The use of random graph theory to assess the quality of sequential source code check-ins

Mahir Arzoky[1], Stephen Swift[1], Steve Counsell[1]
and James Cain[2]

Brunel University, Middlesex, UK
{mahir.arzoky, stephen.swift, steve.counsell}@brunel.ac.uk

Quantel Limited, Newbury, UK
james.cain@quantel.com

Abstract. Software module clustering is the problem of automatically partitioning the structure of a software system using low-level dependencies in the source code to understand and improve the system's architecture. Munch, a clustering tool based on search-based software engineering techniques, was used to modularise a unique dataset of sequential source code software versions. This paper investigates whether the dataset used for the modularisation resembles a random graph by computing the probabilities of observing certain connectivity. Modularisation will not be possible with data that resembles random graphs. Thus, this paper demonstrates that our real world time-series dataset does not resemble a random graph except for small sections where there were large maintenance activities. Furthermore, the random graph metric can be used as a tool to indicate areas of interest in the dataset, without the need to run the modularisation.

Keywords: software module clustering; modularisation; SBSE; random graph; time-series; fitness function

1 Introduction

Large software systems tend to have complex structures that are often difficult to comprehend due to the large number of modules (classes) and inter-relationships that exist between them. As the modular structure of a software system tends to decay over time, it is important to modularise. Modularisation is the process of partitioning the structure of software system into subsystems. It makes the problem at hand easier to understand, as it reduces the amount of data needed by developers [7]. Subsystems group together related source-level components to assist with system's understandability. Subsystems can be organised hierarchically to allow developers to navigate through the system at various levels of details, they include resources such as modules, classes and other subsystems [7].

Graphs can be used to make the software structure of complex systems more comprehensible [17]. They can be described as language-independent, whereby compo-

nents such as classes or subroutines of a system are represented as nodes and the inter-relationships between the components are represented as edges. Such graphs are referred to as Module Dependency Graph (MDG). Creating an MDG of the system does not always make it easy to understand the system's structure; graphs could be partitioned to make them more accessible and easier to comprehend. Mancoridis et al [14] were the first to use MDG as a representation of the software module clustering problem. There have been a large number of studies [8] [11] [12] [18] using the search-based software engineering approach to solve the software module-clustering problem. In previous studies, techniques that treat clustering as an optimisation problem were introduced. A number of various heuristic search techniques, including Hill Climbing were used to explore the large solution space of all possible partitions of an MDG.

This paper performs modularisation on source code check-ins (commits), taking advantage of the fact that the dataset is a time-series. The nearer the source codes in time, the more similar they are expected to be. The aim is to use code structure and sequence to obtain more effective modularisation and also to locate the possible occurrence of major changes, in particular refactorings. We look to verify the quality of the results of the modularisation by finding out whether the time-series dataset resembles a random graph. There are currently few studies [4] [16] [19] that use random graph theories for source code analysis. For this paper, we aim to calculate the probabilities of the graphs (software versions) resembling a random graph for the whole dataset. We look to investigate whether the probabilities increase as the maintenance increases and whether the architecture resembles more randomness throughout the life of the project.

The paper is organised as follows: Section 2 describes the clustering algorithms and fitness functions. Section 3 describes the creation and pre-processing of the source data and the experiment. Section 4 discusses the results and Section 5 draws conclusions and outlines future work.

2 Experimental Methods

2.1 Clustering Algorithm

This work extends that of Arzoky et al [2] [3] and, follows Mancoridis et al and Mitchell [14] [17], who first introduced search-based approach to software modularisation. The clustering algorithm was re-implemented from available literature on Bunch's clustering algorithm [18] to form a tool called Munch. Munch is a prototype implemented to carry out experimentations of different heuristic search approaches and fitness functions. Munch uses an MDG as an input and produces a hierarchical decomposition of the system structure as an output. Closely related modules are grouped into clusters that are loosely connected to other clusters. A cluster is a set of the modules in each partition of the clustering.

The clustering algorithm uses a simple random mutation Hill Climbing approach to guide the search. It is a simple, easy to implement technique that has proven to be

useful and robust in terms of modularisation [18]. The aim is to produce a graph partition that minimises coupling between clusters and maximises cohesion within each cluster. Coupling is defined as the degree of dependence between different modules or classes in a system, whereas cohesion is the internal strength of a module or class [7].

