Open Access Article

ENHANCE SOFTWARE MODULE CLUSTERING OF OBJECT-ORIENTED SYSTEMS USING DYNAMIC WEIGHTED MODULE DEPENDENCY GRAPHS AND MULTI-OBJECTIVE HGW2O ALGORITHM

Harleen Kaur
Research scholar Department of Computer Science and Engineering, NIT Jalandhar
(email- er.harleenkaur@gmail.com)

Geeta Sikka.
Associate professor in Department of Computer Science and Engineering, NIT Jalandhar
(email: sikkag@nitj.ac.in)

Abstract— As the software systems evolve with the rapid change in requirements to adapt to the latest trends and technologies the system structure is vitiated. To repossess the cognizance of the system re-modularization of the code becomes essential. To restructure the package structure of a system, software maintainer must have complete knowledge f the existing system structure. This paper focuses on enhancing the lost structure of a system by analyzing the system structure using weighted module dependency graphs, where dependency relations are weighted according to the strength of the relation. Static and dynamic module dependency graphs are generated and their efficacy is tested on three different problem instances. The generated weighted module dependency graphs are further used to re-modularize the system using two Multi objective algorithms for evaluation of the efficacy of the approach.

Index Terms— Grey Wolf optimization, whale optimization, HGW2OA, MDG, Weighted module dependency graphs

摘要：随着软件系统随着需求的快速变化而发展以适应最新的趋势和技术，系统结构已经损坏。收回对系统重新模块化代码的认识变得至关重要。要重构系统的包结构，软件维护人员必须对现有系统结构有完整的了解。本文侧重于通过使用加权模块依赖图分析系统结构来增强系统的丢失结构，其中依赖关系根据关系的强度进行加权。生成静态和动态模块依赖图，并在三个不同的问题实例上测试它们的功效。生成的加权模块依赖图进一步用于使用两个多目标算法对系统进行重新模块化，以评估该方法的功效。

索引词——灰狼优化、鲸鱼优化、HGW2OA、MDG、加权模块依赖图

INTRODUCTION

Software has now become an essential component of almost all big firms, whether they are social, government or private. Since the mechanism is constantly changing, software upgrades or changes are inevitable. The

Received: July 02, 2021 / Revised: August 22, 2021 / Accepted: September 11, 2021 / Published: September 30, 2021
About the authors Harleen Kaur
Corresponding author- Email: er.harleenkaur@gmail.com

constraints are blurred as the system framework disintegrates, and the framework seems monolithic rather than modular [1,16]. As a result, attempting to correct a bug in one module will cause a bug in the other. As a result, maintaining and evolving huge, complex, and intricate structures becomes difficult. Inadequately modularized software is thought to be a source of problems in terms of acceptance, maintenance, and screening. Seeking suitable modularization is indeed a time-consuming and difficult process, as it must appeal to the engineer and developer while adhering to sound design concepts. Software clustering is among the trained frameworks for achieving consistency and increasing device comprehension.

When a system becomes enormous, the basic principle of divide and rule is used to disintegrate this into sub - systems; additionally, if a sub - system becomes excessively enormous, it must be disintegrated as well. To disintegrate a framework, designers must first comprehend how well the modules are connected, whether they are connected firmly or weakly. Clusters of extremely robust modules are aimed at creating a design that is maintainable. The very first phase in clustering software modules is to create a Module Dependency Graph (MDG) to represent the structure of the software system [11,12]. Dependencies have been taken into account because they aid in determining a software system's efficiency. A tightly coupled framework can have numerous dependencies due to the possibility of numerous cyclic dependencies; such dependencies can cause the implementation to stall for a variety of reasons, including repetitions, code complexity, and so on. MDG enables the creation of a comprehensive view of the source code's existing dependencies. Additionally, MDGs aid in trying to trace code smells within the system, which facilitates refactoring. A sequence of nature-inspired meta-heuristic optimization algorithms has been suggested to enhance the clustering and modularization of legacy software systems. Such nature-inspired algorithms enable us to increase the solution's efficacy and rapidly obtain an appropriate optimum solution in order to solve complex problems effectively. Additionally, it is acknowledged that a single optimizer is incapable of meeting the precision requirement when trying to solve the problems of software engineering attributed to the prevalence of numerous design parameters and constricted conditions. Consequently, hybrid algorithms appear to become the most viable technique for accomplishing these goals. The algorithm GREY WOLF, which is focused on the collective relations and social behavior of spotted hyenas, is found to have a high capacity for global search and thus enhanced exploration abilities. Moreover, it is constrained by a lower population density and slow convergence. As a result, its robustness continues to suffer, as this may occasionally become trapped in local optima. But at the other hand, the WHALE algorithm generates optimum solution with a higher degree of convergence owing to its strategy to leader and follower selection [6,7]. Additionally, WHALE has fewer controlling parameters and a more capable local search. As a result, we proposed a new algorithm called HGW2O to acquire the desired globally optimal solution to confined optimization issues with enhanced exploitation and exploration abilities Thereby, HGW2O contributes to avoiding premature convergence and achieving a stability between solution quality and computational time [4,6].

