Applying Graph Partitioning-Based Seeding Strategies to Software Modularisation

Ashley Mann(✉), Stephen Swift, and Mahir Arzoky

Brunel University London, UB8 3PH Uxbridge, UK
{Ashley.Mann,Stephen.Swift,Mahir.Arzoky}@brunel.ac.uk

Abstract. Software modularisation is a pivotal facet within software engineering, seeking to optimise the arrangement of software components based on their interrelationships. Despite extensive investigations in this domain, particularly concerning evolutionary computation, the research emphasis has transitioned towards solution design and convergence analysis rather than pioneering methodologies. The primary objective is to attain efficient solutions within a pragmatic timeframe. Recent research posits that initial positions in the search space wield minimal influence, given the prevalent trend of methods converging upon akin local optima. This paper delves into this phenomenon comprehensively, employing graph partitioning techniques on dependency graphs to generate initial clustering arrangement seeds. Our empirical discoveries challenge conventional insight, underscoring the pivotal role of seed selection in software modularisation to enhance overall outcomes.

Keywords: Software Engineering · Heuristic Search · Software Modularisation · Graph Partitioning

1 Introduction

1.1 General Background

As software systems grow, maintenance becomes challenging for incoming engineers unfamiliar with the original code, often leading to the need for significant overhauls or discontinuation of extensive legacy systems. To ensure sustainable management, creating modular subsystems is crucial. Instead of portraying them as clusters in the source code, a more practical approach is representing dependencies in graph form. Mancoridis et al. define the software modularisation problem as arising from the exponential complexity of interconnected software module relationships within evolving systems. This is often approached as a heuristic-search-based clustering problem to identify optimal representations by clustering subsystems based on the strength of their relationships [26].

The escalating complexity, often addressed through evolutionary computation, is evident in various software implementations, including both single-objective [3,16] and multi-objective [18,27] approaches. Pioneering methodologies aim to enhance the structure of software systems. Optimisation of subsystems extends to diverse attributes, such as classes, methods, and variables.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
S. Smith et al. (Eds.): EvoApplications 2024, LNCS 14634, pp. 240–258, 2024.
https://doi.org/10.1007/978-3-031-56852-7_16

Methodological advancements now consider type-based dependence analysis [22], multi-pattern clustering [8], and effort estimation [32]. These efforts explore pre-processing and post-processing improvements alongside optimisation strategies.

The modularisation of software, mainly through heuristic search and evolutionary computation methodologies, extensively incorporates graph theory and data clustering. Academic works commonly use graph representations of software systems, employing data clustering for nodes and implementing algorithms to assess cluster quality [2,19,37]. Despite graph creation not inherently enhancing software engineers' understanding of architecture structure, language-independent graphs can focus on specific relationships or entire systems [10,30,35]. Clustering arrangements can be portrayed through various methods, such as a one-dimensional vector, a two-dimensional cluster-based structure, or a one-dimensional constrained representation known as a restricted growth function, which, despite its constraints, exhibits distinctive properties [7]. Clustering arrangement measurement typically addresses cohesion and coupling, striving for optimal cohesion within clusters and minimal coupling between clusters, fostering the creation of clearly defined groups [2].

1.2 Motivation

A recent study conducted by our research group explores varied representations for clustering arrangements and different starting points, providing insights into the search space of software systems [25]. The study highlights the list-of-lists representation as the most robust, emphasising its significance in problem-solving. Notably, the paper suggests that the starting point choice is inconsequential, as various representations converge towards similar outcomes regarding final fitness, especially one and two-dimensional list-based ones.

This paper is motivated by exploring converging results based on starting points. Our primary objective is to determine whether alternative starting positions can replicate or potentially improve previous findings. If diverse starting positions tend to converge toward a similar region in the search space, we aim to uncover the reasons behind this convergence. Is there a basin of attraction leading to a potential global optimum solution, or do these methods unintentionally get stuck in closely adjacent local optima?

In recent years, a discernible research gap has emerged in clustering arrangement representation and software graph representation. Additionally, up to the present time, there is a notable absence of publications in the field of software modularisation specifically dedicated to addressing the concept of starting points. While we recognise that meta-heuristics, such as Iterated Local Search [21], can generate seeded starting points based on previous experimental iterations, our reference pertains to the primary initial search, distinct from subsequent iterations.

Building upon this motivation, we aim to explore innovative approaches for generating starting points that surpass the performance of previous experiments. If our findings suggest the existence of a basin of attraction, our goal is to devise more efficient methods to reach this point faster than conventional approaches.

However, even if the evidence points in a different direction, our overarching objective is to develop a more efficient method for navigating and exploring the search space.

This paper focuses on enhancing software system clustering by integrating graph partitioning techniques with seeded search methods applied to graph-based representations. Situated within Search-Based Software Engineering, our research particularly centres on software modularisation. To achieve our goal, we begin with a domain background, introduce innovative concepts, outline our experimental procedure, and present our results.

2 Related Work

2.1 Bunch and Munch

Exploring software modularisation can be achieved using tools such as Bunch [23] and Munch [4] [5]. Bunch, developed by Mancoridis et al., combines a Steepest Ascent Hill Climbing (SAHC) and Genetic Algorithms for improved clustering arrangements [23,24]. On the other hand, Arzoky et al., Munch employs Random Mutation Hill Climbing (RMHC) for enhanced performance and ease of implementation [4]. Both strategies use different fitness functions - Bunch utilises the *MQ* fitness function, while Munch employs *EVM* and *EVMD* [4,24,34]. Despite employing different measurement strategies, MQ, EVM and EVMD yield similar clustering results [17]. However, the exhaustive nature of Bunch may hinder performance when runtime is a critical consideration.

2.2 Starting Points and Search Space

In the context of a heuristic-search-based clustering problem, the quest for optimal solutions necessitates delving into the search space, which comprises all conceivable arrangements of a clustering configuration. This exploration entails generating an initial clustering arrangement known as the starting point. Subsequently, through mutation (searching), this arrangement is modified and compared to the graph representation of the software. The goal is to enhance the clustering of nodes that demonstrate robust relationships. Before embarking on a search, a crucial decision lies in determining the optimal starting point for seeking an improved clustering arrangement.