2.2 Fitness Function

A fitness function is used to measure the relative quality of the decomposed structure of system into subsystems (clusters). Previously, we experimented with several fitness functions: the Modularisation Quality (MQ) metric of Mancoridis et al [14], and the EValuation Metric (EVM) of Tucker et al [20].

EVM rewards maximising the cohesiveness of the clusters, clustering with a high number of intra-module relationships, but it does not directly penalise inter-clustering coupling. It searches for all possible relationships within a cluster and rewards those that exist within the MDG and penalises those that does not exist within the MDG [14]. For the following formal definition of EVM, a clustering arrangement C of n items is defined as a set of sets $\{c_1, \ldots, c_m\}$, where each set (cluster) $c_i \subseteq \{1,\ldots,n\}$ such that $c_i \neq \phi$ and $c_i \cap c_j = \phi$ for all $i \neq j$. Note that $1 \leq m \leq n$ and $n > 0$. Note also that $\bigcup_{i=1}^{m} c_i = \{1,...,n\}$. Let MDG M be an n by n matrix, where a '1' at row i and column j (M_{ij}) indicates a relationship between variable i and j, and '0' indicates that there is no relationship. Let c_{ij} refer to the j^{th} element of the i^{th} cluster of C. The score for cluster c_i is defined in Equation 2.

$$EVM(C,M) = \sum_{i=1}^{m} h(c_i, M) \tag{1}$$

$$h(c_i, M) = \begin{cases} \sum_{a=1}^{|c_i|-1} \sum_{b=a+1}^{|c_i|} L(c_{ia}, c_{ib}) & ,\text{if } |c_i| > 1 \\ 0 & ,\text{Otherwise} \end{cases} \tag{2}$$

$$L(v_1, v_2, M) = \begin{cases} 0 & ,v_1 = v_2 \\ +1 & ,M_{v_1 v_2} + M_{v_2 v_1} > 0 \\ -1 & ,\text{Otherwise} \end{cases} \tag{3}$$

To speed up the process of the modularisation, we introduced EValuation Metric Difference (EVMD), a faster version of the EVM function. EVMD was selected as the fitness function for the modularisations as it is more robust than MQ and faster than EVM [2]. It utilises an update formula on the assumption that one small change is being made between clusters. It is a faster way of evaluating EVM, where the previous fitness is known and the current fitness is calculated, without having to do the

move. It produces the same results as EVM, but effectively reduces the computational operations from O($n\sqrt{n}$) to O($\sqrt{n}$).

For the formal definition of EVMD, let f_{old} be the EVM fitness function. Also, let x be the *from* cluster, y be the *to* cluster and z be the *index*. Function G, defined in Equation 5, determines the relationship (from MDG M) that exists between variable v and cluster k. Equation 3 simply checks whether it is a positive or negative influence (i.e. does a relationship exist?).

$$EVMD(f_{old}, C, x, y, z, M) = f_{old} - G(C_x, C_{xz}, M) + G(C_y, C_{xz}, M) \quad \textbf{(4)}$$

$$G(C_k, v, M) = \sum_{i=1}^{|C_k|} L(c_{ki}, v, M) \quad \textbf{(5)}$$

From this point forward EVM will be used when referring to the EVMD metric.

2.3 HS Metric

Homogeneity and Separation (HS) is an external coupling metric defined in [1] to measure the quality of the modularisation. HS is based on the Coupling Between Objects (CBO) metric, first introduced by Chidamber and Kemerer [6]. CBO (for a class) is defined as the count of the number of other classes to which it is coupled [6].

HS is a simple coupling metric that calculates the ratio of the proportion of internal and external edges. As shown in Equation 6, HS is calculated by subtracting the number of links within clusters from the number of links that are between clusters, and then dividing the answer by the total number of links (to normalise it). It searches through all the links within the MDG, finding all the pairs that are not equal to 0. If the two variables are in the same cluster, H is incremented, and if they are in different clusters, S is incremented. The more links between the clusters the worse the modularisation, as only internal links are modularised. A value of +1 is returned if all the links are within the modules, a value of −1 is returned if all links are external coupling, and 0 is produced if there is an equal number.

