# BDM Socio - Technical Optimizer

Boi Biagio
b.boi@studenti.unisa.it
Universitá degli studi di Salerno
Salerno, Italy

Silvio Di Martino
s.dimartino21@studenti.unisa.it
Universitá degli studi di Salerno
Salerno, Italy

Gerardo Martino
g.martino11@studenti.unisa.it
Universitá degli studi di Salerno
Salerno, Italy

## ABSTRACT

Recent studies have proven that software development is a social activity, in fact it is possible to measure the level of coordination between developers and the organization in order to improve the quality of software product in various respects. Software Products are constantly evolving, so maintenance activities play a crucial role in enabling you to have a useful product. With the support of tools and Refactoring technique, we can improve the source code automatically, but one of the relevant key factor that at moment these software do not perform is how we can improve the social-technical aspect, meaning improve a restructuring of the developer community and team rearrangement to improve a software product. The study starts with a repository mining activity, and then builds the graphs useful to represent the communication among developers (dev-dev), the collaboration among developers (dev-file) and the dependencies among files (file-file). After that by using the Genetic Algorithms we have processed the information collected in order to optimize the relationships between the members of the team [1].

## KEYWORDS

mining, optimization, refactoring, genetic algorithms

## 1 INTRODUCTION

In detail the first step that we did was searching for tools and libraries to get the necessary information. In particular, to mine the information from communications between developers we used the *GitHub API* to get data on issues (comments and answers); while regarding information related to commit effettuati from developers on source files, through the framework *PyDriller*, we have easily extracted information from any Git repository, such as commits, developers, modifications, diffs, and source codes, and quickly export as CSV file, the mining of information ended with the study of the relationship between files, in particular the structural code dependency analysis was carried out with *Depends*, a source code dependency extraction tool, designed to infer syntactical relations among source code entities, such as files and methods, from various programming languages.

The next step was the representation of this information by means of adjacent lists, that through the framework *NetworkX*, we have been easily transformed into graphic elements (graphs). Once the three individual graphs have been obtained, they were subsequently merged to have all the communication, collaboration and coupling information represented by one single structure.

Finally the last step consists of the execution of a Genetic Algorithm (GA) through the framework *Deap*, able to find and optimize a version of the starting graph given as input, taking into account the objective functions to be optimised [1].

## 2 BACKGROUND

### 2.1 Dev-Dev Implementation

To obtain data regarding the developer, methods have been implemented through the APIs made available by GitHub, that allow us to extrapolate various information from the repositories, including the list of developers who contributed to the development and the number of responses of those to the questions. The extracted output allowed us to build a square matrix, on whose rows and columns the developers find space. The element *(i, j)* will be an integer representing the number of responses to the same thread between developer i and developer j. Here is an example matrix:

| | dev | | |
|---|---|---|---|
| dev | 2 | 2 | 2 |
| | 2 | 2 | 2 |
| | 2 | 2 | 2 |
| | 2 | 2 | 2 |

**Figure 1: Example of Dev-Dev matrix.**

After the construction of the above matrix and through the use of the *NetworkX*, a Python tool for the creation, manipulation, and study of data structures, we obtained an undirected weighted graph in which the nodes represent the developers and the edges the communications among them, as shown in Figure 2. Meaning that if a developer participated in at least one issue thread with another developer, an edge is created to connect them. The weights are represented by the total number of replies given in a shared thread over all the threads of the project, without considering the criticality of the issue.

### 2.2 Dev-File Implementation

Then through the use of the *PyDriller* library, it was possible to analyze the repository and extrapolate information on the commits. In particular, methods have been implemented to obtain information on the number of commits made by each developer on each file. Another matrix was then built, on whose rows we find the developers and on whose columns we find the files: element *(i, j)* will be an integer representing the number of commits that the developer i has made on file j. Here is an example matrix:

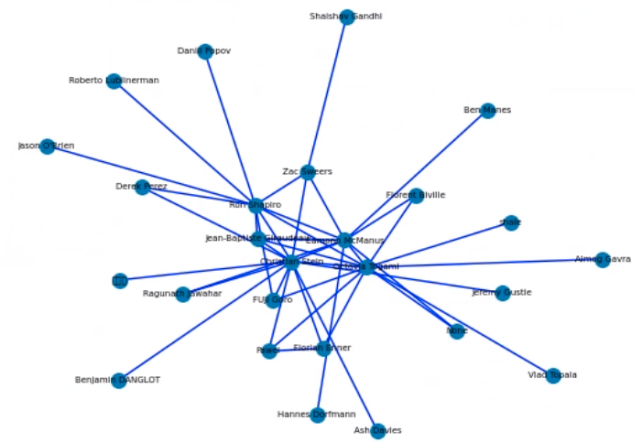

**Figure 2: dev-dev graph**

| | file | | |
|---|---|---|---|
| dev | 1 | 1 | 1 |
| | 1 | 1 | 1 |
| | 1 | 1 | 1 |
| | 1 | 1 | 1 |

**Figure 3: Example of Dev-File matrix.**

Also in this case, through *NetworkX* , a bipartite graph has been obtained as shown in Figure 4, as we have only two types of nodes, developer and file, connected. A developer node is connected to a file node if the developer has committed at least once that file. The weight on the edges consists of the developer's total number of commits on that file in the considered.