Several starting points are available when searching for local optima, which, in our context, represents the nearest approximation to the optimal clustering arrangement that maximises the cohesion of each cluster within the search space. We provide three illustrative examples: we can cluster all nodes individually for maximum coupling (Fig. 1), together for maximum cohesion (Fig. 2), or randomly (Fig. 3).

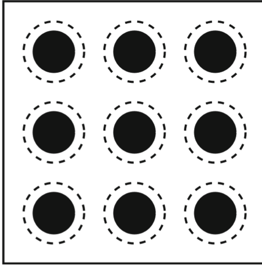

Fig. 1. Independent

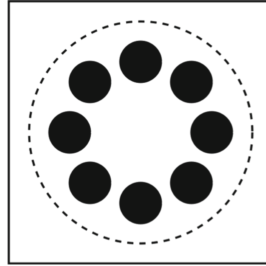

Fig. 2. All In One

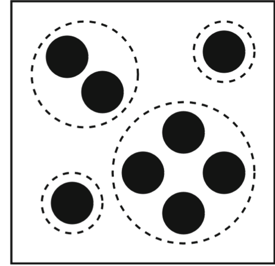

Fig. 3. Random

3 Research Questions

We aim to address research inquiries regarding our endeavour to discover improved starting points for software modularisation. We aim to uncover more effective strategies for achieving optimal outcomes. In this paper, we outline the following research questions that we intend to investigate:

1. What is the performance difference between graph-partitioned clustering arrangements and randomly generated ones when applied to large and small software systems?
   (a) When using hill climbing with various initial clustering arrangements on the same software system, do the solutions converge to similar outcomes or do disparities persist?
   (b) How do the runtimes of searches using graph partition and randomly generated clustering arrangements vary, and are there trade-offs between runtime and solution quality?
2. Is there a significant disparity between the Weighted Kappa[1] values of the final clustering arrangements and a gold standard[2], and what is the nature of this comparison?

Initially, we aim to evaluate whether the graph-based initial clustering arrangements result in enhanced outcomes compared to randomly generated configurations through Munch. We aim to contrast the clustering patterns derived from graph partitioning with those generated randomly across software systems of varying sizes. Alongside assessing our fitness function, we analyse the documented improvements at the final iteration. This entails identifying the convergence point and scrutinising the runtime of the search, which encompasses both the initialisation of the starting configuration and the subsequent search process. By possessing information about the ultimate fitness value and the corresponding

[1] Weighted Kappa is employed to assess the similarity of clustering arrangements and is applied in Sects. 5.3, 6, and 7.

[2] A gold standard represents the theoretical best solution for a given problem, a rarity in real-world datasets where it is seldom known.

iteration when it is achieved, we aim to discern the genuine impact of the initial clustering configurations on the search dynamics. We aim to determine whether specific clustering arrangements contribute to a faster convergence, enabling us to refine our search methodology for reaching the convergence point earlier and mitigating the risk of potential time loss.

In addition to assessing the effectiveness of our initial clustering configurations based on fitness, convergence, and runtime, we also evaluate the final clustering arrangements against gold standards using Weighted Kappa (WK) [1]. WK serves as a measure of agreement between two clustering arrangements, explicitly focusing on modularisation. As the WK values increase, the level of agreement between the two solutions also rises. A WK value 1 signifies identical clustering arrangements, while 0 indicates empirical dissimilarity. A WK value of 0.5 or higher indicates a robust structural similarity between the two clustering configurations. We opt for WK over other methods, such as Adjusted Rand [29], due to its ease of implementation, longstanding presence in the field, and well-established interpretability/quality scale. The authors also note that WK and Adjusted Rand are identical.

4 Methods

Our focus now shifts towards the methodologies aligned with our exploration of optimal starting points for software modularisation. We present our selected search method and detail our implementation of graph partitioning designed to yield appropriate starting points.

4.1 Munch

As previously indicated, software modularisation is characterised as a heuristic-search-based clustering problem. Therefore, our initial consideration lies in devising a strategy for heuristic search before delving into the discussion of our implementation of graph partitioning for generating starting points. We adopt a reverse-engineered adaptation of Arzoky et al.'s Munch to address this [5]. This adaptation has been enhanced to afford us the flexibility to determine the commencement of our exploration and the nature of our search strategy. We will now delve into an exploration of the various components that constitute Munch.

Foremost, Munch uses Module Dependency Graphs (*MDG*) as our graph-based representation of software systems. As defined by Mancoridis et al., *MDGs* illustrate subsystem connections to gauge relationships between components. In the context of our current research, we designate the nodes of *MDG* as software classes, and the edges represent interconnected relationships. *MDGs* prove versatile, capable of describing software structure over time or facilitating the segmentation of extensive software systems for enhanced comprehension. Let *MDG* M be an n by n symmetric binary matrix, where a 1 at row x and column y (M_{xy}) indicates a relationship between software components x and y, and 0

indicates that there is no relationship. To avoid confusion throughout this paper, MDG and graph are considered synonymous.

$$M_{xy} = \begin{cases} 1 & \text{if a relationship exists between } x \text{ and } y \\ 0 & \text{otherwise,} \end{cases}$$

For Munch, we adopt a list-of-list-based cluster representation based on its ease of implementation. A list-of-list clustering arrangement (C) is defined as a list ($[C_1, ..., C_k]$), with each subset list/cluster (C_i) containing $1, 2, ..., n$ elements. These subsets must be non-empty ($C_i \neq \emptyset$), and they should not share any common items ($C_i \cap C_j = \emptyset$) for different subsets. Effective optimisation problem-solving requires consideration of the search space, exploration strategy, and fitness function. Equation 1 illustrates all possible ways to partition C_k clusters containing n elements. Note that $1 \leq k \leq n$. We justify opting for lists over sets in the implementation, emphasising the advantages of simpler implementation and reduced computational complexity, particularly in scenarios involving non-indexed sets. Since each cluster and cluster element requires indexing, the search space aligns with Eq. 1, deviating from the set nature characterised by Bell(n).

$$\sum_{k=1}^{n} \left(\frac{n!}{k! \cdot (n-k)!} \cdot k! \right) \tag{1}$$

Before delving into the search strategy of Munch, it is essential to define the fitness function. The primary goal of a search strategy is to uncover a clustering arrangement that most effectively aligns with the ideal modular structure of the software system. The assessment entails analysing the subsets $[C_1, ..., C_k]$, where the elements (C_i), representing $1, 2, ..., n$, illustrate their relationships within the *MDG*. To avoid confusion, we will refer to the subsets as clusters.