Software modules and clustering of software modules

Software Module

Software modularity seems to be an internal quality characteristic that has an effect on the outer quality characteristics of software. Modularity enables advancement flexibility and it can be applied at any abstract level. This is a structural engineering concept that extends from specified requirements to executable code [1,5].

Clustering of Software Modules

When software modularization is viewed as a problem of clustering, analogous entities are clustered together and distinct entities are usually added to distinct groups. A resemblance between both the different bodies is computed depending on the types and characteristics of the traits or attributes.

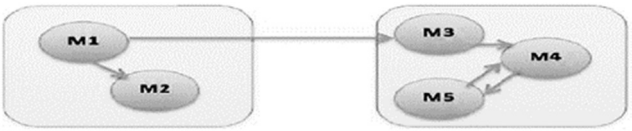

Figure 1MGD Clustering [2]

An entity is often a software module, a class or a routine, or a class, whereas an attribute is a software artefact including a package, file, documentation, function, test case, or query. Formal features are global variables that entities use and calls are generally made by entities, whereas informal features are comments or identifiers inside an entity. The following are the different stages of cluster analysis [2,9,13]:

- Characteristic Retrieval and filtering: To apply a clustering methodology system, facts or features must be retrieved from software platform.
- Calculation of similarity: Similarity is quantified using several resemblance coefficients; these coefficients may be numeric, binary, software-specific, or probabilistic.
- Cluster creation: Once an initial step of filtering and feature fact extraction are complete, the Cluster Creation process can begin. Clustering algorithms fall under a range of subjects, including hierarchical, construction algorithms, graph-theoretic, and optimization algorithms.
- Results visual representation and validation: It is critical to evaluate the outcomes in order to take appropriate steps to minimize the impact of weak points and to improve the system. A system with a higher proportion of intra-edge dependencies than inter-edge dependencies is referred to as a finely clustered systems with low coupling and high degree of cohesion [14,15].

Static, Dynamic and Hybrid Approaches for Dependency Analysis

Information about Dependency

Numerous software engineering activities and analytical techniques, like clustering, modularization, testing, clone detection, refactoring, and fault detection are implemented by numerous program-dependency models [18], like call graphs [10], module dependence graphs [19,20], system dependence graphs [19], and dependence condition graph (DCG) [17]. Such models are being used to retrieve the configuration of the software system. Decomposing a system's software into smaller, more malleable clusters is often a popular technique for improving a huge system's comprehension.

Static System

- A software system's static study demonstrates the dependencies between the software objects/entities that are used for cluster analysis.

- The objects may be source files, packages or classes but are limited to classes throughout this study, and the dependencies are procedure/function variables and calls references.
- The static information in a software system is definitive, i.e., the characteristics of objects are derived from discrete sets of values with no particular ordering.
- So, if dependencies are modelled statically, the possible sign of accomplishing perfect modularization tends to decrease.
- Due to the complexity and behavioural characteristics of software systems, structural analysis relationships are insufficient for software clustering. Numerous dependencies exist during the execution of a programme. These dependencies impose a particular order mostly on operations' execution which could be hard to trace statically at times.

Dynamic System

- It is determined how frequently each procedure in a file or class calls processes in other files or classes.
- Dynamic call graphs are created utilising dynamic information, in which accessing/calling relations are retrieved from execution trace amounts and presented graphically.
- Dynamically assessed dependencies aid in improving modularization by giving a clear understanding of the system structure via the initial call trace.

The instance programme in figure ill2ustrates how and when to change the reference to a superclass object to a subclass object. Whenever a superclass technique which has been overruled in a subclass is challenged via the superclass object, it definitely implements the overruled definition rather than the superclass description, since methodologies in Java are dynamically deployed. Whenever this code is statically evaluated, this dependency between superclass A and subclass B may be missed [7,8].

This section discusses the weighted MDG of an OO system in detail, with the goal of enriching the dependency information. This study discusses a method for calculating the weights of relations using the code, where weights are not solely based on the number of invocations, but rather by the type of dependency. For the purposes of this research, an MDG class is regarded a module, and the relation between modules denotes their interdependence.