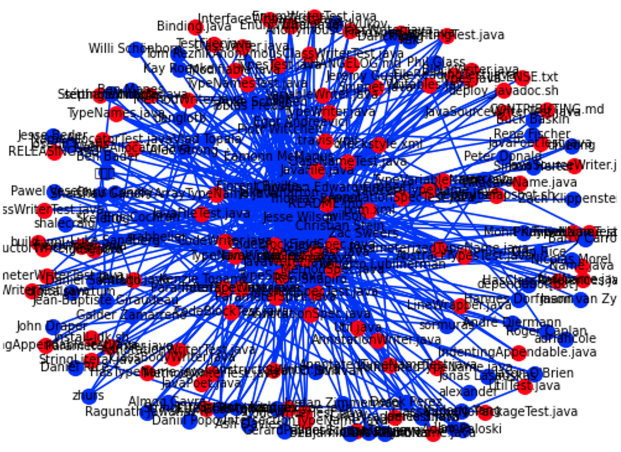

**Figure 4: dev-file graph**

## 2.3 File-File Implementation

Finally, to automate the dependency extraction process we invoked the Depends framework in order to obtain information regarding the dependencies between the various source files.

The framework *Depends* supports major dependency types, including:

- `Call`: function/method invoke
- `Cast`: type cast
- `Contain`: variable/field definition
- `Create`: create an instance of a certain type
- `Extend`: parent-child relation
- `Implement`: implemented interface
- `Import/Include`: for example, java import, c/c++ import, ruby require
- `Parameter`: as a parameter of a method
- `Return`: returned type
- `Throw`: throw exceptions
- `Use`: use or set variables
- `ImplLink`: the implementation link between call and the implementation of prototype

The extracted data led to the construction of a third matrix, on whose rows and columns we find the files: the element *(i, j)* corresponds to the total number of dependencies between the files *i* and *j* (in terms of the sum of the different types of dependencies) as shown in Figure 3.

| | file | | |
|---|---|---|---|
| file | 0 | 0 | 0 |
| | 0 | 0 | 0 |
| | 0 | 0 | 0 |
| | 0 | 0 | 0 |

**Figure 5: Example of File-File matrix.**

Using *NetworkX*, the file-file graph has been obtained, as shown in Figure 6, in which a file node is connected with another if it contains a reference to a code component contained in another file or vice versa. The obtained graph is then undirected, and the weights on the edges represent the actual number of references present among the two source files.

At the end of the extraction activity, we proceed with the creation of a single matrix, which consists of a merge of the above 3 matrices obtained. Below, we show an example of a resulting matrix, obtained by combining the 3 exemplary matrices as shown in Figure 4.

It is necessary to emphasize the fact that in the merge it was necessary to transpose one of the three matrices *(DEV-FILE –> FILE-DEV)* in order to maintain consistency and coherence in the structure.

# 3 GENETIC ALGORITHM EXECUTION

In this phase we use the framework *Deap* for rapid prototyping and testing of *Genetic Algorithm* techniques. So we initialized the population in which each individual is represented by an adjacency

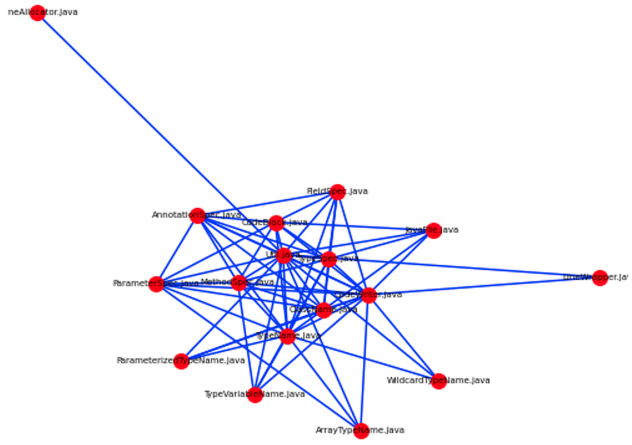


**Figure 6: file-file graph**

| | *file* | | | *dev* | | |
|---|---|---|---|---|---|---|
| ***file*** | 0 | 0 | 0 | 1 | 1 | 1 |
| | 0 | 0 | 0 | 1 | 1 | 1 |
| | 0 | 0 | 0 | 1 | 1 | 1 |
| | 0 | 0 | 0 | 1 | 1 | 1 |
| ***dev*** | 1 | 1 | 1 | 2 | 2 | 2 |
| | 1 | 1 | 1 | 2 | 2 | 2 |
| | 1 | 1 | 1 | 2 | 2 | 2 |
| | 1 | 1 | 1 | 2 | 2 | 2 |

**Figure 7: Combination of the three matrix**

matrix that is structured in the same manner of the starting matrix and pseudo-randomly value.