For our replication of Munch, it is unsurprising we introduce *EVM* as our selected fitness function. We opt for *EVM* over Bunch's *MQ* due to its demonstrated robustness against noise and suitability for real-world software systems, as substantiated by research [17]. When provided with an arrangement C and an *MDG*, *EVM* evaluates and scores each cluster by considering the number of intra-relationships in the *MDG*. To prevent any potential confusion, we establish the definition of *EVM* as the aggregate of individual cluster scores, denoted as *SubEVM* (refer to Eqs. 2 and 3). *EVM* aims to maximise the score of relationships within a specified clustering arrangement. However, a potential drawback exists, as *EVM* may mistakenly assign high scores to clustering arrangements with high cohesion. Even minor adjustments to a solution can significantly enhance its fitness.

$$EVM(C, MDG) = \sum_{i=1}^{k} SubEVM(C_i, MDG) \tag{2}$$

$$SubEVM(C_i, MDG) = \sum_{a=1}^{|C_i|-1} \sum_{b=a+1}^{|C_i|} (2M(C_{ia}, C_{ib}) - 1) \tag{3}$$

To enhance the efficiency of Munch, we incorporate Arzoky et al.'s *EVMD*. This method generates a score aligning with *EVM* by integrating past *EVM* outcomes and determining the new result based on the classes designated for exchange. It demonstrates computational efficiency by computing the new fitness before implementing any modifications. Throughout this paper, we choose to utilise *EVM* as a collective term, encompassing both *EVM* and *EVMD*, to prevent potential confusion in future discussions about *EVM*.

Concerning the mentioned modifications, the inclusion of *EVMD* enables the execution of "Try/Do Moves." This variant of Small-Change, involving the random mutation of clustering arrangements, reduces computational overhead by initially testing the result of a small change (Try Move) before actual implementation (Do Move). To effectively utilise *EVMD*, the small-change process is limited to two elements simultaneously.

Finally, our focus shifts to the heuristic search. As mentioned earlier, Arzoky et al.'s Munch primarily employ RMHC as its heuristic search method. Despite implementing the ability to alter the heuristic search in our Munch, we opt to persist with RMHC. This choice is motivated by its reliability, ease of implementation, and superior performance compared to stochastic heuristics, such as SAHC. Below, we present Algorithm 1, elucidating how Munch searches for enhanced clustering arrangements. For practical reasons, we choose to employ EVM in the pseudocode example, even though we leverage Arzoky et al.'s EVMD fitness function to enhance performance:

Algorithm 1. Munch

```
 1: function MUNCH(Iterations, MDG)
 2:     Let C be a clustering arrangement              ▷ Random or Seed Starting Point
 3:     Let F = EVM(C, MDG)                            ▷ Current Fitness
 4:     for i = 1 to Iterations do
 5:         Let C′ = C                                 ▷ Copy of C
 6:         Choose two random clusters X and Y (X ≠ Y) from C,  ▷ Move Operator
 7:         Move a random variable from cluster X to Y in C′    ▷ Move Operator
 8:         Let F′ = EVM(C′, MDG)                      ▷ New Fitness
 9:         if F′ ≥ F then                             ▷ Compare Fitness
10:             Let C = C′, F = F′                     ▷ Continue using best Solution
11:         end if
12:     end for
13:     return C                                       ▷ Output is C
14: end function
```

4.2 Graph Partitioning

So far, we have established the importance of graphs and clustering arrangements regarding software modularisation. Now, we focus on using the structure graphs to discover new clustering arrangement starting points. Specifically, our

focus shifts to the Fiedler Vector [12]. This vector is linked to the second smallest eigenvalue, the Fiedler Eigenvalue, of a Laplacian Matrix [13]. Denoted as $L_{n \times n}$, a Laplacian Matrix is defined as $L = D - A$ where D represents the degree matrix of A which represents the connections between nodes [9]. In this context, $A \equiv MDG$. The Fiedler Vector is distinctive in its capability to enable a nearly perfect binary split of any given matrix. With this characteristic in mind, we have developed a tool that generates starting points through the recursive decomposition of graphs until no more Fiedler Vectors can be produced.

We generate a tree structure to facilitate the recursive decomposition of input graphs. The root of the tree is our input software graph and clustering arrangement. The clustering arrangement must begin with all nodes placed in a single cluster. This initial cluster will be subsequently split alongside the graph, ultimately leading to our final clustering arrangement, representing a fully decomposed software graph.

Leveraging our understanding of the Fiedler Vector, we identify the Fiedler eigenvalue of its attributed graph at each tree node, deduce its associated eigenvectors, and establish a well-balanced, binary-split graph partition. Simultaneously, we split the associated cluster with each partition, ensuring a one-to-one relationship between the subgraph's nodes and the associated cluster concerning the root MDG. This approach allows us to maintain traceability as we proceed with the decomposition. The new branches that emerge from the root node are reintroduced into a recursive function that continues to iterate until it identifies all possible partitions.