For the formal mathematical definition of the HS metric, we define a function $P(v,C)$ which returns the cluster number within C that variable (class) v resides.

$$HS(C, M) = \frac{H(C, M) - S(C, M)}{H(C, M) + S(C, M)} \quad \textbf{(6)}$$

$$H(C, M) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (1 - \delta(M_{ij}, 0)) \delta(P(i, M), P(j, M)) \quad \textbf{(7)}$$

$$S(C, M) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (1 - \delta(M_{ij}, 0)) (1 - \delta(P(i, M), P(j, M)) \quad \textbf{(8)}$$

We use Kronecher's Delta function $\delta(i,j)$, which is defined as follows:

$$\delta(i, j) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \quad \textbf{(9)}$$

2.4 Weighted-Kappa

Weighted-Kappa (WK) [1] is a simple statistical metric for the comparative assessment of two or more components. For this paper it is used for the comparison of two clustering arrangements. It rates the agreement between the classification decisions made by two or more observers (clustering methods). It not only measures similarity but also takes into account the degree of disagreements. The WK value ranges from −1.0 (no concordance) to 1.0 (complete concordance). A high WK value suggests that the two clustering arrangements are similar, whereas a low value suggests that they are dissimilar. A value of approximately 0 is normally observed for two random clusters.

3 Experiment

3.1 Data Creation

The large dataset used for this paper is from processed source code of an award winning product line architecture library, provided by Quantel Limited. The dataset consists of information on different versions of a software system over time. The dataset comprises of over 0.5 million lines of C++ code collected over the period 17/10/2000 to 03/02/2005, with 503 versions (check-ins) in total. There are roughly 2-3 days' gap between each check-in (corresponding to a graph), giving a total time span of 4 years and 4 months for the dataset [5].

Table 1. Class Relation Types

Class relationship	Description
Attributes	Data members in a class
Bases	Immediate base classes
Inners	Any type declared inside the scope of a class
Parameters	Parameters to member functions of a class
Returns	Return from member functions of a class

A total of 6120 classes exist in the system, however, not all classes exist at the same time slice; there are between 434 and 2272 of classes that exist at a particular point in time, referred to as "active" classes. Classes generally "appear" and "disappear" at various time points through the dataset. One reason for these occurrences is that when a class is renamed, it will appear in the dataset as a new class with a new

identifier. The dataset consists of five time-series of un-weighted graphs. For this paper, graphs of the five types of relationship were merged together to form the 'whole system' for particular time slices. Table 1 describes how each graph represents a relationship between classes.

Also, for this paper, the MDGs were significantly reduced; all modules that were not produced by Quantel and are not active at the time slice were removed. Classes not produced by Quantel include the Standard Template Library (STL), Windows COM Interface classes and components from a 3rd part library. This required a re-write of the Munch tool which has reduced the runtime of the modularisation process considerably. There are now between 202 and 1193 active classes at any one point. Fig 1 shows all of the active classes at each software check-in (graph), all of the graphs are ordered in time.

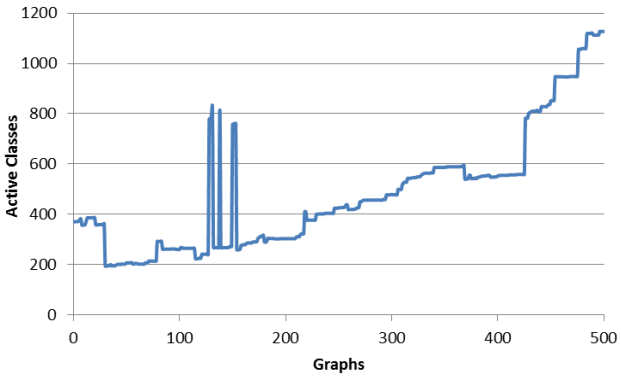

Fig. 1. Quantel's active classes at each software check-in

3.2 Absolute Value Difference (AVD)

Our previous work [2] [3] showed that there were no significant changes to the source code between two successive software versions. We produced a set of results showing the similarity between the graphs by subtracting every two successive binary matrices from each other's. Equation 10 shows how the AVD is calculated for each graph, where X and Y are two n by n binary matrices (MDGs). An AVD value of 0 indicates that two matrices are identical, whereas a large positive value indicates that they are different. A value between 0 and a large number gives a degree of similarity.

$$AVD(X,Y) = \sum_{i=1}^{n}\sum_{j=1}^{n}\left|X_{ij} - Y_{ij}\right| \quad \textbf{(10)}$$

Fig. 2 shows the AVDs of the full dataset of 503 graphs. The majority of the graphs have very low AVD, as there were only a few days of development between each check-in. In fact, 46 per cent of the graphs have an AVD of 0. However, sudden peaks and drops can be observed from the plot, which could indicate where major changes or refactoring activities occurred.

