of the developers involved in the project for its resolution. An issue may be *Closed* when it is considered resolved or not relevant.

Several meta-models have been proposed with the goal of handling the different types of entities and relationships within a collaborative software repository (for example GHTorrent (Gousios, 2013; Gousios &

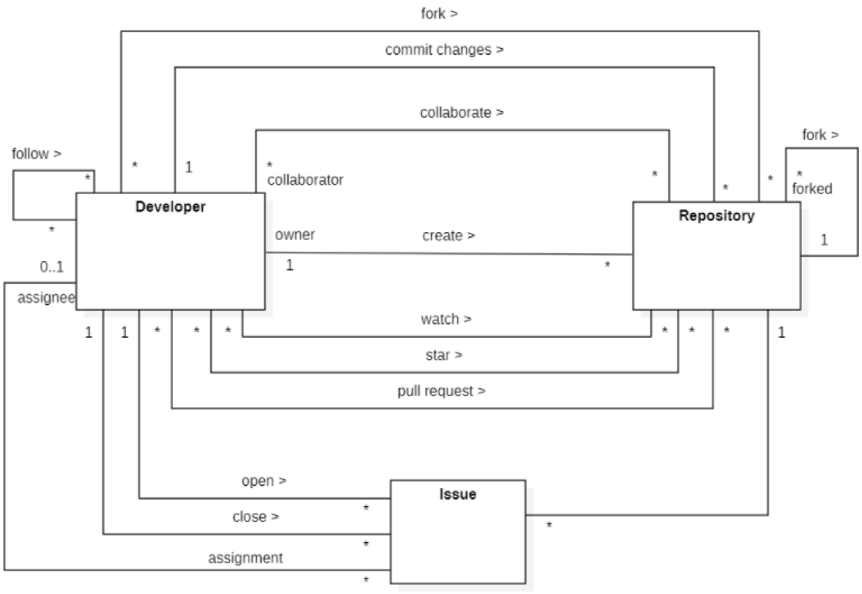

Fig. 2. Conceptual model of a collaborator software repository.

Spinellis, 2012) and Software Heritage,[4]) but without focusing on the relevance of the examined relationships for a specific analysis task. On the other hand, other meta-models are oversimplified or target-focused on specific aspects of open source repositories, such as the commit actions of developers on the same repositories (see Hou et al. 2021 for more details).

Our data model shown in Fig. 2 tries to capture and handle all the most useful information from a collaborative software repository for DSN analytics purposes. Our model considers both information demonstrating direct relationships between two developers (e.g. the follow and the collaboration relationships) and indirect relationships between developers (e.g., both participation in the activities on the same Repository) and demonstrates how they can be very useful for DSN analysis. As delineated in Section 1, the proposed information model relies on a heterogeneous graph structure that accommodates diverse entities, such as developers, issues, and repositories, as nodes, and allows the existence of multimodal relationships within a Developer Social Network (DSN). It is noteworthy that the suggested information model differs from the one proposed by Hou et al. (2021), which employs a homogeneous graph structure using only developer nodes and a single edge type, i.e., collaboration intensity, to summarize information about the joint commits performed by different developers on the same repositories. To exploit the rich semantics present within our proposed information models based on heterogeneous networks, graph embedding techniques were necessary to handle analytical tasks on high-dimensional graphs (Yang et al., 2020). We propose to leverage data detailed in Fig. 2 by using a heterogeneous information network (our DSN). The aim is to fuse multi-typed interacting components to improve semantics related to nodes and links in order to group together developers who have shown interest in the same repositories and/or applications (*Community detection* task).

Definition 2 (*Developer Social Network*)**.** Let D, I and R be respectively the sets of developers, issues and repositories, we define the *Developer Social Network* as an oriented graph $G = (V, E)$, where V is the heterogeneous set of vertices ($V = D \cup I \cup R$) and E is the edge set that can be established between two nodes.

[4] https://www.softwareheritage.org/

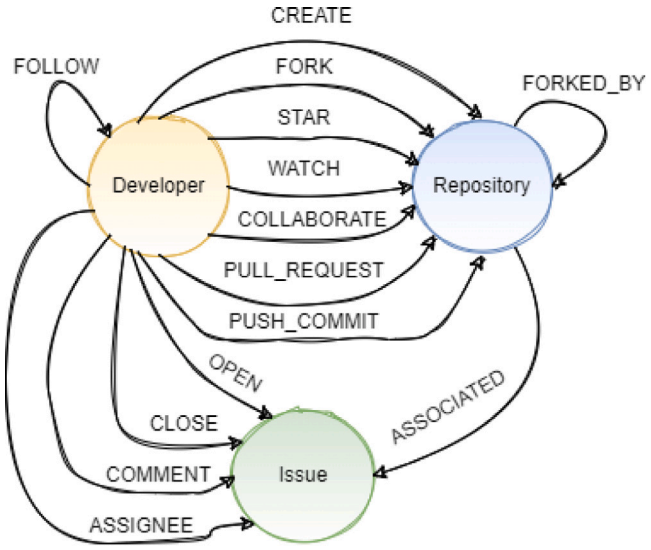

Fig. 3. Developer social network model.

Fig. 3 shows at a glance our idea of DSN, which is composed by three entities: (i) *Repositories*, (ii) *Developers*, (iii) *Issues* . The Figure also summarizes the different types of actions or relations among developers, repositories, and issues, that we will rely on in the considered community detection tasks.

3.3. Community detection process

On the basis of the introduced DSN, our goal is to detect the set of communities whose members are developers showing some common activities on several repositories.

Despite different community detection approaches have been proposed in the literature on heterogeneous graphs, and in particular, on those supporting DSN analysis from a social perspective, the majority of them suffers of computational and memory costs due to the sizes that such graph-based structures can reach (Fu et al., 2021).

Representation learning involves automatically finding and learning a latent representation of data that can be used as input features for supervised machine learning algorithms for various prediction tasks (Bengio et al., 2013). In particular, *Graph embedding* aims to represent a graph in a low dimension space, while preserving network topology, thus facing with the graph large size challenge. Therefore, we propose a community detection model, whose aim is to unveil developers' community by relying on *Graph embedding* techniques for representing members of a DSN in a low-dimensional spatial representation that preserves the network topology, satisfying the 2nd order of proximity. Thus, we performed an embedding of the DSN to a n-dimensional space through graph embedding techniques, whose points are clustered into a set of communities by using unsupervised machine learning algorithm with the aim to optimize the number of clusters. In particular, our aim is to use an algorithm, satisfying fast convergence, scalability and fit for scattered data.

In Algorithm 1 we present our community detection algorithm based on graph embedding (*BERTO*). The input of our algorithm are: (i) the selection criterion used to select a subset of the developers and repositories on GitHub, (ii) the chosen graph embedding technique and (iii) the clustering algorithm. The output of *BERTO* will be $C = \{C_1, \ldots, C_n\}$ the set of the discovered communities.