4.3 Starting Points

After generating a tree, we have two starting point approaches. Algorithm 2 illustrates the initial method for creating a "Leaf" arrangement. We gathered all leaf nodes from the tree, identified by their lack of children. Subsequently, we arrange these leaf nodes in ascending order based on SubEVM (see Eq. 3) and then incorporate nodes with unique clusters into our clustering arrangement. We organise all leaf nodes in ascending order to prevent branches from becoming disconnected at different depths, possibly leading to duplicate values. In an ideal scenario, all leaf nodes, regardless of depth, should be unique, and therefore, we incorporate this logic for peace of mind.

Algorithm 3 exemplifies our alternative approach to constructing a clustering arrangement. In this method, we recursively traverse the tree, evaluating the cohesion of each node in comparison to its children. This ensures the creation of a clustering arrangement containing all unique values, emphasising the highest possible cohesion within the context of the MDG for a given tree. We refer to this starting point as our "Max" arrangement.

Apart from Leaf and Max, our modified version of Munch can generate clustering arrangements randomly distributed uniformly, denoted as "Random." Due to publication constraints, we abstain from delving into the intricacies of this method. In summary, a "uniformly distributed random" arrangement is defined by a clustering setup generated through the utilisation of Bell Numbers, Stirling

Algorithm 2. BuildLeaf

```
1: function BuildLeaf(root)
2:     Let leafNodes be a stack of all leaves of root
3:     Sort leafNodes in descending order of SubEVM                ▷ see Eq. 2
4:     Let C be empty clustering arrangement
5:     while leafNodes is not empty do
6:         Let node be popped leafNodes                  ▷ Pop top node from stack
7:         if node is not in C then
8:             Let C = [C [node]]   ▷ Add unique cluster (node) to arrangement (C)
9:         end if
10:    end while
11:    return C                                                   ▷ Output is C
12: end function
```

Algorithm 3. BuildMax

```
1: function BuildMax(root)                                   ▷ Start Recursion
2:     Let C = empty clustering arrangement
3:     Let C = populateArrangement(root, C)
4:     return C                                                   ▷ Output is C
5: end function
6: function populateArrangement(node, C)                  ▷ Recursive Function
7:     if node has children then
8:         Let Fp be SubEVM of parent node                        ▷ See Eq. 2
9:         Let Fc be sum SubEVM of children                        ▷ See Eq. 2
10:        if Fp > Fc then            ▷ If the SubEVM of parent node is greater
11:            Let C = [C [node]]                ▷ Add node to arrangement C
12:        else                                          ▷ Continue Recursion
13:            Let populateArrangement(left child of node, C)
14:            Let populateArrangement(right child of node, C)
15:        end if
16:    else
17:        Let C = [C [node]]                   ▷ Add leaf node to arrangement C
18:    end if
19:    return C                                                   ▷ Output is C
20: end function
```

Numbers of the Second Kind [20,33,36], and their interconnected relationships [11].

5 Experimental Setup

Before presenting the Munch results of our graph partitioning tool, we need to establish an empirical framework.

5.1 Graph Collection and Pre-processing

First, we must collect software systems. Throughout our research, we developed a specialised tool that extracts open-source software systems using GitHub's RESTful API [14]. GitHub is our platform of choice for several compelling reasons. With a substantial user base exceeding 94 million developers, a continuously growing number of 52 million open-source repositories, and a cumulative total of 413 million contributions [15], we have access to a wide and diverse range of graphs.

Collecting and forming these graphs is often neglected in academic literature, creating a challenge in determining the authenticity of these systems - whether they are genuinely open-source, artificially generated, or specific to certain industries. Generating MDGs requires understanding the relationships between each class within a given software system. This can be achieved using software metric tools such as SciTools Understand [31], which provide pairwise relationships to build a symmetric graph. After our extractor downloads the desired software system, we manually process each system using SciTools Understand. Future efforts will explore using GitHub's TreeSitter parsing system [6] to automatically generate MDGs.

In this experiment series, we collect 50 "Small" open-source MDGs with class counts from 100 to 300, chosen based on relevance and high popularity ("stars") using the GitHub API. Due to storage constraints and the laborious manual MDG creation, we aim to develop an automated MDG generator, contemplating additional storage allocation pending study outcomes. Additionally, we have five "Big" MDGs (class counts: 1000 to 1500) sourced from prior research and industry collaboration, allowing exploration of size and characteristic-based result variations. Refer to Appendix A for a detailed breakdown. The terms "Small 50" and "Big 5" distinguish the two MDG groups in this paper.

5.2 Experiment Setup

Our experiments are described as follows. First, we collect Munch results for each MDG using all starting point combinations and iterations, as outlined below. Secondly, we collect Gold Standard results involving high-iteration/high-fitness outcomes to compare with our initial experiments. Finally, we analyse the results and present our findings concerning our outlined Research Questions.

1. For each experiment, for every graph (Small 50 and Big 5):
   (a) Select one of three starting points (Leaf, Max, Random)
   (b) Select one of three iterations (10k, 100k, 1m)
   (c) Run Munch
   (d) Document final iteration statistics and associated clustering arrangement
   (e) Repeat Steps a-c 250 times
2. Repeat Step 1 until all starting points and iterations are explored.

For our gold standards, we generate a Random starting point for each graph and run Munch for 100 million iterations, collecting the same information as

in our initial experiments. We repeat the process 250 times to ensure that we compare our initial experiment clustering arrangements to an absolute-best gold standard. Although conducting more iterations would have been preferable, it was impractical due to the extended runtime, taking several days per graph. To streamline experimental runs with our chosen iteration increments, we implemented parallel thread management, allowing multiple instances of Munch to run concurrently while optimising CPU and memory usage.

5.3 Data Collection and Analysis

We gather data on the fitness scores of the ultimate clustering configurations, pinpoint the convergence point (the last iteration demonstrating improved fitness), and gauge the runtime. Furthermore, we document the final clustering configurations into text files. Employing these files, we have crafted a bespoke tool to methodically evaluate the WK between the ultimate configurations derived from our initial points in contrast to our gold standards.