Strength	Coupling Type
High	Inheritance
	Content coupling
	Common coupling
	Control coupling
	Stamp coupling
	Data coupling
	Routine call coupling
	Type-use coupling
	Include/Import
Low	

Figure 2: Types of couplings[3]

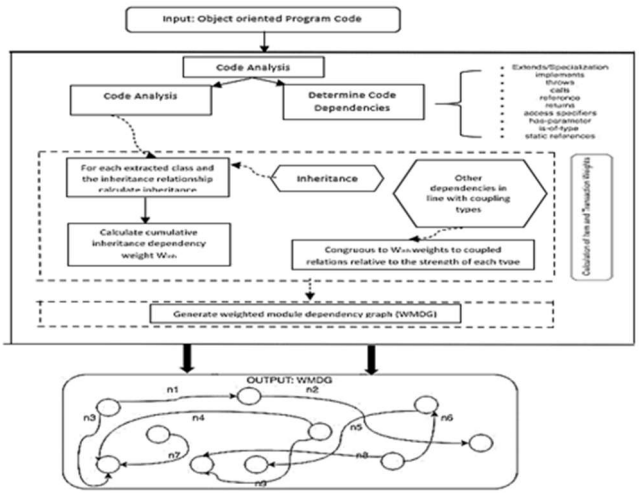

Figure 3 System architecture [1]

The proposed method is depicted in figure 3, where the object-oriented (OO) programme code is used as an input.

- Code Analysis and Dependency Mapping Code: It is analysed to retrieve classes and relations between them; the classes and relations are then mapped to the appropriate coupling types to evaluate the strength of relationship.
- Complexity of Inheritance: The inheritance relations to every class are retrieved from the source code and the Inheritance Class Complexity (ICC) is computed. Additionally, a cumulative inheritance dependency weight was discovered.
- Strength of Coupling Dependency: The strength of the surviving relations is measured in accordance with the inheritance dependency weights.

Code analysis is performed on the input java code to retrieve all of the methods, classes, and the relations between them.

HGW2O algorithm

Grey Wolf Optimizer Algorithm

GWO is inspired by the spotted hyena's encircling, attacking, hunting, and searching behaviours. The encircling property is simulated in the following manner [4,6,8]:

Here, c = current iteration; $(M\_h)\vec{}$= distance enclosed by hyena to reach prey; $N\vec{}$ and $(N\_p)\vec{}$=position vectors of spotted hyena and prey, respectively; Additionally, the absolute value and multiplication vector are signified by means of || and symbols, respectively.

Here, nos denote the number of candidate solutions, is $S\vec{}_h$denotes the cluster of optimal solutions, and $T\vec{}$ denotes a random vector with values between [0.5, 1]. $N\vec{}$ (c + 1) gives the optimum solution and also assists in reconfiguring the places of several other spotted hyenas.

WHALE algorithm

WHALE is a recently developed bio-inspired meta-heuristic optimization technique [3,4,6] that is motivated by salp swarm behaviour. To explore and find food mostly in ocean depths, such swarms form a salp chain. The populace of this chain is divided into two distinct groups, namely leaders and followers. The salp at the top of the chain is referred to as the leader, while the residual salps are referred to as followers.

By incorporating the leader and follower, the HGW2O algorithm, which manages to combine the WHALE and GREY WOLF algorithms, adjusts the basic framework of the enhanced GREY WOLF algorithm. Such an assimilation increased flexibility in terms of reaching a good balance in both convergence and diversity, along with rapidly arriving at the optimal solution. HGW2OA initializes the prelim parameters and the count of spotted hyenas with values. The efficacy of each searching agent would then be

estimated using the fitness function in order to obtain non-dominated solutions that are then archived. This archive would then be navigated in order to determine the optimal search agent. After obtaining the cluster of optimal solutions from the archive utilizing equations 8 and 9, the position of each searching agent is modified using proposed enhanced median approach described in till a satisfying information is achieved for each search agent. Med () is indeed the mathematical operation of Median, and it chooses its most optimum solution nearby middle solutions from sorted (Sort ()) team of spotted hyenas. The subsequent phase would be to use the WHALE algorithm's leader and follower polling method to recalculate and then further modify the places of search. The integration of such a WHALE process aids in emancipating a solution from local optima and also improves the solution quality with the quickest rate of convergence.

At the end of each iteration, if newly inserted solutions extend far beyond existing grid bounds, HGW2OA recomputes the grid locations to accommodate the new solutions.