The *BERTO* algorithm is given as follows: (i) a subset of relationships between Developers, Issues and Repositories are extracted from Github according to a given Selection Criterion (*SC*), (ii) while there are relationships to analyze (iii) an element r is fetched from the relationships set R, and (iv) it is added to the DSN. (v) The whole DSN is given as input to the chosen embedding *GE* technique to compute the nodes embedding that will feed (vi) the clustering Technique *CT* responsible for providing in output the identified K communities. The Selection Criterion *SC* is used for choosing and categorizing the data to be retrieved from GitHub in terms of: programming language, the time frame concerning the repository creation date, and the number of stars assigned to the repositories. There are several factors that can influence the choice of clustering techniques, such as the type of data to be analyzed, available resources, and time constraints. In the case of the BERTO algorithm, we chose K-Means as our clustering technique because of its time-efficiency and manageable hyperparameter tuning phase, which are important for practicality and efficiency in real-world scenarios (Chen et al., 2022; Minh et al., 2022). K-Means has a time complexity of $O(n)$, which is superior to other partition clustering algorithms such as *CLARANS* and *PAM*. It also outperforms clustering algorithms of other categories such as density-based algorithms like DBSCAN and OPTICS, and hierarchical ones like Chameleon, in terms of time complexity. However, K-Means' scalability is limited as the dimensionality of the input data increases, as observed in (Saxena et al., 2017). To address this issue in the BERTO algorithm, we use a graph-embedding stage that reduces the dimensionality of the input data. Another advantage of K-Means is that it only requires the identification of the optimal number of clusters (K), whereas other algorithms such as DBSCAN require tuning of other parameters for optimization. This tuning process can be more complex than the one used for K-Means, which can simply use the elbow rule to identify the optimal value of K.

The computational complexity achieved by our proposed algorithm *BERTO* is $O(n\ logn)$, given from a cost of (i) $O(n)$ to explore all the relationships set ($n = |R|$), (ii) $O(n)$ for the graph embedding and clustering phase, (iii) and $O(logn)$ for the graph updating phase.

Algorithm 1 *emBEdding gRaph communiTy detectiOn* (*BERTO*)-algorithm

procedure EMBEDDING GRAPH COMMUNITY DETECTION (*BERTO*)-ALGORITHM(SC,GE,CT,K)
  – Input: SC (Selection Criteria)
  – Input: GE (Graph embedding)
  – Input: CT (Clustering Technique)
  – Input: K (Number of Clusters)
  – Temporary: R (Relationships Set)
  – Temporary: DSN (Developer Social Network)
  – Temporary: V_e (List of node embeddings)
  – Output: C (List of identified community)
  R← QUERY_GITHUB(SC)
  while ($R \neq \emptyset$) **do**
    r ← dequeue(R)
    DSN ← UPDATE_GRAPH(DSN,r) ▷ DSN building
  V_e ← EMBEDDING_GRAPH(DSN,GE) ▷ Nodes list V_e with graph embedding GE on the DSN
  C ← IDENTIFICATION_COMMUNITY(V_e,CT)
  return C ▷ List of identified communities

3.4. Running example

In this section, we describe a running example of how it is possible to generate a DSN from GitHub data and how it is possible to detect communities of developers by embedding the DSN graph. We suppose to rely on the following information:

1. $(commit, Giancarlo, Repo1)$
2. $(commit, Antonino, Repo1)$
3. $(open\_by, AnnaRita, Repo2)$
4. $(pull\_request, Porfirio, Repo2)$
5. $(comment, Marco, Issue1)$
6. $(watch, Giancarlo, Repo2)$
7. $(comment, Giancarlo, Issue1)$
8. $(star, Marco, Repo2)$
9. $(comment, Porfirio, Issue2)$
10. $(open\_by, Vincenzo, Issue2)$
11. $(comment, AnnaRita, Issue2)$
12. $(create, Vincenzo, Repo3)$

Each operation is characterized by: (i) a relationship (a commit, a creation of repositories and so on) (ii) a source and destination object.

The obtained DSN, shown in Fig. 4, is then fed as input to the embedding algorithm (in this example *Complex*), which provides an embedding representation of each developer node within our network. Successively, an euclidean distance between two nodes has been applied in order to build the distance matrix, shown in Table 1.

Finally, the vectors obtained from the embedding step are given as input to the K-Means clustering algorithm. Choosing $K = 2$, the community partition obtained is:

$C_1 = \{AnnaRita, Giancarlo, Marco, Porfirio, Vincenzo\}$
$C_2 = \{Antonino\}$

We expected the obtained clusters because we can see from Example 1 how the developer node *Antonino* has only a unique connection to a single repository and in common with only one user, namely *Giancarlo*. In turn, the other users show more collaboration activities, as example, developer $Vincenzo$ despite being connected to only two entities ($Issue2$ e $Repo3$), the issue he worked on is a node of strong interest for the other developers as well, which makes $Vincenzo$ also strongly connected with the other developers who worked on the same issue. In conclusion, it can be interpreted that $AnnaRita$, $Giancarlo$, $Marco$, $Porfirio$ and $Vincenzo$ are probably interested in the same projects, or are close collaborators (as, for example, can be inferred from their interactions on $Repo2$), while *Antonino* has no close ties with any of them; in fact, only $Giancarlo$ has interaction with *Antonino* on a marginal project $Repo1$.

Table 1
Distance matrix.

	Anna Rita	Antonino	Giancarlo	Marco	Porfirio	Vincenzo
Anna Rita	0	0.7	0.3	0.2	0.2	0.4
Antonino	0.7	0	0.6	0.6	0.7	0.8
Giancarlo	0.3	0.6	0	0.3	0.2	0.5
Marco	0.2	0.6	0.3	0	0.3	0.5
Porfirio	0.2	0.7	0.2	0.3	0	0.4
Vincenzo	0.4	0.8	0.5	0.5	0.4	0

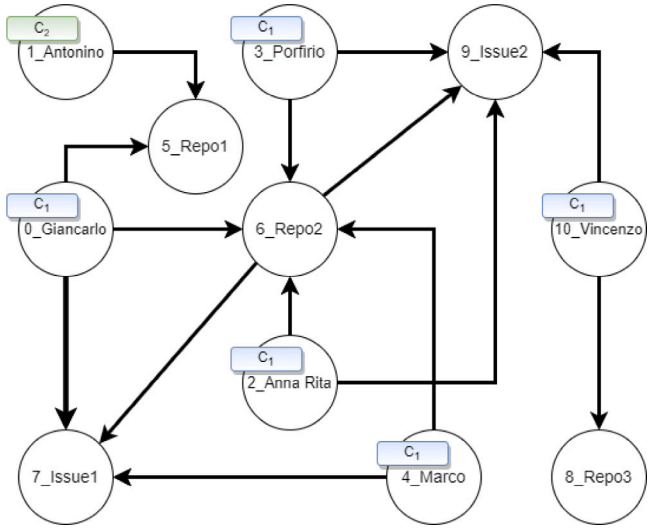

Fig. 4. DSN representation for the Running Example. The heterogeneous graph is composed by 11 nodes and 12 edges.