We have compiled a dataset of 275,000 files, combining the initial experiment results and gold standards. To enhance the manageability of these results for analysis, we employ MS Access, MS Excel, and Python for data processing. Due to the extensive volume of results and constraints in page space, our principal methodology involves computing averages across all data. Additionally, we streamline our findings by identifying and formatting the optimal results, providing a count of these instances per starting point type, thereby highlighting the suitability of each.

6 Results

Initially, we present RMHC results for each starting point category across selected iterations. Our research evaluates the performance of diverse starting points in searches across graphs of varying sizes, considering fitness, convergence, and runtime. The goal is to identify similarities or disparities in these aspects based on our predefined research questions. When implemented on large and small software systems, the performance differences among Leaf, Max, and Random are apparent in Tables 1[3] and 2[4]. Max consistently demonstrates superior fitness across iterations, as evidenced by the average final fitness values obtained from our three starting points over the specified iterations.

Table 3[5] details the average final convergence statistics across iterations, indicating the iteration where improvement was observed. A strong resemblance between the average fitness and convergence strongly implies a correlation, potentially indicating a basin of attraction where all solutions converge. Compared to Random in Tables 1 and 2, Leaf and Max achieve final fitness levels more rapidly across all iterations, notably enhancing results. For smaller graphs,

[3] Values formatted bold in Table 1 signify the highest average final fitness.
[4] Values formatted bold in Table 2 signify the highest average final fitness.
[5] Values formatted bold in Table 3 signify the shortest average convergence point.

Table 1. Average Final Fitness using Starting Point by Iterations

Size	10k			100k			1m		
	Leaf	Max	Random	Leaf	Max	Random	Leaf	Max	Random
102	60.664	**60.972**	60.516	72.660	**73.032**	72.544	73.248	73.236	**72.984**
105	**56.060**	56.016	56.052	**68.024**	67.696	67.844	**69.356**	69.192	69.344
112	88.616	**90.116**	87.532	99.512	**100.000**	99.920	101.040	**101.268**	101.108
119	66.968	**68.096**	66.204	80.100	**80.224**	80.168	80.960	**81.048**	80.900
123	99.516	**102.200**	98.068	**115.932**	115.688	114.896	117.808	**117.908**	117.660
127	82.340	**89.820**	79.452	105.680	**106.072**	105.272	107.216	**107.556**	107.040
129	81.796	**86.764**	80.500	98.544	**99.196**	98.848	99.800	99.856	**100.072**
130	37.612	**37.828**	35.592	53.016	53.064	**53.204**	53.832	53.660	**53.968**
135	97.748	**105.836**	96.864	120.684	**120.796**	120.636	122.200	122.052	**122.228**
135	101.480	**114.520**	100.760	133.832	**134.052**	133.908	**135.168**	134.920	135.000
141	74.428	**77.876**	71.392	88.224	88.552	**88.832**	89.832	**89.952**	89.928
145	**51.880**	51.856	47.848	**65.812**	65.624	65.672	**66.468**	66.380	66.372
151	105.948	**115.840**	102.016	**137.616**	137.008	137.224	139.276	139.132	**139.324**
158	84.080	**90.924**	76.424	112.664	112.532	**112.828**	115.260	115.292	**115.368**
161	85.580	**90.444**	82.204	**115.968**	114.972	115.748	118.912	118.848	**119.232**
163	83.980	**92.048**	73.400	110.248	**110.640**	109.728	**112.864**	112.780	112.192
164	79.768	**80.568**	75.164	105.464	**105.940**	105.408	107.884	107.928	**108.020**
169	42.464	**45.196**	35.432	73.156	**73.528**	73.212	75.316	**75.424**	75.372
172	86.924	**94.396**	82.732	123.580	122.596	**124.684**	128.760	128.228	**129.076**
172	100.636	**116.444**	91.548	137.196	**137.936**	137.168	**141.620**	141.372	140.928
172	15.804	**16.088**	11.796	**41.640**	41.488	41.372	44.280	44.388	**44.464**
175	80.768	**84.536**	71.776	**125.328**	125.196	125.108	133.864	133.884	**133.948**
175	108.704	**125.236**	101.228	164.620	**166.920**	164.512	171.248	**171.840**	171.032
176	82.360	**96.328**	72.364	112.776	112.844	**113.164**	115.760	**115.764**	115.688
179	101.700	**123.148**	93.196	148.364	148.160	**148.752**	151.900	151.336	**151.940**
198	98.008	**114.484**	84.572	139.196	**140.164**	139.192	144.092	144.276	**144.552**
199	90.328	**99.060**	83.392	153.600	150.504	**153.812**	166.760	**167.100**	166.664
200	91.704	**104.004**	70.864	128.124	**128.336**	127.912	131.672	131.268	**131.832**
206	113.880	**137.076**	103.772	182.220	**184.900**	181.952	191.008	**191.616**	190.964
208	62.528	**69.752**	52.000	127.376	**127.772**	127.512	130.656	**130.760**	130.724
209	82.896	**90.680**	59.672	131.196	**131.596**	129.776	137.768	**138.024**	137.244
210	65.216	**77.376**	48.576	102.340	**103.320**	101.980	106.336	**106.576**	106.404
210	76.324	**88.716**	54.560	116.984	**117.444**	116.688	118.224	**118.272**	118.060
211	66.624	**73.748**	44.256	113.068	113.320	**113.408**	120.872	120.880	**122.092**
214	118.440	**132.280**	100.060	187.688	**188.472**	187.412	198.188	**198.980**	198.084
216	43.364	**52.080**	47.336	120.720	**121.368**	120.732	126.764	126.764	**126.780**
224	110.880	**140.856**	89.708	184.596	**187.608**	184.308	197.920	**198.212**	198.144
233	92.680	**120.196**	75.740	173.608	**175.156**	172.776	**184.992**	184.808	184.600
234	89.564	**100.176**	57.732	143.928	**144.152**	142.660	**150.776**	150.756	150.652
234	111.308	**127.900**	84.244	184.216	**186.116**	183.916	**196.420**	196.260	195.996
235	93.872	**108.148**	65.808	158.052	**160.152**	158.588	172.260	**173.060**	172.596
240	82.872	**95.224**	54.924	148.992	**151.108**	147.784	159.104	159.128	**159.236**
240	126.100	**143.636**	98.224	**214.848**	213.208	213.488	234.844	**235.044**	234.472
242	108.404	**139.252**	76.388	177.708	**179.356**	177.092	**186.904**	187.304	187.280
246	72.876	**80.716**	41.404	139.548	**139.996**	139.340	148.820	148.524	**148.876**
252	99.484	**110.880**	61.124	159.868	**161.400**	159.000	174.656	**175.284**	174.616
252	134.832	**171.012**	108.420	243.296	**254.296**	242.448	279.192	**281.980**	278.472
258	123.152	**157.012**	94.904	208.428	**208.740**	206.764	**227.840**	225.928	227.144
264	114.188	**168.672**	72.400	227.132	**229.244**	226.552	239.488	238.788	**239.720**
294	131.708	**182.384**	75.388	251.476	**255.352**	249.916	276.912	276.692	**276.972**
Count	2	**48**	0	8	**34**	8	10	**20**	**20**