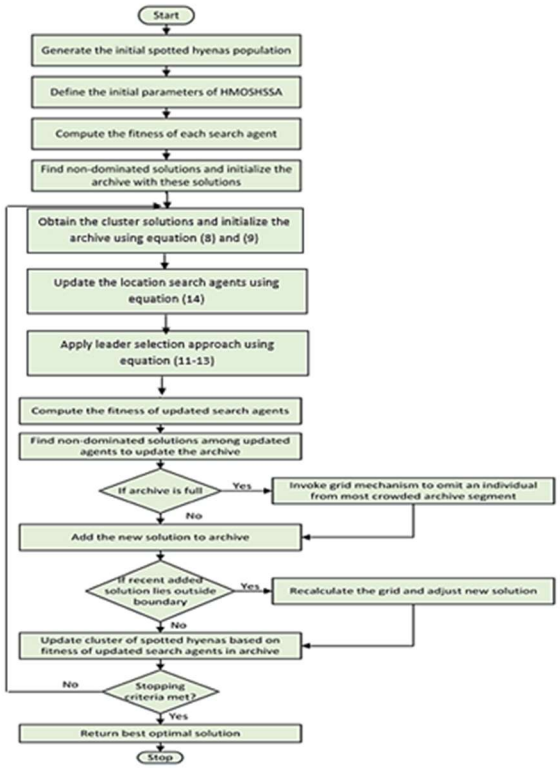

Figure 4: HGW2O algorithm [1]

II RELATED WORK

Geeta Sikka and Harleen Kaur [1] concentrates on enhancing the MDG in order to improve comprehension of the design methodology by setting specific weights to various types of code dependencies. Weights are assigned to each type of coupling relation. To create weighted MDGs, a naive function has also been described. MDGs have been produced through both static and dynamic analysis. The efficacy of the methodology is evaluated by using these graphs for software re-modularization. Qusay Alsarhan et al. [2] carried out an extensive review of 143 research articles using software module cluster analysis from the well literature datasets in order to obtain valuable information. The collected data were often used to address numerous questions about state-of-the-art clustering strategies, clustering application forms in software design, clustering processes,

assessment methods, and clustering algorithms. Mohammed H. Qais et al. [3] propose the TSO i.e., Transient Search Optimization algorithm, which would be a modern physical-based meta-heuristic optimization algorithm. The transient behaviour of switched electrical circuits with storage elements including capacitance and inductance inspired such algorithm. The TSO algorithm's exploitation and exploration was validated by comparing its statistical (standard and average deviation) performance to some of the most recent 15 optimization algorithms utilizing 23 benchmarks. Raja Masadeh et al. [4] suggest a hybrid approach that is based on Whale and Grey wolf optimal algorithms (WGW) to prioritise software system requirements by incorporating the strengths of each algorithm. Furthermore, the data set used here is RALIC, which would be a prerequisite for a specific software development project in order to evaluate the proposed process. When compared to RALIC results, the proposed method gives a 91 percent accuracy in requirements prioritisation. Rathee and Chhabra [5] conducted an empirical study to analyse the impact of various dependency relations in modelling dependency amongst various software elements. According to an empirical assessment, a software system's change history plays a major role in modelling dependency relations, and therefore must be used during conjunction with other relationships for more precise measurements. Then, by merging conceptual, structural, and change history-based relationships among software components, a new weighted dependency measuring framework is developed, with evolutionary dependency relationships given greater weight.

To address multi-objective issues, Seyedali Mirjalili et al. [6] propose a new optimization algorithm named Multi-objective Salp Swarm Algorithm (MSSA) and Salp Swarm Optimizer (SSA). Both the algorithms are put through their paces with a set of mathematical optimization functions. Marine propeller and Airfoil design are two difficult design engineering issues that have been resolved. The performance of SSA and MSSA is shown by the quantitative and qualitative results. Gaurav Dhiman and Vijay Kumar [7] suggest the spotted hyena optimization algorithm, which is named after hyenas. On 29 possibly the best benchmark test functions, the SHO algorithm is put to the test. The obtained results on uni-modal benchmark test functions confirm SHO's exploitation property. The obtained results on multi - modal benchmark test functions confirm SHO's exploration ability. SHO's suitability in real-world applications is demonstrated by the outcomes of five restricted engineering issues. Huang Jinhuang et al. [9] suggested a multi-agent evolutionary algorithm, dubbed MAEA-SMCPs, to resolve this issue. Three evolutionary operators are intended with intrinsic properties of SMCPs in mind to enable agents to achieve the goals of cooperation, competition, and self-learning. Practical problems are being used to validation of the model of MAEA-SMCPs inside the experiments. The findings suggest that MAEA-SMCPs are capable of identifying clusters of great quality with few variations. Vineeta Singh [11] introduces a review of the literature on various metaheuristic search strategies that are being used to solve the issue of software module clustering mostly in software maintenance phase of the software development process. Mark Shtern and Vassilios Tzerpos [13] identify critical directions for future research mostly in field of software clustering that require extra interest in terms of developing more