4. Experimental analysis

4.1. Goals

The objective of this experimentation is the evaluation of the proposed Community Detection algorithm *BERTO*. We want to evaluate and compare the performance of the technique obtained by applying different graph embedding techniques and find the best-performing one. In addition, we want to compare the performance obtained by adopting the proposed information model with respect to the ones obtained by adopting the model proposed for the *ABDCI algorithm* of Hou et al. (2021). Our information model is based on a heterogeneous graph that takes into account not only developers but also other entities such as issues and repositories, along with their diverse types of relationships, as shown in Fig. 2. In contrast, Hou et al. (2021) used a homogeneous graph, defined as a *Software Ecosystem Network* (SEN) by the authors, that only represents developers connected by a single type of edge. The aim of our experiment is to determine whether the use of a heterogeneous graph, can facilitate a more distinct identification of communities compared to a homogeneous graph. Additionally, we aim to compare the time complexity of the two approaches. Furthermore, the community detection algorithm (ABDCI) (Hou et al., 2021) is based on hierarchical clustering and achieves a time complexity of $O(n^2)$, which is higher respect to the obtained by our proposed approach (BERTO) based on K-Means and equal to $O(n\ logn)$. In order to assess the performance of the technique, we evaluate the quality of the detected Developers Communities, using a modularity metric to reward clustering that reveals cohesive communities of developers. Furthermore, we intent to evaluate the robustness of the proposed technique as different samples of developers and repositories from GitHub are considered, having different sample sizes and different repository selection criteria.

Finally, we want to study the computational cost and time needed for the execution of the community detection algorithm on different information samples.

4.2. Research questions

In order to pursue the presented goals, we have formulated four research questions to be answered by the experimentation.

- RQ1 How does the modularity of the Developers Communities produced by the *BERTO* algorithm vary depending on the adopted graph embedding technique?
- RQ2 How does the modularity of the Developers Communities produced by the *BERTO* algorithm vary with different information models?
- RQ3 How does the modularity of the Developers Communities produced by the *BERTO* algorithm vary for different Github repositories samples?
- RQ4 How does the execution time needed to evaluate the Developers Communities produced by the *BERTO* algorithm vary for different Github repositories samples?

4.3. Variables and metrics

4.3.1. Independent variables

Different sets of independent variables have been considered in the different phases of the experiments we carried out, i.e. the *graph embedding technique*, the *information model*, the *repository sample type*, the *repository sample size* and the *embedding* and *cluster size*. Comprehensive information regarding the aforementioned variables can be found in Table 3.

In the context of the research question RQ1, we considered three categories of *graph embedding technique*, based on how they capture the network topology (Cai et al., 2018) and used three different algorithms as representatives of these categories: (i) Proximity Preservation methods, (ii) Message-Passing methods, (iii) Relation Learning methods. Specifically, an embedding algorithm was selected for each category, *Hin2Vec*, *RGCN* and *ComplEx*, respectively.

1. *HIN2Vec*: a proximity preservation method based on random walk and meta-path. Taking as input a Heterogeneous Information Network (HIN) and a set of meta-paths, the HIN2Vec framework performs multiple prediction training tasks, that are jointly based on the targeted set of relationships taken as input (Fu et al., 2017).
2. *RGCN*: based on message passing, it aims to learn node embedding by aggregating the information from the neighborhood. RGCN differs from standard link predictors because it employs node neighborhood information to learn the vector representations of the node/graph (Schlichtkrull et al., 2018; Thanapalasingam et al., 2021).
3. *ComplEx*: a relation learning method, that considers complex-valued embeddings. The use of complex embedding allows for binary relationship management, both symmetric and asymmetric. Link prediction can be solved as a binary tensor completion problem, where each slice of the tensor is an adjacency matrix of the relationships that are present in the graph (Trouillon et al., 2017, 2016).

In the context of the research question RQ2, we considered two different *information models*, i.e. the one presented in Section 3 w.r.t. the one built in Hou et al. (2020) that considered only the *joint commit* interactions. In that work, Hou et al. presented the ABDCI algorithm, which combines developer interaction information and network topology information and defined the intensity of collaboration in order to build a DSN. They assumed that developers executing commits into the same repository always have similar skills and preferences and should be in the same cluster (Hou et al., 2021). The computational complexity of the algorithm *ABDCI* is $o(n^2)$ which is higher than the complexity achieved by our proposed model *BERTO*.

In addition, we have varied two parameters of the *BERTO* algorithm in order to study the parameter values combinations providing the better clustering performance, i.e. the *embedding size* and the *cluster size*.

The embedding size is a critical decision: the key factors for the optimal selection are mainly related to the availability of computational resources and the choice of a trade-off value that offers complexity reduction and no information losses, so the embedding size will determine the level of information compression. A larger size offers a model with high information content, at the expense of high complexity, while a smaller size will offer higher computational performance but losing the heterogeneous information in the model. The performance of the clustering is closely related to the choice of the cluster size value K. It is important to consider values that are reasonably large, in order to reflect the characteristics of the dataset, but at the same time significantly small compared to the number of objects to be analyzed, which is why we desire clustering of the data (Pham et al., 2005).

4.3.2. Dependent variables

We have considered two dependent variables, i.e. the *modularity* of the detected developer communities and the *time* needed to execute the *BERTO* algorithm.

To evaluate how good is the quality of the division of the networks into communities, we use a quantitative and topological measure called *Modularity* (Newman, 2016), whose score value is an indicator of the quality of the clustering (high values corresponds to better clusterings).

Modularity Q can be defined according to Eq. (2), where A_{ij} is a single element of the adjacency matrix A. In particular, $A_{ij} = 1$ if there is an edge that connects the nodes i and j, $A_{ij} = 0$ otherwise. g_i is the number of the community which node i belongs to, and $\delta(i, j) = 1$ only if $i = j$, that means that only if the two nodes belong to the same community $\delta(g_i, g_j) = 1$. k_i indicates the degree of the node i. Lastly $m = \frac{1}{2}\sum_{ij} A_{ij}$ is the total number of edges in the graph.

$$Q = \frac{1}{2m}\sum_{ij}(A_{ij} - \frac{k_i k_j}{2m})\delta(g_i, g_j) \tag{2}$$

Modularity turns out to be one of the most widely used metrics in optimization methods to detect communities in networks (Hou et al., 2021; Moscato & Sperlì, 2021). Its value may vary between −1 and 1: a higher value indicates a strong community structure (for more details see the paper Chakraborty et al. (2017)).

The recorded performances are evaluated by the aggregated execution times inherent to the embedding phase and the community identification phase.

4.4. Objects

The objects of the experiments consisted of selected subsets of repositories from GitHub, which were analyzed in order to extract the information to be represented by graphs. In order to obtain these graphs, we have performed different queries on GitHub adopting different filtering criteria. In particular, we have varied the values of three query parameters, i.e. *the number of stars* received by the repository (that could be an indicator of the general appreciation of the corresponding project), the *creation date* of the repository (that has been used to restrict the search to a set of projects created within a specific time window) , and the *programming framework* of the source code stored in the repository.

In detail, we focused on the repositories that have at least 30 stars, were created between January 1, 2020 and January 20, 2020 and have been labeled as *Python*, *Java* or *Android* projects. Once we retrieved the repositories with these characteristics, we captured for each of them all the information related to owners, collaborators, forks, issues and any other relationships of the data model presented in Section 3. The decision to start our search from repositories with a certain number of stars is because we wanted to consider popular repositories that arouse a high interest from GitHub users. It is important to note that, unlike Python and Java, the ”Android” category is a set of technologies (e.g. Java, XML, Kotlin, HTML5), thus we manually labeled the repositories devoted to the development of applications for the Android ecosystem.

In the context of RQ1 and RQ2 we have considered the graph built on the *Full* subset of repositories (which includes all three subsets of repositories), whereas in the context of RQ3 and RQ4, we have considered four different graphs, respectively based on the *Python*, *Java*, *Android* and *Full* subsets of repositories.