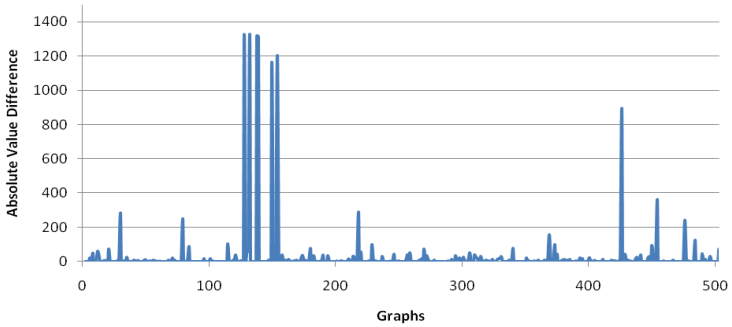

Fig. 2. Plot showing the AVDs of the full dataset

3.3 Experiment Procedure

For this paper, we have devised one experiment to modularise the full dataset of 503 graphs. The run time for each modularisation was 10 million iterations. The starting clustering arrangement consisted of every variable in its own cluster. It assumes that all classes are independent; there are no relationships. The experiment was repeated 25 times as Hill Climbing is a stochastic method and there is a risk of the search only reaching a local maximum.

3.4 Random Graph

A random graph can be modelled with a set of n nodes, adding edges between them at random. One of the most commonly studied random graph models is the one introduced by Erdős–Rényi [9] and Gilbert [10], denoted $G(n,p)$. An edge can occur independently with probability $0 < p < 1$. Edges are chosen randomly for a fixed set of n nodes and each edge is chosen to be added or removed from the graph with probability p. Thus, the expected number of edges can be calculated as in Equation 11, however, the number of edges can change randomly and all graphs have $p \neq 0$ of being selected.

$$E = p\frac{n(n-1)}{2} \tag{11}$$

We generated the expected distribution of edges based on the Erdős–Rényi random graph model. Subsequently, we created the observed distribution from each MDG. We used the binomial distribution to compute the probability of observing $1 \ldots n-1$ connectivity. p is calculated from the density and the density is calculated from the MDG. The density is simply calculated by dividing the number of edges by the total number of edges that there could have been. Lastly, we use the Kolmogorov-Smirnov test [15] to give us the probability that the two distributions are equivalent i.e. whether it is a random graph or not.

4 Results and Discussion

For each graph in the dataset we recorded the frequency of the number of edges. There will be no nodes that have 0 edges as everything is connected to each other. For this paper, all of the modules that are not produced by Quantel and all of the non-active classes were removed. For example, for Graph 1, there are 85 classes that are connected to only 1 class and there are 66 classes that are connected to 2 classes. Fig 3 shows the connectivity of Graph 105 for both the observed and the expected number of edges. It can be observed from the plot that there is a noticeable difference between observed and expected edges; this might be due to the high probability value (0.0343) of this graph resembling a random graph.

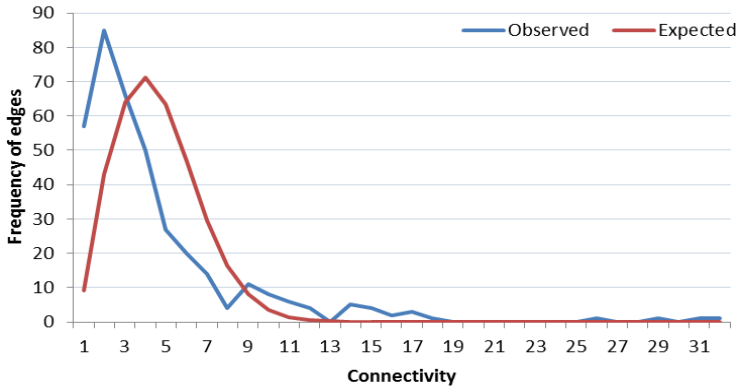

Fig. 3. Connectivity against the frequency of edges for Graph 105 from the dataset

Fig 4 displays the probability values of whether a graph resembles a random graph for the full dataset. From the plot it can be seen that the majority of the probabilities have extremely small values that range from 1.3086E-05 to 2.2806E-52. The lower the probability values the less the graph resembles a random graph, which suggests that the majority of the graphs are not random. However, few of the graphs have probability values of up to 0.0343 which indicate that there is a 3.4 per cent chance of these graphs resembling a random graph. These values are reasonably high and it shows that there is an area of randomness in the way the software is structured at these points. Modularisation is not possible with data that resembles random graphs.