efficient and effective software engineering clustering methods. To a certain end, the authors begin by reviewing the state of the art in studies on software clustering. They start debating the cluster analysis which have garnered its most attention from researchers and explain their advantages and disadvantages. Muhammad et al. [14] categorise and assess a huge number of relations which may exist among entities in an object-oriented (OO) system. Studies on modularisation are performed and use a variety of hierarchical clustering algorithms. The test results suggest the relations that enhance the quality of the algorithms' results but may therefore be regarded more significant for software clustering. Bangare et al. [15] made an experimental model for validating metrics used to assess the modularization quality of an Object-Oriented Software System. The work presented here details the work performed using a code analyzer. The API idea is often used as a foundation for structural metrics. To validate the metrics, the researchers compare the results obtained on a sample input code file to the modularized variants of the OO java software code file.

PROPOSED METHODOLOGY

$$X(t+1) = \frac{x_1 + x_2 + x_3}{3} \dots\dots (1)$$

$$D = |C * X(t) - X(t)| \dots\dots\dots (2)$$

$$X(t+1) = D` . e^{bl} . \cos 2\pi l + X * t \dots . (3)$$

3.1 Proposed Methodology

Step-1: Initialize population X_i (I = 1,2,3……, n)

Step-2: Initialize a, A, and C

Step-3: Calculate fitness of each search agent

$X_a =$ the best

$X_p =$ the second best

$X_\delta =$ the third best

Step-4: While (t < max no. of iterations)

$for\ each\ search\ agent$

update a, A, C, l, and p

$if1\ (p < 0.5)$

$if2\ (|A| < 1\ )$

$update\ the\ position\ of\ current\ search\ agent\ using\ equa$

$else\ if2\ (|A| \geq 1\ )$

$update\ the\ position\ of\ current\ search\ agent\ using\ equa$

$end\ if2$

$else\ if1(p \geq 0.5)$

$update\ the\ position\ of\ current\ search\ agent\ using\ equ$

$end\ if1$

$end\ for$

$check\ if\ any\ search\ agent\ goes\ beyond\ the\ search\ space$

$calculate\ fitness\ of\ each\ search\ agent$

$update\ X_a, X_p, \& X_\delta$

$t = t + 1$

$end\ while$

Step-5: $return\ X_a$

			MOHGW2O			
Performance Metric	**Problem Instance**	**Parameter**	**Un-weighted relations**	**WCC**	**WDR (Static)**	**WDR (Dynamic)**
Percentage of moved classes from their original packages	JavaCC	Mean	78.23	19.271	14.71	11.23
		STD	10.92	3.1	5.11	3.88
	Junit	Mean	86.62	12.444	12.44	9.63
		STD	8.11	3.111	3.99	3.77

	Java Servlet API	Mean	65.33	7.31	8.8	6.98
		STD	10.23	1.29	1.01	1.78
RRAI-MQ evaluation	JavaCC	Mean	0.419	3.912	4.866	6.112
		STD	0.009	0.042	0.057	0.058
	Junit	Mean	0.398	5.959	6.001	7.401
		STD	0.019	0.065	0.071	0.052
	Java Servlet API	Mean	0.361	4.728	3.844	6.348
		STD	0.012	0.074	0.045	0.038
Percentage improvement in MMF	JavaCC	Mean	0.451	0.574	0.694	0.683
		STD	0.011	0.012	0.012	0.016
	Junit	Mean	0.642	0.731	0.83	0.811
		STD	0.021	0.066	0.051	0.063
	Java Servlet API	Mean	0.285	0.277	0.531	0.596
		STD	0.022	0.0	0.029	0.028
				41		
Percentage improvement in MQ	JavaCC	Mean	30.71	29.53	31.38	31.89
		STD	3.76	3.84	3.13	3.42
	Junit	Mean	68.52	61.33	71.39	71.01
		STD	3.83	3.96	4.77	3.78
	Java Servlet API	Mean	35.67	29.32	39.26	43.98
		STD	2.19	2.61	3.01	2.63

Table1 : Proposed approach experiment results value on different performance metrics