Table 2 reports the main characteristics of these four subsets of experimental objects, i.e. the number of repositories in each set, the total number of involved developers, the total number of found issues, and the total number of relationships.

4.5. Design of experiments

In order to answer each of the four proposed research questions, an experiment organized in four phases was carried out, where each phase is characterized by different sets of objects, independent and dependent variables and factors.

The selection of graph embedding algorithms for the experiment was guided by the work of Cai et al. (2018) which classifies embedding algorithms into three categories based on their method of capturing network topology: (i) Proximity Preservation, (ii) Message Passing, and (iii) Relationship Learning. For each category, we selected one embedding algorithm, respectively Hin2Vec, RGCN and ComplEx. As demonstrated in the response to RQ1, we found that ComplEx outperforms the other algorithms and thus was chosen as the embedding algorithm for our experimentation.

In the first phase, to answer RQ1, the sample of repositories called *Full* and the heterogeneous graph based on the information model presented in Section 3 were considered. We evaluated *modularity* (dependent variable) values obtained by considering three different graph embedding techniques (independent variable): *Hin2Vec*, *RGCN*, *ComplEx*. For this purpose, the *BERTO* algorithm was executed by varying two factors, i.e. the embedding size and the cluster size. In detail, for the embedding size the values of *50, 150 and 300* were considered, while for the cluster size K the values of *50, 150 and 250* were considered in order to evaluate which of these combinations produced the best modularity values.

In the second phase of the experiment aiming at answering RQ2, we considered the same sample of repositories (*Full*) but we varied two information models as independent variable: the one that we have proposed in Section 3 (BERTO) and the one considering commits on shared repositories as the only relationship between developers, coherently with the *ABDCI algorithm* proposed by Hou et al. (2021). The *modularity* of the detected community was considered as dependent variable, and it was evaluated by also varying in this case the embedding size and the cluster size using the same set of values considered in the first phase of the experiment.

In order to answer RQ3, we considered different samples of repositories beyond the *Full* one. The other three considered samples are the ones called *Java*, *Python* and *Android*. In this phase we considered only the heterogeneous graphs based on the information model presented in

Table 2
Dataset characterization.

	Android	Java	Python	Full
Number of repositories	8,883	7,881	18,570	35,334
Number of developers	56,231	30,262	81,612	168,105
Number of issues	4,310	3,520	11,310	19,140
Number of relationships	961,433	155,389	1,128,458	2,245,280

Table 3
Design of Experiments: Phase 1 involves the determination of the most effective embedding technique in terms of modularity, from among the three options available. This process is carried out for each of the three embedding techniques. Phase 2 involves fixing the embedding technique selected in Phase 1 and considering the information model as the independent variable. Phase 3 involves changing the analyzed sample while keeping the information model and clustering technique constant, and observing the variations in modularity that occur across different values of cluster and embedding size. Phase 4, the execution time is established as the dependent variable, and the impact of sample size and clustering and embedding dimensions is evaluated.

	Phase 1	Phase 2	Phase 3	Phase 4
Research question	RQ1	RQ2	RQ3	RQ4
Objects	Full	Full	Full, Java, Python, Android	Full, Java, Python, Android
Independent variables	Graph Embedding technique (Hin2Vec, RGCN, ComplEx)	Information Model (Berto, ABCDI Model)	–	–
Fixed factors	Information Model (Berto)	Graph Embedding technique (ComplEx)	Information Model (Berto), Graph Embedding technique (ComplEx)	Information Model (Berto), Graph Embedding technique (ComplEx)
Varying factors	Embedding size (50, 150, 300), Cluster size (50, 150, 250)	Embedding size (50, 150, 300), Cluster size (50, 150, 250)	Embedding size (50, 150, 300), Cluster size (50, 150, 250)	Embedding size (between 50 and 500), Cluster size (between 50 and 300)
Dependent variables	Modularity	Modularity	Modularity	Execution Time

Section 3. The dependent variable in this phase is still the *modularity*, and the variable factors are still the embedding size and the cluster size.

Finally, in the fourth phase of the experiment, we have considered the same objects involved in the third phase but we have considered a different dependent variable, i.e. the *execution time*. The execution times were obtained via the Python library "time". This time was measured when we executed our technique in a cloud-based execution environment built on Google Colab,[5] involving a Python 3.6 implementation of the Community Detection algorithm *BERTO* and an execution engine composed by 2 CPU (2.2 GHz), 13 GB of RAM. The *NetworkX* and *SciKit-Learn* libraries were used for analyzing graphs and using unsupervised learning algorithms. Also in this phase, we have considered embedding size and cluster size as factors to be varied, but we have considered larger samples of possible values: the clustering size varied between 50 and 500, while the cluster size varied between 50 and 300.

The characteristics of the four phases of the experiment are summarized in Table 3.

4.6. Results

In this Section we report the results we obtained in each phase of the experiment and the corresponding answers to each considered RQ:

4.6.1. RQ1

The execution of the BERTO algorithm for the three adopted graph embedding techniques and for different values of embedding size and K produced the results shown in Table 4. Due to computational limitations we were unable to compute the embedding vectors for RGCN with embedding size = 300. We can observe that the adoption of the ComplEx algorithm produced the greater values of modularity w.r.t. the ones obtained with the other techniques, for any value of embedding size and K, thus we can affirm that the ComplEx techniques appear the most suitable to this problem.

The poor results in terms of modularity obtained by HIN2Vec (Fu et al., 2017) can be attributed to the fact that this learning framework is based on meta-paths and thus is unable to obtain a latent representation of nodes that reflects the network topology.

In the RGCN (Schlichtkrull et al., 2018) framework, the vector representation of neighboring nodes is collected and then transformed for each type of relationship separately. It turns out that the calculation of node embedding is done using only one vector from the neighborhood. Therefore, RGCN is unable to compute a representation of the network topology in the embedding space.

This type of issue has not shown up with Complex (Trouillon et al., 2016), which uses complex valued embedding. In particular, when there is only one kind of relation between entities Complex uses the real part of low-rank normal matrices to represent the relationship, allowing the framework to capture the topology of the networking inside the computed embedding vector of the nodes.

4.6.2. RQ2

Answering RQ1 we concluded that ComplEx is the best performing embedding technique among the three ones we proposed. To answer RQ2 we executed *BERTO* algorithm with ComplEx embedding technique for the two different information models and for different values of embedding size and *K*. Table 5 reports the obtained *modularity values* after applying *BERTO* to our model where we consider all types of interaction, and the *modularity values* obtained using the *ABDCI algorithm* proposed by Hou et al. (2021), which is only based on shared commits. Our model does not seem to give us a better result in terms of modularity. We therefore investigated how developers are distributed across communities for the two techniques we are comparing. Based on 30 experiments, in Table 6 we show median and standard deviation of the node distribution inside the communities/clusters. Focusing on the standard deviation, we can see that our model has smaller values than the model proposed by Hou et al. (2021). This means that the *BERTO* algorithm applied to our information model, despite not always obtaining better modularity values than the *ABDCI algorithm*, is able to find a more balanced community division, i.e. a more equal distribution of nodes within the various communities.

High standard deviation values indicate that the distribution is unbalanced, meaning that there will be few clusters, or in the worst case only one cluster, with most of the nodes inside, whereas all the other clusters will contain few nodes. Having all developer nodes within the same cluster is not an optimal outcome in a community detection task. Instead, our model, by obtaining a lower STD value, was able to obtain a more balanced distribution of developer nodes within clusters, meaning that it was able to better categorize the various developers in the different communities, providing us a more meaningful result than that obtained by *ABDCI algorithm*.