Due to the extremely small probability values produced we have computed the natural logarithm of these probabilities. Fig 5 shows the natural logarithm of the probability values ($\ln(p)$), the higher the value the more the graph resembles a random graph. From the plot it can be observed that graphs 100-180 have higher $\ln(p)$ values which indicates that at these points the graphs more resemble random graphs.

Fig 6 shows a plot of the $\ln(p)$ against active classes for the whole dataset. A general relationship can be observed from the plot, which shows that as the number of active classes increases $\ln(p)$ decreases, apart from the large peaks and drops between graphs 100-200. A value of -0.372 is produced when correlating $\ln(p)$ against active classes. This still indicates a high correlation as there are over 500 pairs of observations; the 1 per cent significance level is at 0.115.

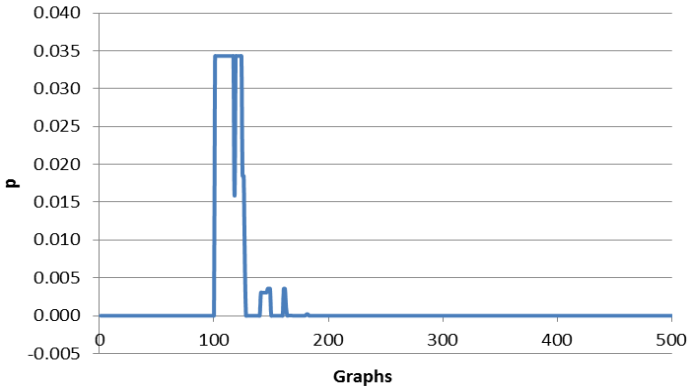

Fig. 4. Probability values representing the randomness of the graph

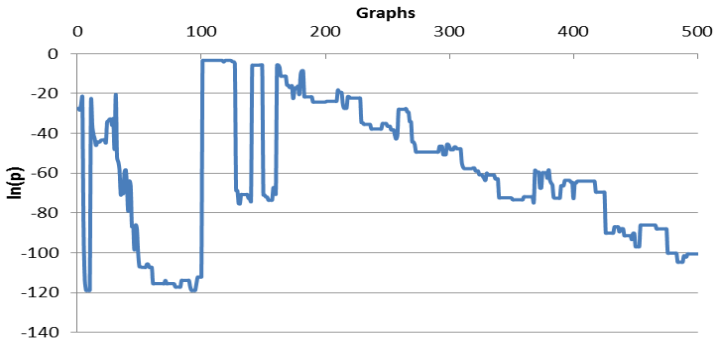

Fig. 5. The natural logarithm of the probability values for the whole dataset

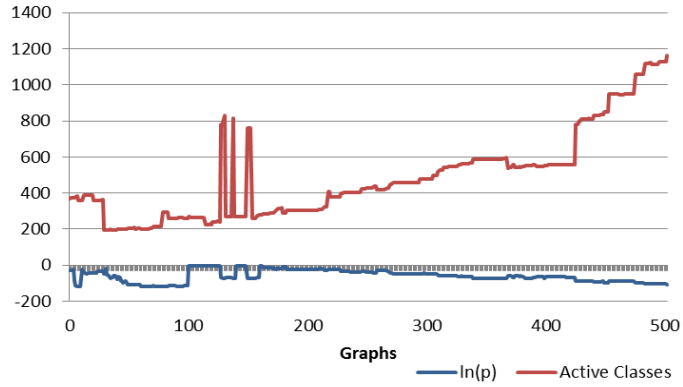

Fig. 6. The natural logarithm of the probability values against active classes

Fig 7 shows the relationship between ln(p) and EVM. It can be observed that as EVM increases, ln(p) decreases. To find out how strong is this relationship we correlated the two values for the whole dataset and for graphs 100-200 only. A value of 0.266 is produced for the whole dataset and −0.513 is produced for graphs 100-200.