The keypoint is the definition of a multi-objective fitness function: the main goal is *maximizing* the communication and the collaboration between developers, *minimizing* the coupling among components. Besides these three objectives, it is also essential to minimize the changes from the starting graph to the output graph to prevent the GA from excessively altering the original structure while optimizing the objectives. As suggested by Palomba et al. [1], this equation reports the fitness function, to have a better comprehensibility we refer to different parts of the graph with different names, i.e., $G_{dev-dev}$, $G_{dev-file}$, and $G_{file-file}$.

$$f(G)=\begin{cases} max(\sum com_{d_i,d_j})\forall(d_i,d_j) \in G_{dev-dev} \\ max(\sum min(col_{d_i,f}, col_{d_j,f})\forall(u,v) \in G_{dev-file} \\ min(\sum cou_{f_i,f_j})\forall(u,v) \in G_{file-file} \\ min(dist(G_s,G)) \end{cases}$$

where $com_{d_i,d_j}$ is the degree of communication between developers $d_i$ and $d_j$ in the dev-dev graph, max($col_{u_i;v}$; $col_{u_j;v}$) represents the number of co-commits of developers $d_i$ and $d_j$ on the file $f$, $cov_{f_i,f_j}$ is the degree of coupling between files $f_i$ and $f_j$ , and dist($G_s$, G) is the minimum number of operations to transform graph $G_s$ to graph G.

In order to implement the first two objective functions, the basic idea is to compute a value based on these two values:

- `'x'`: the total number of commits on a certain file
- `'y'`: the summation of the number of replies on a given thread by the developers with at least one commit on that file

In other words, we want to compute the degree of communication for each file: so, given those 2 values, we calculate the ratio between '$y$' and '$x$'. The goal is to *maximize* this value, given the assumption that a high number of replies means good teamwork.

The third objective function has been implemented by the computation of the summation of the total number of dependencies: the goal, clearly, is to *minimize* it.

The fourth objective function has been implemented by the computation of the summation of the differences between the previous values of the *file-file* matrix and the new elements computed by the mutation operator. Our goal is to *minimize* this summation, in order to prevent the *Genetic Algorithm* from excessively altering the original structure.

Since we have a particular structure with some constraint we cannot use the standard mutation functions; so it's been necessary to declare and use our customized function, that will take a mutant with a probability of 20%, and for each element inside the mutant (for each row in the matrix) will change the value in a range from 0 to the mean of the value plus a value of 10 with a probability based on the number of changes already done. In this way, the mutation function will impact relatively on our graph.

For the crossover function we haven't done any kind of customization since we simply select childs on a probability of 50%. The choice fell on the 'CxOnePoint' function that executes a one point crossover on the input sequence individuals. The two individuals are modified in place. The resulting individuals will respectively have the length of the other. With this strategy we guarantee a good randomness but without modifying the entire structure; preventing in this way, the increase of modifications.