[5] https://colab.research.google.com/

Table 4
(RQ1) - Results of *BERTO* algorithm in terms of modularity by varying technique and embedding size.

BERTO algorithm modularity									
K	emb_50			emb_150			emb_300		
	Complex	RGCN	HIN2Vec	Complex	RGCN	HIN2Vec	Complex	RGCN	HIN2Vec
50	77,75%	4,21%	2,71%	82,64%	2,11%	2,32%	78,57%	N/A	3,59%
150	67,44%	2,91%	0,82%	77,19%	1,41%	1,16%	72,49%	N/A	1,25%
250	55,18%	2,17%	0,73%	68,23%	1,47%	0,91%	64,55%	N/A	1,02%

Table 5
(RQ2) - Comparison of the average modularity, varying *K*, for *BERTO* and *ABDCI* (Hou et al., 2021).

Modularity comparision						
K	emb_50		emb_150		emb_300	
	BERTO	*ABDCI*	*BERTO*	*ABDCI*	*BERTO*	*ABDCI*
50	77,75%	68,35%	82,64%	86,34%	78,57%	82,31%
150	67,44%	35,83%	77,19%	80,74%	72,49%	76,90%
250	55,18%	25,27%	68,23%	73,49%	64,55%	69,86%

Table 6
(RQ2) - Comparison of the Median and Standard Deviation (STD) of community members, varying K, for *BERTO* and *ABDCI* (Hou et al., 2021).

Median and STD Comparison												
K	emb_50				emb_150				emb_300			
	BERTO		*ABDCI*		*BERTO*		*ABDCI*		*BERTO*		*ABDCI*	
	Median	STD	Median	STD	Median	STD	Median	STD	Median	STD	Median	STD
50	44,12	73,97	38,10	130,79	58,08	35,41	53,04	50,48	53,03	50,99	49,60	57,93
150	14,03	26,08	16,10	24,73	18,10	12,96	19,00	15,98	19,09	14,08	18,07	17,39
250	6,53	16,78	10,02	12,57	10,01	9,37	10,00	11,36	11,03	7,50	11,04	8,42

Table 7
(RQ3) - Evaluation of Modularity for the four considered repository samples (Full, Python, Java, Android).

Modularity evaluation with different samples												
K	emb_50				emb_150				emb_300			
	Full	Python	Java	Android	Full	Python	Java	Android	Full	Python	Java	Android
50	77,75%	84,97%	78,77%	62,81%	82,64%	81,64%	69,83%	72,39%	78,57%	78,91%	64,89%	70,59%
150	67,44%	80,73%	28,81%	34,79%	77,19%	69,29%	35,62%	53,75%	72,49%	64,53%	32,63%	52,46%
250	55,18%	59,27%	14,41%	22,59%	68,23%	56,01%	19,02%	38,98%	64,55%	55,63%	17,39%	38,02%

4.6.3. RQ3

In order to have a confirmation of the validity of our algorithm with different graphs corresponding to different samples of repositories, we executed *BERTO* algorithm with ComplEx embedding technique with our Information Model for different Github repositories samples (Java, Python and Android), and for different values of embedding size and *K*. Table 7 reports the modularity achieved for each considered combination, showing that our model is robust in terms of modularity when varying the sample analyzed.

4.6.4. RQ4

In Figs. 5 and 6 are reported the measured execution times for the *BERTO* algorithm using ComplEx embedding techniques applied to our Information Model with the different considered samples of repositories. In particular, in Fig. 5, we see that the execution time increases linearly as the sample size and embedding size increase. The largest times are recorded with the *Full* sample, regardless of the embedding size; this is to be expected because it is the larger than the others.

Another point we can observe from Fig. 5 is that as the number of nodes within our graph increases, the execution times increase more as the embedding size increases. In fact, by placing our attention on the blue line in Fig. 5, which represents the case where we are analyzing the graph of the *Full* sample, we can see that the execution times increase faster than in the cases where the analyzed samples are smaller (the other lines in the Figure). This means that the embedding size should be chosen on the basis of the size of the graph to support more efficiently the SNA tasks.

Instead in Fig. 6, by varying *K* it is possible to observe the execution times, depending on the size of the embedding. It can be seen that the times decrease by reducing *K* and the embedding size, conversely they increase with a bigger *K* and the embedding size. Furthermore, from Fig. 6 we can notice that varying *K* does not have a relevant impact on execution time.

Looking at the modularity we obtained in the previous experiments, and focusing on Fig. 5, we can clearly say that we get the best modularity results with high values of *K* and embedding size. However, these two parameters impact execution times, leading us to have high execution times as their values increase. In case we have time to run our experiments, it is therefore better to choose higher values of *K* and embedding size, since with high values of embedding size we are able to capture the topology of the network better and better, and get higher results in terms of modularity.

On the other hand, in the case where we want to be careful about execution time, there is clearly a need for a trade-off between execution time and the modularity. For example, in our experiment shown in Fig. 5, we can see that once we pass the threshold of embedding size equals to 250, execution times begin to grow much faster as this value increases, so it is reasonable to choose an embedding size value between 200 and 250.

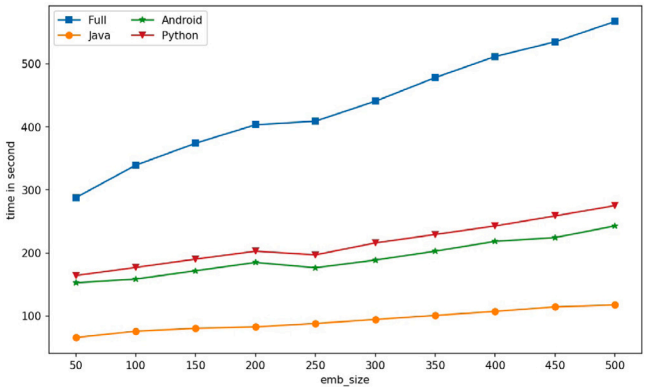

Fig. 5. Running times by varying the embedding size in *BERTO*.

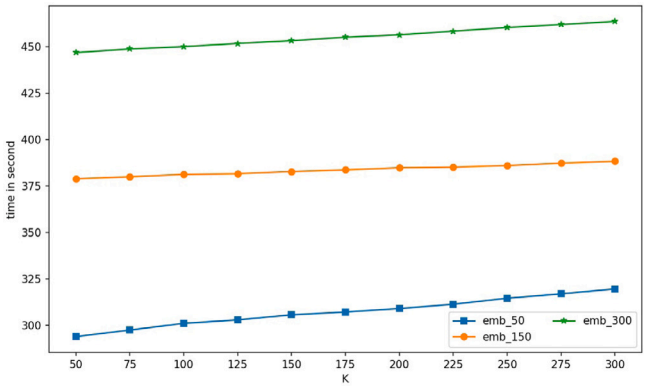

Fig. 6. Running times by varying K value in BERTO.

In practice, depending on the acquired data a parameters tuning phase is necessary to obtain a trade-off between community goodness and execution times.