Table 2. Average Final Fitness using Starting Point by Iterations

Size	10k			100k			1m		
	Leaf	Max	Random	Leaf	Max	Random	Leaf	Max	Random
1037	170.924	**329.764**	–1360.344	345.940	**471.344**	123.476	731.892	**755.984**	737.256
1164	163.252	**276.048**	–1853.428	286.364	**376.868**	–188.728	588.528	**606.000**	589.680
1311	87.880	**225.908**	–2152.008	284.280	**400.460**	–297.580	681.848	**719.780**	675.820
1440	124.664	**283.652**	–2553.976	351.456	**479.476**	–520.688	741.544	**789.288**	708.468
1441	54.932	**146.112**	–2648.656	230.028	**304.932**	–718.304	543.324	**571.064**	493.336
Count	0	**5**	0	0	**5**	0	0	**5**	0

Table 3. Convergence Statistics

Iterations	Starting Point	Small 50			Big 5			Count
		Min	Max	Avg	Min	Max	Avg	
10k	Leaf	8970.444	9889.524	9560.283	9670.992	9887.296	9816.322	0
	Max	**8892.732**	**9873.904**	**9430.464**	**9634.588**	**9876.404**	**9799.476**	**6**
	Random	9129.948	9917.304	9710.809	9971.996	9977.348	9974.834	0
100k	Leaf	**53996.848**	96390.908	80512.020	98831.280	99120.060	98980.680	1
	Max	54158.648	**95036.248**	**79139.873**	**98552.824**	**99091.176**	**98786.134**	**5**
	Random	54598.832	96333.324	80817.637	99769.868	99858.020	99828.650	0
1m	Leaf	84590.520	**691299.112**	375566.554	989895.556	993509.896	991965.960	1
	Max	**77164.980**	706030.088	**368672.253**	**987032.192**	**991801.380**	**989805.248**	**5**
	Random	85098.216	692030.048	377192.918	990564.152	994979.076	993397.874	0

convergence is reached well before the considered iterations. Although final iterations align for smaller graphs, more iterations could enhance the likelihood of reaching local optima in larger datasets.

In contrast to average final fitness and convergence, Table 4[6] highlights cumulative average runtimes presented for each start and subsequent search at various iterations, measured in milliseconds. Notably, these reported runtimes represent summed average runtimes, excluding additional computational overhead related to data I/O. While Random clustering allows faster processing, the overall statistical significance of runtimes is debatable. This prompts consideration of potential trade-offs between runtime efficiency and solution quality.

Table 5[7] displays WK results, juxtaposing clustering configurations resulting from our initial starting points against gold standards. In multiple statistics and iterations, Leaf and Max consistently surpass Random. The notable closeness between Max results and their corresponding gold standards in smaller graphs contrasts the generally low agreement observed for larger graphs.

[6] Values formatted bold in Table 4 signify the shortest runtime in milliseconds.
[7] Values formatted bold in Table 5 signify the highest Weighted Kappa agreement.

Table 4. Average sum of runtime in milliseconds

Iterations	Small 50			Big 5		
	Leaf	Max	Random	Leaf	Max	Random
10k	298.797	332.881	**185.662**	24.231	**24.180**	199.044
100k	792.818	833.359	**408.770**	**119.465**	120.744	259.044
1m	5389.751	5607.426	**2574.516**	996.988	1012.166	**673.654**
Count	0	0	**3**	**1**	**1**	**1**

Table 5. *WK* against Gold Standard Statistics

Iterations	Starting Point	Small 50				Big 5				Count
		Min	Max	Avg	StDev	Min	Max	Avg	StDev	
10k	Leaf	0.215	0.565	0.386	**0.092**	0.002	0.002	0.002	**0.000**	2
	Max	**0.241**	**0.609**	**0.440**	0.094	**0.040**	**0.156**	**0.093**	0.043	**6**
	Random	0.198	0.580	0.358	0.102	0.010	0.027	0.015	0.007	0
100k	Leaf	0.398	**0.847**	0.637	**0.097**	0.012	0.021	0.016	**0.005**	3
	Max	0.376	0.835	**0.643**	0.099	**0.059**	**0.176**	**0.117**	0.044	**4**
	Random	**0.402**	0.843	0.635	0.097	0.058	0.107	0.077	0.022	1
1m	Leaf	0.490	0.897	0.715	0.084	0.128	0.317	0.198	0.079	0
	Max	0.480	**0.904**	**0.715**	0.086	**0.211**	0.359	0.264	**0.060**	**4**
	Random	**0.497**	0.897	0.715	**0.083**	0.197	**0.386**	**0.267**	0.084	**4**

7 Summary of Main Findings

In summary, we aimed to show that graph-partitioning can generate starting points capable of improving the results of software modularisation. We encapsulate the findings to address the research inquiries in the following manner:

- Max starting point:
  - Attains the highest average fitness over 10k, 100k, and 1m iterations, with a pronounced emphasis on lower iteration counts.
  - Attains the highest count of average convergence across all iterations while sustaining the optimal average final fitness.
  - Attains the maximum average agreement (WK) with gold standards across 10k and 100k for both Small 50 and Big 5 graphs, highlighting noteworthy performance, especially in lower iterations.
- Leaf starting point:
  - Demonstrate fitness levels equal to or surpassing Random across all iterations, especially in the early stages
  - Surpasses Random with higher average fitness levels on large datasets at 10k and 100k iterations
  - Consistently exhibits faster convergence compared to Random.
- Random starting point:

- Shows a quicker average total runtime in milliseconds compared to Leaf and Max.
- Better suited for smaller datasets; however, an improvement over Max and Leaf necessitates higher iterations.
- Demonstrates greater resemblance to the gold standard than Max in larger systems at 1m iterations.