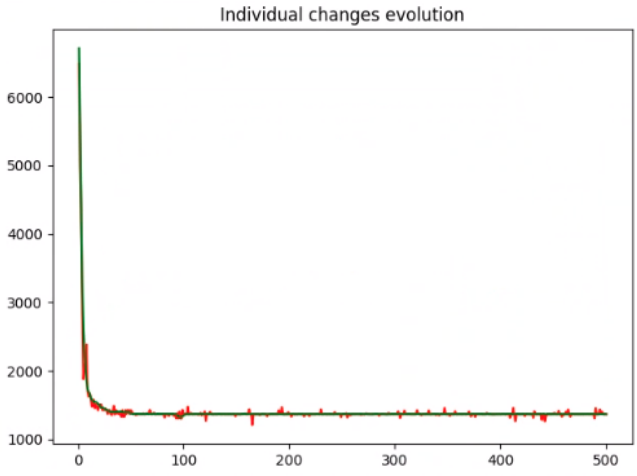


**Figure 8: Individual change evolution**

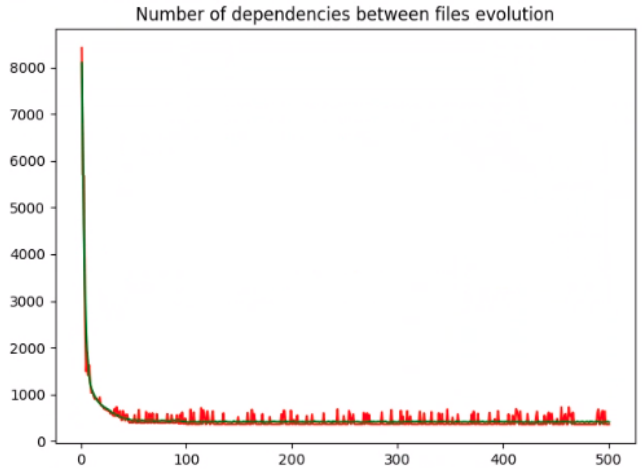


**Figure 9: Number of dependencies between files evolution**

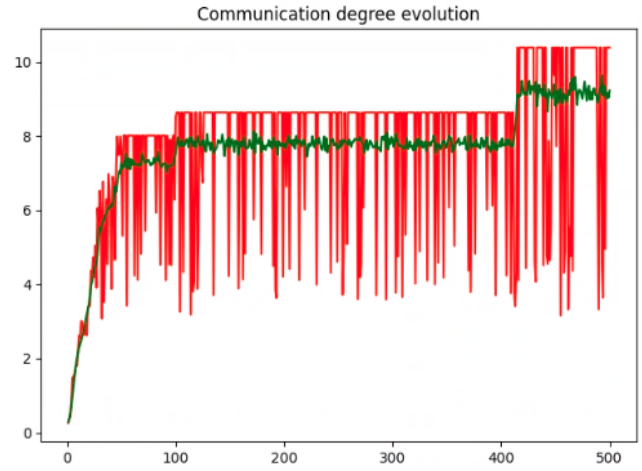


**Figure 10: Communication degree evolution**

## 4 CONCLUSION

Indeed, as for the following figures, it is possible to see how the summation of the dependencies and the number of changes of the best individual in the population decrease during the evolution, as shown in Figure (8,9).

On the contrary, communication's degree correctly increases during evolution, but contrary from the other graphics, in our opinion, it doesn't evolve similarly due to the poor amount of data that we have, as shown in Figure 10. These are the expected and wanted results, since the goals have been achieved.

Finally, let's compare the real result of this evolution: the final population has output a new file-file matrix, and this is the resultant graph. Considering the *NetworkX* functionalities, the higher are the weights, the more clustered are the nodes and so the shorter are the edges. On the contrary, as the second graph tells, the nodes are more spaced since lower are the weights, as shown in Figure 11.

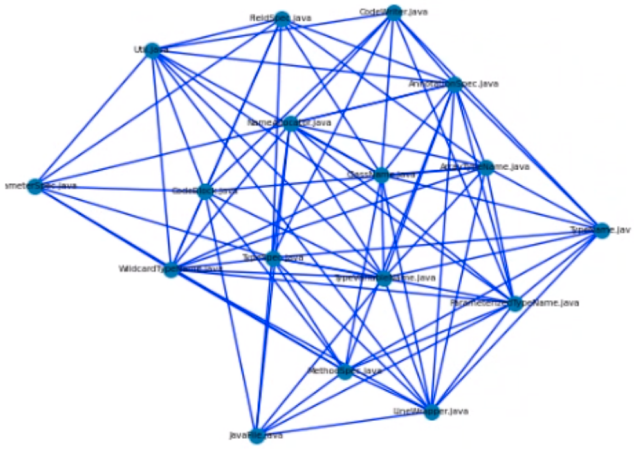

**Figure 11: New File-File Graph**

This work made it possible to confirm the existence of a relationship between the communication between developers and the dependencies between files. The genetic algorithm was necessary in order to identify a strategy to optimize these relationships through the defined objective functions. A possible future development of the project is the improvement of these functions in order to better identify what are the factors that affect socio-technical congruence.

## REFERENCES

[1] Manuel De Stefano, Fabiano Pecorelli, Damian Andrew Tamburri, Fabio Palomba, and Andrea De Lucia. [n.d.]. Refactoring Recommendations Based on the Optimization of Socio-Technical Congruence. ([n. d.]).

## A ONLINE RESOURCES

- `Project Repository`: https://github.com/biagioboi/socio-technical-congruence-optimizer.git
- `Github API`: https://docs.github.com/en
- `PyDriller`: https://pydriller.readthedocs.io/en/latest/
- `Depends`: https://github.com/multilang-depends/depends.git
- `NetworkX`: https://networkx.org/
- `Deap`: https://github.com/deap/deap