4.7. Example

To evaluate the practical relevance of the communities detected by our algorithm we have selected a sample of clusters and a sample of the developers belonging to these clusters and we have analyzed in detail their activities. In the following we will present the results of the analysis involving three developers that were assigned to the same cluster. For privacy reasons, we have hidden the names of the developers and of the repositories involved in this example.

We evaluated cluster #16 from the clustering with K=50 clusters having the higher Q value. 71 different developers belong to this cluster. Among them we found many developers who created or collaborated in repositories related to the development of utilities for the Android environment, in particular to mining of data during the use of Android devices. For example, we verified that the developer *Dev1* from Florida, US, created the *Rep1* repository to implement an utility for collecting stack traces generated by unexpected crashes of Android applications on rooted devices. This project was implemented in Java and Kotlin and had a good success in terms of numbers of project forks and stars (19 forks and more than 200 stars at the month of March 2023).

Dev2 is an Italian student who developed (and also published on Google Play) an Android app for monitoring the use of the network of apps running on an Android system, within a repository *Rep2*. This app was realized in Java and C and had a greater popularity on Github, having more than 100 forks and almost 1000 stars. *Dev1* is one of the collaborators to this project who also made some commits on it.

Dev3 from Peru is a developer who forked both *Rep1* and *Rep2* repositories and continued developing them in the context of his forks. *Dev3* is a less active developer than *Dev1* and *Dev2*, but it can be seen from his public profile that he is mainly interested in Android projects and has often made changes related to their localization. With respect to this subset of developers that the BERTO algorithm grouped together within the same cluster, we may conclude that this grouping seems significant, since these developers actually share common interests and have worked on common projects. In future, it is reasonable that these three developers may collaborate in other projects based on Android internals.

4.8. Threats to validity

Threats to Conclusion Validity. There are threats to conclusion validity due to the selection of the sample of repositories involved in our study and to the choice of the information model.

The experiment carried out in this study involves a sample of the public repositories currently on Github. They have been chosen in an arbitrary way by filtering the ones that have been created on a short period of time and that rapidly achieved a large number of stars. This selection may represent a threat to the conclusion validity because this sample could be not representative of the overall set of the public repositories on Github, but we think that our selection is able to limit possible selection biases. In order to achieve more confidence on our conclusions, in the context of the third phase of the experiment, we have evaluated the modularity of the Developers Communities produced by the BERTO algorithm for three different Github repositories samples (obtained by respectively filtering only Java, Python and Android repositories).

Our conclusions about modularity and execution times depend on the amount of information considered in our model. Of course, a model including more details about the considered interactions (for example the text associated to comment, the result of a issue or a pull request), or a modularity metric weighting in a different way the different relationships (e.g. pull requests may have a greater weight with respect to comments on issues) may produce different results. We have concluded that our model, that is richer than the one proposed by the ABCDI one (Hou et al., 2021) is able to obtain better communities in terms of modularity, but we cannot exclude that another model could be better. In future work we plan to extend our analyses on other possible models, taking also into account the correspondent increase in execution time.

Threats to internal validity. Since the embedding algorithms are not completely deterministic, a threats to internal validity due to randomness is present. In order to limit this threat, the community detection algorithm BERTO has been repeated 30 times and the mean, median and standard deviation values have been evaluated.

Threats to external validity. These threats limit the ability to generalize the results of our experiment. In community detection problems it is difficult to define a ground truth of the expected communities that should be returned, because the concept of community is often elusive: a community includes people who have similar interests and involvement in the same or similar projects, but these developers could not be aware of their belonging to a particular community. For this reason, studies related to community detection in social networks often use only topological measures to assess the goodness of the detected communities. We have adopted the same modularity metric used in works similar to ours (Hou et al., 2021; Moscato & Sperlì, 2021) in order to have fair comparisons between the results of the different algorithms and configurations.

5. Conclusions

In this paper we addressed the problem of community detection inside a software repository ecosystem modeled as a particular social network and using SNA facilities. The main novelty of this work was the definition of a heterogeneous graph-based model able to capture and handle in an effective way a comprehensive set of useful, complex and strongly-correlated information about a DSN.

Then, we have challenged the problem of automatically discovering *communities* of developers sharing interests for similar projects, and we leveraged different graph embedding techniques together with clustering to overcome huge graph-size analysis issues. It should be noted that the embedding algorithms utilized in our study employ message passing techniques for information accumulation from neighboring nodes, as well as meta-path and random-walk based algorithms to compute embedding vectors representative of the initial graph (Yang et al., 2020). Although the set of possible relationships between entities is diverse (as shown in Fig. 2), we currently utilize embedding algorithms without estimating the weight of these relationships. However, as a future development, we may consider customized versions of the embedding algorithms that also take into account an estimation of the weights of the defined relationships. This would allow the algorithms to select random-walks and meta-paths used for embedding generation based on the weight of relationships, enabling more emphasis to be placed on relationships with higher weights during the computational process of the embedding.

In order to validate our approach, we performed an experiment where we studied the effectiveness and the performance of the proposed community detection techniques by considering different instances of DSNs and different types of graph embedding. The experiment involved data extracted from GitHub, one of the most used and popular software repository ecosystem, from which we were able to construct a heterogeneous graph synthesizing relevant relationships between Developers and other entities, besides the commit relationship.

This approach has shown his capability in producing a clear division in cohesive communities.

As we model the data in our original dataset, where we considered three different kinds of nodes (i.e. Developers, Issues and Repositories) we are conscious that there are various kinds of relationships and entities that can be found inside a software ecosystem.

Because GitHub is a very rich source of information, in our future works we will focus on using all the heterogeneous information that we can gather from it and try to identify different types of communities. Furthermore, we plan to apply our model also on different software repositories and to show how the proposed community detection approach can be very useful for more specific analytic tasks such as (i) influence analysis to discover the most important developers within a given community, (ii) team formation to detect the group of developers w.r.t. a given technology within several communities in order to start more quickly a new project in according to agile paradigm, (iii) recommendation to automatically suggest the most appropriate developers to solve an issue, etc.

CRediT authorship contribution statement

Marco De Luca: Conceptualization, Methodology, Validation, Formal analysis, Writing – original draft, Writing – review & editing. **Anna Rita Fasolino:** Conceptualization, Methodology, Validation, Formal analysis, Writing – original draft, Writing – review & editing. **Antonino Ferraro:** Conceptualization, Methodology, Validation, Formal analysis, Writing – original draft, Writing – review & editing. **Vincenzo Moscato:** Conceptualization, Methodology, Validation, Formal analysis, Writing – original draft, Writing – review & editing. **Giancarlo Sperlí:** Conceptualization, Methodology, Validation, Formal analysis, Writing – original draft, Writing – review & editing. **Porfirio Tramontana:** Conceptualization, Methodology, Validation, Formal analysis, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

References

Avelino, G., Passos, L., Hora, A., & Valente, M. T. (2016). A novel approach for estimating truck factors. In *2016 IEEE 24th international conference on program comprehension* (pp. 1–10). http://dx.doi.org/10.1109/ICPC.2016.7503718.

Batista, N. A., Brandão, M. A., Alves, G. B., da Silva, A. P. C., & Moro, M. M. (2017). Collaboration strength metrics and analyses on GitHub. In *Proceedings of the international conference on web intelligence* (pp. 170–178). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/3106426.3106480.

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *35*(8), 1798–1828. http://dx.doi.org/10.1109/TPAMI.2013.50.