			NSGA-II			
Performance Metric	**Problem Instance**	**Parameter**	**Un-weighted relations**	**WCC**	**WDR (Static)**	**WDR (Dynamic)**
Percentage of moved classes from their original packages	JavaCC	Mean	78.23	19.27	14.71	11.23
		STD	10.92	3.1	5.11	3.88
	Junit	Mean	86.62	12.44	12.44	9.63
		STD	8.11	3.11	3.99	3.77
	Java Servlet API	Mean	65.33	7.31	8.8	6.98

		STD	10.23	1.29	1.01	1.78
RRAI-MQ evaluation	JavaCC	Mean	0.419	3.912	4.866	6.112
		STD	0.009	0.042	0.057	0.058
	Junit	Mean	0.398	5.959	6.001	7.401
		STD	0.019	0.065	0.071	0.052
	Java Servlet API	Mean	0.361	4.728	3.844	6.348
		STD	0.012	0.074	0.045	0.038
Percentage improvement in MMF	JavaCC	Mean	0.451	0.574	0.694	0.683
		STD	0.011	0.012	0.012	0.016
	Junit	Mean	0.642	0.731	0.83	0.811
		STD	0.021	0.066	0.051	0.063
	Java Servlet API	Mean	0.285	0.277	0.531	0.596
		STD	0.022	0.041	0.029	0.028
Percentage improvement in MQ	JavaCC	Mean	30.71	29.53	31.38	31.89
		STD	3.76	3.84	3.13	3.42
	Junit	Mean	68.52	61.33	71.39	71.01
		STD	3.83	3.96	4.77	3.78
	Java Servlet API	Mean	35.67	29.32	39.26	43.98
		STD	2.19	2.61	3.01	2.63

Table2 : Existing approach experiment results value on different performance metrics

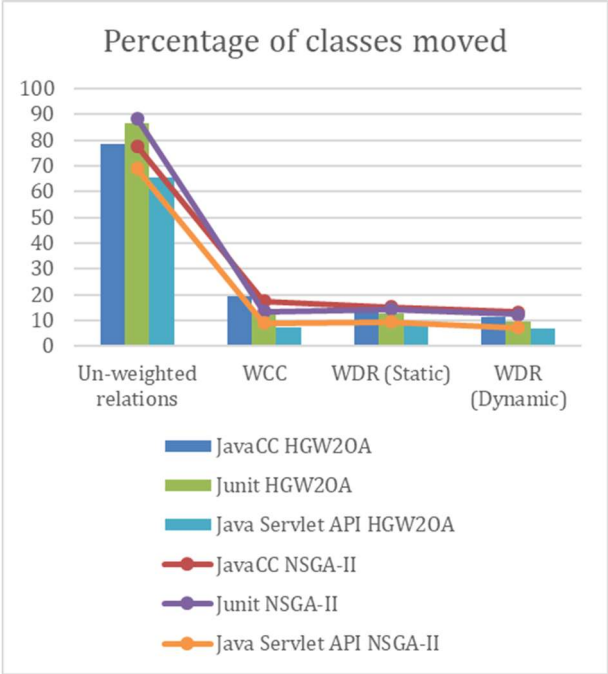

Fig5 Comparison of moved classes

In fig 5 show the Percentage of classes moved is highest when unweighted dependency relations are considered. Weighted class connections and weighted dependency relations (static & dynamic) show that there are significantly lesser

number of classes that have been moved from original packages. MOHGW2O algorithm shows better results as compared to NSGA-II. Percentage of classes moved is highest when unweighted dependency relations are considered. Weighted CC and WDR (static & dynamic) show that there are significantly lesser number of classes that have been moved from original packages. HGW2O algorithm shows better results as compared to NSGA-II.

In figure 6 RRAI(MQ) values very clearly indicate that the un-weighted relations underperform since all the values are less than the baseline value (i.e., 1). Modularization achieved using Weighted CC and the WDR (Static and dynamic) produce acceptable results with both the MOHGW2O and NSGA-II indicating minimal modification in the original structure.