These values indicate a strong correlation. In addition, correlating the ln(p) against the HS metric produced −0.403 over the whole dataset, which also indicates a very high correlation. These relationships demonstrate that the modularisation works well for the majority of the dataset (apart from the small activities between graph 100 and 200). It also suggests that the random graph metric can be used to quickly measure how effective the search is going to be and to indicate areas (software check-ins) of interest in the software, such as locating major changes and refactoring activities.

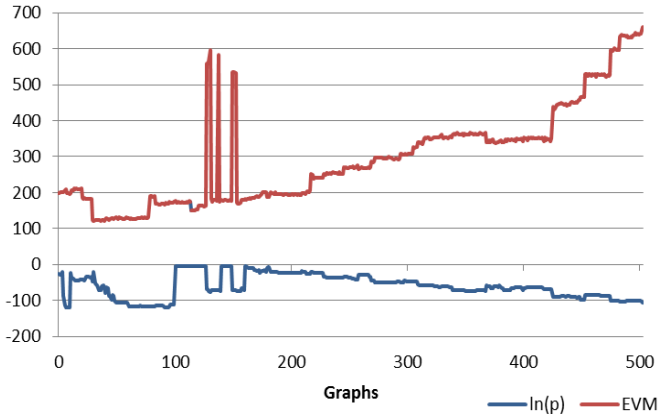

Fig. 7. The natural logarithm of the probability values against EVM

Fig 8 shows a plot of the ln(p) against AVD. Correlating the dataset results of the two values together produced no clear relationship, however, looking at the 100-200 graphs section of the dataset, produced −0.407. This suggests a strong negative correlation for this period, mainly due to the large number of activities.

In addition, Fig 8 shows that there are three time periods (graphs 101-127, 141-149 and 161-163) where there were very large differences in the probability values, revealing that these graph had up to 3.4 per cent chance of resembling a random graph. It is interesting to notice that these large changes in probability values occur just before the sizeable changes in the AVD and active classes. This suggested to us that during this period there was instability in the code. We investigated this further by correlating the results produced with information from the developers; we have had feedback on the results from the senior architect at Quantel, and were provided with all of the check-in comments for the dataset currently being analysed. During this period, the implementation of a new library caused some of the libraries to be unstable and to have unpredictable behaviour, developers were in a state of flux on how to use the libraries. There were few months of implementation that included coding the interface and trying out the libraries in different ways and then a roll back to the previous code. The roll back did not only include the library classes but also their own code. Thus, there were sizable shifts in the number of classes as they went through the different library models. It finally stabilises as they worked out the appropriate model to use. During this whole period there was evidence to suggest early product implementation with many issues in the code. We consider these large changes to be refactoring events and not new functionalities as internal structures of the code was changed without changing the functionality of the software.

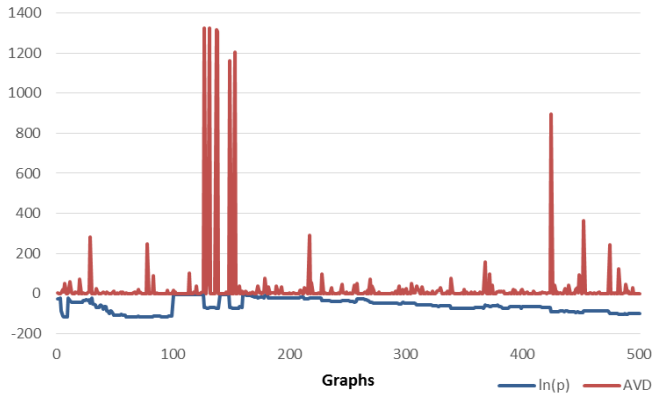

Fig. 8. The natural logarithm of the probability values against AVD

WK values for the clustering results of the 1^{st} graph and the i^{th} clustering results for the full modularisation were produced. There is a decreasing trend of the WK values which suggests that the original structure of the system deteriorates over time. Correlating ln(p) and WK did not produce a high correlation (0.159), however a relationship can still be observed. We look to investigate this further as part of future work.