Distinct fitness variations emerge among Leaf, Max, and Random, with Max consistently outperforming over 10k, 100k, and 1m iterations, notably in Small 50 vs. Big 5 comparisons. Random outperforms Max and Leaf at 1 million iterations for large datasets. However, Max proves to be more suitable for average fitness and faster convergence across iterations and graph sizes. Since there are currently no guidelines for determining the number of iterations based on the size or properties of an MDG, the most prudent approach would be to initiate seeding with Max before executing Munch. WK comparisons show Max starting points yield higher average agreements, with potential improvements around 70% and significant opportunities at 90% agreement in 1m iterations. Thorough exploration is vital for understanding software system graph intricacies. Our commitment to accelerating software modularisation drives deeper exploration, with partition-based clustering performing significantly, especially at smaller iterations, making it compelling for future software optimisation.

8 Generalisability

This publication focuses on utilising graph partitioning for software modularisation. However, the application of graph partitioning for optimising initial positions can extend to other graph-based optimisation problems, contingent on the chosen fitness function. Although we prioritise EVM for its simplicity, other alternatives like MQ are viable. Our aim is to inspire exploration of graph partitioning for seeded optimisation.

9 Future Work

We plan to integrate our graph-based initial clustering with metaheuristics, specifically incorporating seeded starting points into the history of Iterated Local Search, as part of our ongoing investigation [28]. This initiative seeks to evaluate the potential improvement in the exploration of the search space and overall efficiency. Furthermore, our goals include delving deeper into software systems' search space, exploring graph structure, convergence prediction, and other avenues for enhancing software modularisation.

10 Appendix A

This Appendix showcases details about the software system MDG used in our experiments. Below, we showcase the following statistics for each software system:

1. ID
   - Each software system is assigned a unique identifier. We choose not to use the actual names of our software systems because our collection is sourced randomly from GitHub. These software system names can exhibit variation, and we intend to maintain professionalism and steer clear of potentially inappropriate names and software tools.
2. Nodes
   - Also known as vertices, these signify the number of software components (classes) within our Module Dependency Graphs (MDGs).
3. Edges
   - Denotes the number of relationships between software components.
4. Clustering Coefficient:
   - The extent to which nodes tend to cluster. A high score indicates a strong cohesion, while a low score indicates a higher coupling level. We present this statistic as these software systems exhibit remarkably low coefficients, indicating a high coupling level and a deficiency in the initial modular structure. There is potential here to investigate the nature of software structure over time, especially concerning the analysis of open-source software systems (Tables 6 and 7).

Table 6. "Big 5" Software MDG Statistics

Identification	Nodes	Edges	Avg Degree	Clustering Coefficient
1	1037	5470	5.274	0.000
2	1164	2072	1.780	0.000
3	1311	5630	4.294	0.000
4	1440	4889	3.395	0.000
5	1441	3058	2.122	0.000

Table 7. "Small 50" Open-Source Software MDG Statistics

Identification	Nodes	Edges	Avg Degree	Clustering Coefficient
01	102	312	0.061	0.007
02	105	257	0.047	0.002
03	112	436	0.070	0.007
04	119	343	0.049	0.002
05	123	440	0.059	0.004
06	127	409	0.051	0.004
07	129	357	0.043	0.002
08	130	225	0.027	0.001
09	135	387	0.043	0.002
10	135	510	0.056	0.004
11	141	333	0.034	0.001
12	145	274	0.026	0.001
13	151	595	0.053	0.002
14	158	422	0.034	0.001
15	161	413	0.032	0.001
16	163	414	0.031	0.001
17	164	686	0.051	0.002
18	169	387	0.027	0.001
19	172	609	0.041	0.002
20	172	591	0.040	0.002
21	172	357	0.024	0.000
22	175	737	0.048	0.004
23	175	749	0.049	0.003
24	176	371	0.024	0.000
25	179	467	0.029	0.001
26	198	552	0.028	0.001
27	199	1002	0.051	0.003
28	200	450	0.023	0.000
29	206	964	0.046	0.003
30	208	643	0.030	0.001
31	209	732	0.034	0.001
32	210	518	0.024	0.000
33	210	323	0.015	0.000
34	211	599	0.027	0.001
35	214	834	0.037	0.002
36	216	740	0.032	0.001
37	224	937	0.038	0.002
38	233	818	0.030	0.001
39	234	521	0.019	0.000
40	234	930	0.034	0.002
41	235	823	0.030	0.001
42	240	898	0.031	0.001
43	240	1115	0.039	0.002
44	242	836	0.029	0.001
45	246	672	0.022	0.001
46	252	1033	0.033	0.001
47	252	1591	0.050	0.004
48	258	1477	0.045	0.002
49	264	730	0.021	0.001
50	294	1275	0.030	0.001