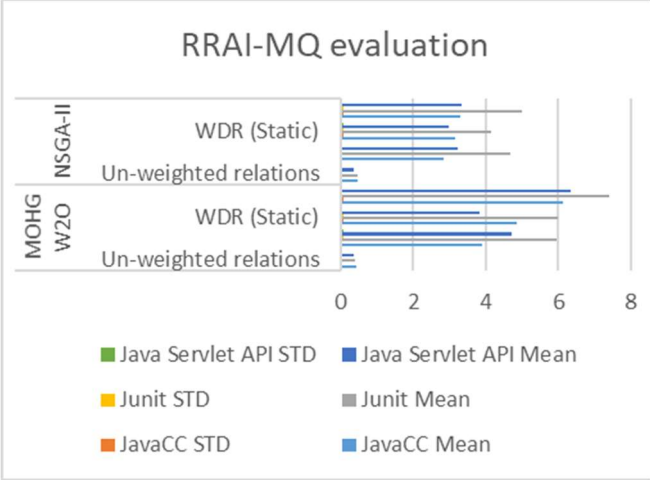

Fig6: Comparison of RRAI-MQ evaluation

The best results showing significant improvement with higher cohesion and low coupling are attained using the WDR modularized with MOHGW2O.in figure 7 show Percentage improvement in MMF for un-weighted relations and weighted CC instances is The best results showing significant improvement with higher cohesion and low coupling are attained using the WDR modularized with MOHGW2O in comparison to NSGA-

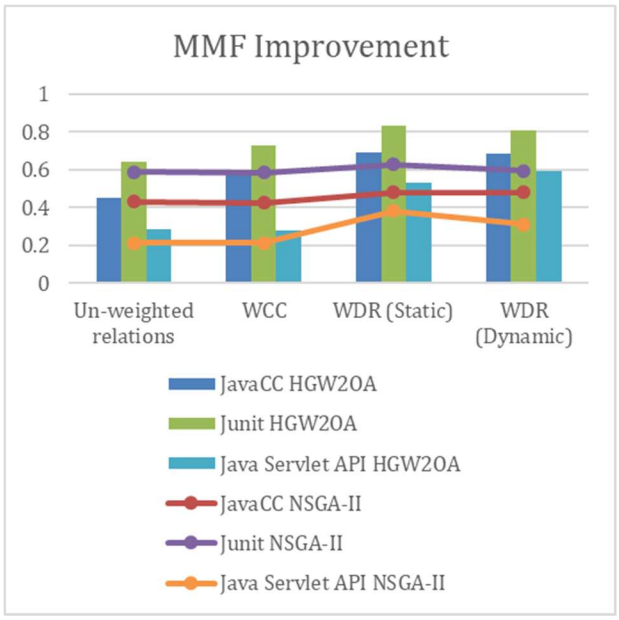

Fig7: Comparison of Percentage improvement in MMF

WDR and un-weighted relation instances In figure 8 show more improvement in the MQ percentages with MOHGW2O. NSGA-II improvement percentage is lesser as compared to MOHGW2O.

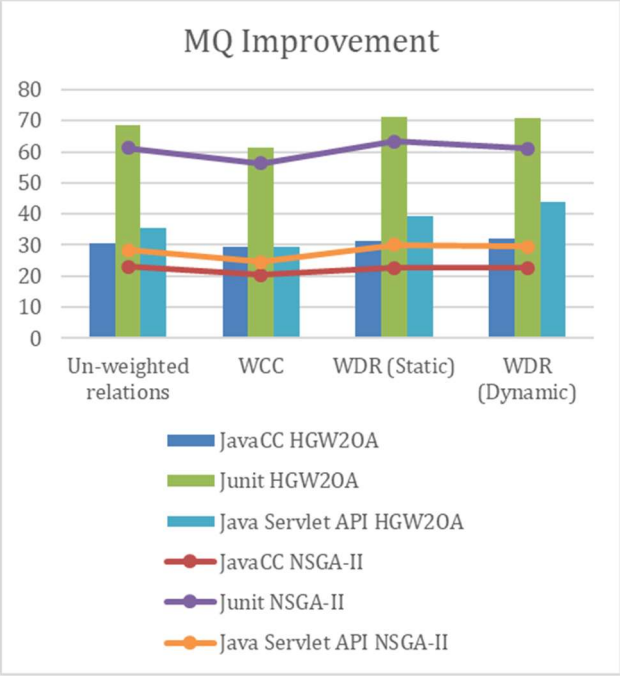

Fig8: Comparison of Percentage improvement in MQ

II.Percentage improvement in MMF for un-weighted relations and weighted CC instances is quite similar.WDR and un-weighted relation instances show more improvement in the MQ percentages with HGW2OA. NSGA-II improvement percentage is lesser as compared to HGW2OA.