5 Conclusions and Future Work

In this paper we have demonstrated that our time-series system does not resemble a random graph except for very small sections of the datasets where there were large activities. Thus, from the results it can be seen that the random graph metric can be used as a tool to focus on and indicate areas of interest in the dataset (without running the modularisation), such as where the system is starting to decay, if the link between classes is random or strongly resemble a random graph then the software is decaying. In addition, by looking at the source code check-ins, we look to use random graphs, observing when a certain percentage of randomness occurs, to indicate whether/when refactoring should be performed. The random graph test may also be used to indicate areas where a higher running time of the modularisation is needed. We look to investigate these relationships further as part of future work.

There are a number of random graph models; however, for this paper we only used the Erdős–Rényi model which calculates the probability of a node being connected to another node. Other models which include the degree of the number of edges that are connected to a node as opposed to the connectivity of the node will be investigated for future work. Furthermore, we look to investigate the relationship further by trying it on few other different systems.

References

1. Altman, D. G.: Practical Statistics for Medical research. Chapman and Hall (1997)

2. Arzoky, M., Swift, S., Tucker, A., Cain J.: Munch: An Efficient Modularisation Strategy to Assess the Degree of Refactoring on Sequential Source Code Checkings. IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp. 422−429 (2011)
3. Arzoky, M., Swift, S., Tucker, A., Cain J.: A Seeded Search for the Modularisation of Sequential Software Versions. Journal of Object Technology, vol. 11, No. 2, pp. 6:1−27 (2012)
4. Barabási, A. L., Albert, R., Jeong, H.: Scale-free characteristics of random networks: the topology of the world-wide web. Physica A: Statistical Mechanics and its Applications, vol 281, no. 1, pp. 69−77 (2000)
5. Cain, J., Counsell, S., Swift, S., Tucker A.: An Application of Intelligent Data Analysis Techniques to a Large Software Engineering Dataset. Advances in Intelligent Data Analysis VIII: 8th International Symposium on Intelligent Data Analysis (IDA09). Lecture Notes in Computer Science (2009)
6. Chidamber S. R., Kemerer, C. F.: A metrics suite for object oriented design. IEEE Trans, Software Eng., vol. 20, no. 6, pp. 476−493 (1994)
7. Constantine, L. L., Yourdon, E.: Structured Design. Prentice Hall (1979)
8. Doval, D., Mancoridis, S., Mitchell, B. S.: Automatic clustering of software systems using a genetic algorithm. In Software Technology and Engineering Practice. IEEE Proceedings STEP'99, pp. 73−81 (1999)
9. Erdős, P., Rényi, A.: On the evolution of random graphs. Magyar Tud. Akad, Mat. Kutató Int. Közl. vol. 5, pp. 17–61 (1960)
10. Gilbert, E. N.: Random graphs. The Annals of Mathematical Statistics, pp. 1141−1144 (1959)
11. Harman, M., Hierons, R., Proctor, M.: A new representation and crossover operator for search based optimization of software modularization. Proc. Genetic and Evolutionary Computation Conference, Morgan Kaufmann Publishers, pp. 1351–1358 (2002)
12. Harman, M., Mansouri, S. A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys, vol. 45, no. 1, pp. 11 (2012)
13. Harman, M., Swift, S., Mahdavi, K.: An empirical study of the robustness of two module clustering fitness functions. Genetic and Evolutionary Computation Conference, Washington, DC, pp. 1029–1036 (2005)
14. Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y., Gansner, E. R.: Using automatic clustering to produce high-level system organizations of source code. In International Workshop on Program Comprehension (IWPC'98), IEEE Computer Society Press, Los Alamitos, California, pp. 45−53 (1998)
15. Massey, F. J.: The Kolmogorov-Smirnov Test for Goodness of Fit. Journal of the American Statistical Association. vol. 46, no. 253, pp. 68−78 (1951)
16. Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., Bhattacharjee, B.: Measurement and analysis of online social networks. In Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, pp. 29−42 (2007)
17. Mitchell, B. S.: A Heuristic Search Approach to Solving the Software Clustering Problem. PhD Thesis, Drexel University, Philadelphia, PA (2002)
18. Praditwong, K., Harman, M., Yao, X.: Software Module Clustering as a Multi–Objective Search Problem. IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 264-282 (2011)
19. Roth, C., Kang, S. M., Batty, M., Barthelemy, M.: A long-time limit for world subway networks. Journal of The Royal Society Interface, vol. 9, no. 75, pp. 2540−2550 (2012)
20. Tucker, A., Swift, S., Liu, X.: Variable Grouping in multivariate time series via correlation. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, vol. 31, no. 2, pp. 235–245 (2001)