Bhattacharjee, A., Nath, S. S., Zhou, S., Chakroborti, D., Roy, B., Roy, C. K., & Schneider, K. (2020). An exploratory study to find motives behind cross-platform forks from software heritage dataset. In *Proceedings of the 17th international conference on mining software repositories* (pp. 11–15). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/3379597.3387512.

Boldi, P., Pietri, A., Vigna, S., & Zacchiroli, S. (2020). Ultra-large-scale repository analysis via graph compression. In *2020 IEEE 27th international conference on software analysis, evolution and reengineering* (pp. 184–194). http://dx.doi.org/10.1109/SANER48275.2020.9054827.

Cai, H., Zheng, V. W., & Chang, K. C.-C. (2018). A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, *30*(9), 1616–1637. http://dx.doi.org/10.1109/TKDE.2018.2807452.

Cai, X., Zhu, J., Shen, B., & Chen, Y. (2016). GRETA: Graph-based tag assignment for GitHub repositories. In *2016 IEEE 40th annual computer software and applications conference, volume 1* (pp. 63–72). http://dx.doi.org/10.1109/COMPSAC.2016.124.

Chakraborty, T., Dalmia, A., Mukherjee, A., & Ganguly, N. (2017). Metrics for community analysis: A survey. *ACM Computing Surveys*, *50*(4), http://dx.doi.org/10.1145/3091106.

Chen, Y., Wu, J., & Wu, Z. (2022). China's commercial bank stock price prediction using a novel K-means-LSTM hybrid approach. *Expert Systems with Applications*, *202*, Article 117370. http://dx.doi.org/10.1016/j.eswa.2022.117370.

Dabbish, L., Stuart, C., Tsay, J., & Herbsleb, J. (2012). Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work* (pp. 1277–1286). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/2145204.2145396.

Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P., & Rubei, R. (2020). TopFilter: An approach to recommend relevant GitHub topics. In *Proceedings of the 14th ACM / IEEE international symposium on empirical software engineering and measurement*. New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/3382494.3410690.

Fu, T.-y., Lee, W.-C., & Lei, Z. (2017). HIN2Vec: Explore meta-paths in heterogeneous information networks for representation learning. (pp. 1797–1806). http://dx.doi.org/10.1145/3132847.3132953.

Fu, E., Zhuang, Y., Zhang, J., Zhang, J., & Chen, Y. (2021). Understanding the user interactions on github: A social network perspective. In *2021 IEEE 24th international conference on computer supported cooperative work in design* (pp. 1148–1153). IEEE, http://dx.doi.org/10.1109/CSCWD49262.2021.9437744.

Geiger, F.-X., Malavolta, I., Pascarella, L., Palomba, F., Di Nucci, D., & Bacchelli, A. (2018). A graph-based dataset of commit history of real-world android apps. In *Proceedings of the 15th international conference on mining software repositories* (pp. 30–33). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/3196398.3196460.

Gousios, G. (2013). The GHTorent dataset and tool suite. In *Proceedings of the 10th working conference on mining software repositories* (pp. 233–236). IEEE Press.

Gousios, G., & Spinellis, D. (2012). GHTorrent: Github's data from a firehose. In *2012 9th IEEE working conference on mining software repositories* (pp. 12–21). http://dx.doi.org/10.1109/MSR.2012.6224294.

Hajiakhoond Bidoki, N., & Sukthankar, G. (2018). Network semantic segmentation with application to GitHub. In *2018 International conference on computational science and computational intelligence* (pp. 1281–1284). http://dx.doi.org/10.1109/CSCI46756.2018.00247.

Herbold, S., Amirfallah, A., Trautsch, F., & Grabowski, J. (2021). A systematic mapping study of developer social network research. *Journal of Systems and Software*, *171*, Article 110802. http://dx.doi.org/10.1016/j.jss.2020.110802, URL https://www.sciencedirect.com/science/article/pii/S0164121220302077.

Hou, T., Yao, X., & Gong, D. (2020). Community detection in software ecosystem by comprehensively evaluating developer cooperation intensity. *Information and Software Technology*, *130*, Article 106451. http://dx.doi.org/10.1016/j.infsof.2020.106451.

Hou, T., Yao, X., & Gong, D. (2021). Community detection in software ecosystem by comprehensively evaluating developer cooperation intensity. *Information and Software Technology*, *130*, Article 106451. http://dx.doi.org/10.1016/j.infsof.2020.106451, URL https://www.sciencedirect.com/science/article/pii/S0950584920302007.

Hu, Y., Zhang, J., Bai, X., Yu, S., & Yang, Z. (2016). Influence analysis of Github repositories. *SpringerPlus*, *5*(1), 1268. http://dx.doi.org/10.1186/s40064-016-2897-7, https://app.dimensions.ai/details/publication/pub.1014395175 and https://springerplus.springeropen.com/track/pdf/10.1186/s40064-016-2897-7.

Jarczyk, O., Gruszka, B., Bukowski, L., & Wierzbicki, A. (2014). On the effectiveness of emergent task allocation of virtual programmer teams. In *WI-IAT '14*, *Proceedings of the 2014 IEEE/WIC/ACM international joint conferences on web intelligence (WI) and intelligent agent technologies (IAT) - volume 01* (pp. 369–376). USA: IEEE Computer Society, http://dx.doi.org/10.1109/WI-IAT.2014.58.

Ji, H., Wang, X., Shi, C., Wang, B., & Yu, P. (2021). Heterogeneous graph propagation network. *IEEE Transactions on Knowledge and Data Engineering*, 1. http://dx.doi.org/10.1109/TKDE.2021.3079239.

Jiang, J., Zhang, L., & Li, L. (2013). Understanding project dissemination on a social coding site. In *2013 20th working conference on reverse engineering* (pp. 132–141). http://dx.doi.org/10.1109/WCRE.2013.6671288.

Leibzon, W. (2016). Social network of software development at GitHub. In *2016 IEEE/ACM international conference on advances in social networks analysis and mining* (pp. 1374–1376). http://dx.doi.org/10.1109/ASONAM.2016.7752419.

Luo, Z., Mao, X., & Li, A. (2015). An exploratory research of GitHub based on graph model. In *2015 Ninth international conference on frontier of computer science and technology* (pp. 96–103). http://dx.doi.org/10.1109/FCST.2015.45.

Marlow, J., Dabbish, L., & Herbsleb, J. (2013). Impression formation in online peer production: Activity traces and personal profiles in github. In *Proceedings of the 2013 conference on computer supported cooperative work* (pp. 117–128). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/2441776.2441792.

Matragkas, N., Williams, J. R., Kolovos, D. S., & Paige, R. F. (2014). Analysing the 'biodiversity' of open source ecosystems: The GitHub case. In *MSR 2014*, *Proceedings of the 11th working conference on mining software repositories* (pp. 356–359). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/2597073.2597119.

Minh, H.-L., Sang-To, T., Abdel Wahab, M., & Cuong-Le, T. (2022). A new metaheuristic optimization based on K-means clustering algorithm and its application to structural damage identification. *Knowledge-Based Systems*, *251*, Article 109189. http://dx.doi.org/10.1016/j.knosys.2022.109189.

Mockus, A., Spinellis, D., Kotti, Z., & Dusing, G. J. (2020). A complete set of related git repositories identified via community detection approaches based on shared commits. In *Proceedings of the 17th international conference on mining software repositories* (pp. 513–517). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/3379597.3387499.