V Conclusion

The first step to clustering and re-modularization is generation of Module Dependency Graphs of the existing systems. When considering improvement of system structure researchers think of improving the clustering algorithms but the results in this paper shows that with the same algorithms if we improve the dependency information better modularization can be achieved. Better modularization in terms of high cohesion and low coupling. Static relations in comparison to the dynamic weighted relations showed significant improvement in results, which indicates that the unwanted relations are certainly ignored while creating dynamic dependency relations. The proposed weighted dependency relations in this research when tested for modularization with NSGA-II and HGW2O produced better results as indicated by the MMF and MQ percentage improvements. The results also conclude that HGW2O generated better results as compared to NSGA-II algorithm on taking the dynamic as well as static dependency graphs as input.

The results can further be refined using multiple systems of varying sizes and by increasing the number of runs of clustering algorithm to get more accurate results.

REFERENCES

- Kaur, H. and Sikka, G., 2021. Dynamic Analysis Based Software Modularization Augmenting Weighted Module Dependency Graphs. Journal of Software Engineering Tools & Technology Trends, 7(3), pp.27-40.
- Alsarhan, Q., Ahmed, B.S., Bures, M. and Zamli, K.Z., 2020. Software Module Clustering: An In-Depth Literature Analysis. IEEE Transactions on Software Engineering.
- Qais, M.H., Hasanien, H.M. and Alghuwainem, S., 2020. Transient search optimization: a new meta-heuristic optimization algorithm. Applied Intelligence, 50(11), pp.3926-3941.
- Masadeh, R., Hudaib, A. and Alzaqebah, A., 2018. WGW: A hybrid approach based on whale and grey wolf optimization algorithms for requirements prioritization. Advances in Systems Science and Applications, 18(2), pp.63-83.
- Rathee, A. and Chhabra, J.K., 2018. Clustering for software remodularization by

using structural, conceptual and evolutionary features. Journal of Universal Computer Science, 24(12), pp.1731-1757.

- Mirjalili, S., Gandomi, A.H., Mirjalili, S.Z., Saremi, S., Faris, H. and Mirjalili, S.M., 2017. Salp Swarm Algorithm: A bio-inspired optimizer for engineering design problems. Advances in Engineering Software, 114, pp.163-191.
- Dhiman, G. and Kumar, V., 2017. Spotted hyena optimizer: a novel bio-inspired based metaheuristic technique for engineering applications. Advances in Engineering Software, 114, pp.48-70.
- Dhiman, G. and Kaur, A., 2017, December. Spotted hyena optimizer for solving engineering design problems. In 2017 international conference on machine learning and data science (MLDS) (pp. 114-119). IEEE.
- Huang, J., Liu, J. and Yao, X., 2017. A multi-agent evolutionary algorithm for software module clustering problems. Soft Computing, 21(12), pp.3415-3428.
- Tempero, E. and Ralph, P., 2016, December. A model for defining coupling metrics. In 2016 23rd Asia-Pacific Software Engineering Conference (APSEC) (pp. 145-152). IEEE.
- Singh, V., 2016, March. Software module clustering using metaheuristic search techniques: A survey. In 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom) (pp. 2764-2767). IEEE.
- Kumari, A.C. and Srinivas, K., 2016. Hyper-heuristic approach for multi-objective software module clustering. Journal of Systems and Software, 117, pp.384-401.
- Shtern, M. and Tzerpos, V., 2012. Clustering methodologies for software engineering. Advances in Software Engineering, 2012.
- Muhammad, S., Maqbool, O. and Abbasi, A.Q., 2012. Evaluating relationship categories for clustering object-oriented software systems. IET software, 6(3), pp.260-274.
- Bangare, S.L., Khare, A.R. and Bangare, P.S., 2011, February. Quality measurement of modularized object-oriented software using metrics. In Proceedings of the International Conference & Workshop on Emerging Trends in Technology (pp. 771-774).
- Praditwong, K., Harman, M. and Yao, X., 2010. Software module clustering as a multi-objective search problem. IEEE Transactions on Software Engineering, 37(2), pp.264-282.
- Sukumaran, S., Sreenivas, A. and Metta, R., 2010. The dependence condition graph: Precise conditions for dependence between program points. Computer Languages, Systems & Structures, 36(1), pp.96-121.
- Maia, M.C.O., Bittencourt, R.A., de Figueiredo, J.C.A. and Guerrero, D.D.S., 2010, March. The hybrid technique for object-oriented software change impact analysis. In 2010 14th European Conference on Software Maintenance and Reengineering (pp. 252-255). IEEE.
- Horwitz, S., Reps, T. and Binkley, D., 2004. Interprocedural slicing using dependence graphs. Acm Sigplan Notices, 39(4), pp.229-243.
- Ferrante, J., Ottenstein, K.J. and Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS), 9(3), pp.319-349.