Moradi-Jamei, B., Kramer, B. L., Calderón, J. B. S., & Korkmaz, G. (2022). Community formation and detection on GitHub collaboration networks. In *Proceedings of the 2021 IEEE/ACM international conference on advances in social networks analysis and mining* (pp. 244–251). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/3487351.3488278.

Moscato, V., & Sperlì, G. (2021). A survey about community detection over online social and heterogeneous information networks. *Knowledge-Based Systems*, *224*, Article 107112. http://dx.doi.org/10.1016/j.knosys.2021.107112, URL https://www.sciencedirect.com/science/article/pii/S0950705121003750.

Newman, M. (2016). Equivalence between modularity optimization and maximum likelihood methods for community detection. *Physical Review E*, *94*, http://dx.doi.org/10.1103/PhysRevE.94.052315.

Nguyen, P., Di Rocco, J., Rubei, R., & Di Ruscio, D. (2020). An automated approach to assess the similarity of GitHub repositories. *Software Quality Journal*, *28*(2), 595–631. http://dx.doi.org/10.1007/s11219-019-09483-0, cited By 2.

Pham, D. T., Dimov, S. S., & Nguyen, C. D. (2005). Selection of K in K-means clustering. *Proceedings of the Institution of Mechanical Engineers, Part C (Mechanical Engineering Science)*, *219*(1), 103–119.

Pietri, A., Rousseau, G., & Zacchiroli, S. (2020). Determining the intrinsic structure of public software development history. In *Proceedings of the 17th international conference on mining software repositories* (pp. 602–605). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/3379597.3387506.

Pinto, G., Steinmacher, I., & Gerosa, M. A. (2016). More common than you think: An in-depth study of casual contributors. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering, vol. 1* (pp. 112–123). http://dx.doi.org/10.1109/SANER.2016.68.

Rahman, M. M., & Roy, C. K. (2014). An insight into the pull requests of GitHub. In *MSR 2014*, *Proceedings of the 11th working conference on mining software repositories* (pp. 364–367). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/2597073.2597121.

Saxena, A., Prasad, M., Gupta, A., Bharill, N., Patel, O. P., Tiwari, A., Er, M. J., Ding, W., & Lin, C.-T. (2017). A review of clustering techniques and developments. *Neurocomputing*, *267*, 664–681. http://dx.doi.org/10.1016/j.neucom.2017.06.053, URL https://www.sciencedirect.com/science/article/pii/S0925231217311815.

Schettino, V., Horta, V., Araújo, M. A. P., & Ströele, V. (2019). Towards community and expert detection in open source global development. In *2019 IEEE 23rd international conference on computer supported cooperative work in design* (pp. 350–355). http://dx.doi.org/10.1109/CSCWD.2019.8791872.

Schlichtkrull, M. S., Kipf, T. N., Bloem, P., van den Berg, R., Titov, I., & Welling, M. (2018). Modeling relational data with graph convolutional networks. In *ESWC* (pp. 593–607). http://dx.doi.org/10.1007/978-3-319-93417-4_38.

Seifer, P., Härtel, J., Leinberger, M., Lämmel, R., & Staab, S. (2019). Empirical study on the usage of graph query languages in open source java projects. In *SLE 2019*, *Proceedings of the 12th ACM SIGPLAN international conference on software language engineering* (pp. 152–166). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/3357766.3359541.

Sheoran, J., Blincoe, K., Kalliamvakou, E., Damian, D., & Ell, J. (2014). Understanding "watchers" on GitHub. In *MSR 2014*, *Proceedings of the 11th working conference on mining software repositories* (pp. 336–339). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/2597073.2597114.

Sülün, E., Tüzün, E., & Doğrusöz, U. (2021). RSTrace+: Reviewer suggestion using software artifact traceability graphs. *Information and Software Technology*, *130*, Article 106455. http://dx.doi.org/10.1016/j.infsof.2020.106455, URL https://www.sciencedirect.com/science/article/pii/S0950584920300021.

Sülün, E., Tüzün, E., & Doğrusöz, U. (2019). Reviewer recommendation using software artifact traceability graphs. In *Proceedings of the fifteenth international conference on predictive models and data analytics in software engineering* (pp. 66–75). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/3345629.3345637.

Thanapalasingam, T., Berkel, L., Bloem, P., & Groth, P. (2021). Relational graph convolutional networks: A closer look.

Trouillon, T., Dance, C. R., Gaussier, É., Welbl, J., Riedel, S., & Bouchard, G. (2017). Knowledge graph completion via complex tensor factorization. *Journal of Machine Learning Research*, *18*, 130:1–130:38.

Trouillon, T., Welbl, J., Riedel, S., Gaussier, E., & Bouchard, G. (2016). Complex embeddings for simple link prediction.

Wang, X., Bo, D., Shi, C., Fan, S., Ye, Y., & Philip, S. Y. (2022a). A survey on heterogeneous graph embedding: methods, techniques, applications and sources. *IEEE Transactions on Big Data*.

Wang, X., Bo, D., Shi, C., Fan, S., Ye, Y., & Yu, P. S. (2022b). A survey on heterogeneous graph embedding: Methods, techniques, applications and sources. *IEEE Transactions on Big Data*, 1. http://dx.doi.org/10.1109/TBDATA.2022.3177455.

Wang, D., Cao, J., Qian, S., & Qi, Q. (2019). Investigating cross-repository socially connected teams on GitHub. In *2019 26th Asia-Pacific software engineering conference* (pp. 490–497). http://dx.doi.org/10.1109/APSEC48747.2019.00072.

Wang, Z., & Perry, D. E. (2015). Role distribution and transformation in open source software project teams. In *2015 Asia-Pacific software engineering conference* (pp. 119–126). http://dx.doi.org/10.1109/APSEC.2015.12.

Xie, Y., Yu, B., Lv, S., Zhang, C., Wang, G., & Gong, M. (2021). A survey on heterogeneous network representation learning. *Pattern Recognition*, *116*, Article 107936. http://dx.doi.org/10.1016/j.patcog.2021.107936.

Yamashita, K., McIntosh, S., Kamei, Y., Hassan, A. E., & Ubayashi, N. (2015). Revisiting the applicability of the Pareto principle to core development teams in open source software projects. In *IWPSE 2015*, *Proceedings of the 14th international workshop on principles of software evolution* (pp. 46–55). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/2804360.2804366.

Yang, C., Xiao, Y., Zhang, Y., Sun, Y., & Han, J. (2020). Heterogeneous network representation learning: A unified framework with survey and benchmark. *IEEE Transactions on Knowledge and Data Engineering*, 1. http://dx.doi.org/10.1109/TKDE.2020.3045924.

Zhang, Y., Lo, D., Kochhar, P. S., Xia, X., Li, Q., & Sun, J. (2017). Detecting similar repositories on GitHub. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering* (pp. 13–23). http://dx.doi.org/10.1109/SANER.2017.7884605.

Zihayat, M., Kargar, M., & An, A. (2014). Two-phase Pareto set discovery for team formation in social networks. In *WI-IAT '14*, *Proceedings of the 2014 IEEE/WIC/ACM international joint conferences on web intelligence (WI) and intelligent agent technologies (IAT) - volume 02* (pp. 304–311). USA: IEEE Computer Society, http://dx.doi.org/10.1109/WI-IAT.2014.112.