References

1. Altman, D.: Skewed distributions. Practical statistics for medical research, pp. 60–63. Chapman & Hall, London (1997)
2. Arasteh, B.: Clustered design-model generation from a program source code using chaos-based metaheuristic algorithms. Neural Comput. Appl. **35**(4), 3283–3305 (2023)
3. Arasteh, B., Seyyedabbasi, A., Rasheed, J.M., Abu-Mahfouz, A.: Program source-code re-modularization using a discretized and modified sand cat swarm optimization algorithm. Symmetry **15**(2), 401 (2023)
4. Arzoky, M., Swift, S., Tucker, A., Cain, J.: Munch: an efficient modularisation strategy to assess the degree of refactoring on sequential source code checkings. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp. 422–429. IEEE (2011)
5. Arzoky, M., Swift, S., Tucker, A., Cain, J.: A seeded search for the modularisation of sequential software versions. J. Object Technol. **11**(2), 1–6 (2012)
6. Brunsfeld, M.: Tree-sitter. https://github.com/tree-sitter/tree-sitter (Accessed 1 Jan 2023)
7. Campbell, L.R., et al.: Restricted growth function patterns and statistics. Adv. Appl. Math. **100**, 1–42 (2018)
8. Chen, Y.T., Huang, C.Y., Yang, T.H.: Using multi-pattern clustering methods to improve software maintenance quality. IET Softw. **17**(1), 1–22 (2023)
9. Chung, F.R.: Spectral graph theory, vol. 92. American Mathematical Soc. (1997)
10. Corradini, A., König, B., Nolte, D.: Specifying graph languages with type graphs. J. Logical Algebraic Methods Programm/ **104**, 176–200 (2019). https://doi.org/10.1016/j.jlamp.2019.01.005, https://www.sciencedirect.com/science/article/pii/S235222081730233X
11. Devroye, L.: Sample-based non-uniform random variate generation. In: Proceedings of the 18th Conference on Winter Simulation, pp. 260–265 (1986)
12. Fiedler, M.: A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. Czechoslov. Math. J. **25**(4), 619–633 (1975)
13. Fiedler, M.: Laplacian of graphs and algebraic connectivity. Banach Center Publ. **1**(25), 57–70 (1989)
14. GitHub: Github advanced search (2023). https://github.com/search/advanced, (Accessed 1 Jan 23)
15. GitHub: Octoverse 2022: 10 years of tracking open source (2023). https://github.blog/2022-11-17-octoverse-2022-10-years-of-tracking-open-source/, (Accessed 1 Jan 23)
16. Gupta, N., Kumar, S., Gupta, V., Vijh, S.: Novel automatic approach using modified differential evaluation to software module clustering problem. SN Comput. Sci. **4**(6), 816 (2023)
17. Harman, M., Swift, S., Mahdavi, K., Beyer, H.: An empirical study of the robustness of two module clustering fitness functions, In: genetic and Evolutionary Computation Conference; Conference date: 25–06-2005 Through 29–06-2005, pp. 1029–1036. Assoc Computing Machinery (2005),
18. Kang, Y., Xie, W., Wang, X., Wang, H., Wang, X., Li, J.: Mopisde: a collaborative multi-objective information-sharing de algorithm for software clustering. Expert Syst. Appli., 120207 (2023)
19. Khan, M.Z., et al.: A novel approach to automate complex software modularization using a fact extraction system. J. Mathem. 2022 (2022)

20. Harper, L.H.: Stirling behaviour is asymptotically normal. Annals Math. Stat. **3**(2), 410–414 (1967)
21. Kramer, O.: Iterated local search. In: A Brief Introduction to Continuous Evolutionary Optimization. SAST, pp. 45–54. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-03422-5_5
22. Lu, K.: Practical program modularization with type-based dependence analysis. In: 2023 IEEE Symposium on Security and Privacy (SP), pp. 1256–1270. IEEE (2023)
23. Mancoridis, S., Mitchell, B.S., Chen, Y., Gansner, E.R.: Bunch: a clustering tool for the recovery and maintenance of software system structures. In: Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM 1999). Software Maintenance for Business Change'(Cat. No. 99CB36360), pp. 50–59. IEEE (1999)
24. 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: Proceedings. 6th International Workshop on Program Comprehension, IWPC 1998 (Cat. No. 98TB100242), pp. 45–52. IEEE (1998)
25. Maramazi, F., Odebode, A., Mann, A., Swift, S., Arzoky, M.: Intelligent systems and applications.In: Proceedings of the 2024 Intelligent Systems Conference (Intellisys) vol. 1. LNNS, vol. 822, pp. 470. Springer (2024)
26. Mitchell, B.S., Mancoridis, S.: Clustering module dependency graphs of software systems using the bunch tool. Nat. Sci. Found., Alexandria, VA, USA, Tech. Rep (1998)
27. Prajapati, A., Parashar, A., Rathee, A.: Multi-dimensional information-driven many-objective software remodularization approach. Front. Comp. Sci. **17**(3), 173209 (2023)
28. Ramalhinho-Lourenço, H., Martin, O.C., Stützle, T.: Iterated local search (2000)
29. Rand, W.M.: Objective criteria for the evaluation of clustering methods. J. Am. Stat. Assoc. **66**(336), 846–850 (1971)
30. Savić, M., Rakić, G., Budimac, Z., Ivanović, M.: A language-independent approach to the extraction of dependencies between source code entities. Inf. Softw. Technol. **56**(10), 1268–1288 (2014)
31. SciTools: Understand: The software developer's multi-tool (2023). https://scitools.com/, (Accessed 10 Nov 2023)
32. Tan, A.J.J., Chong, C.Y., Aleti, A.: Closing the loop for software remodularisation-rearrange: an effort estimation approach for software clustering-based remodularisation. arXiv preprint arXiv:2303.06283 (2023)
33. Temme, N.M.: Asymptotic estimates of stirling numbers. Stud. Appl. Math. **89**(3), 233–243 (1993)
34. Tucker, A., Swift, S., Liu, X.: Variable grouping in multivariate time series via correlation. IEEE Trans. Syst. Man Cybernet. Part B (Cybernet). **31**(2), 235–245 (2001)
35. Weiss, K., Banse, C.: A language-independent analysis platform for source code. arXiv preprint arXiv:2203.08424 (2022)
36. Weisstein, E.W.: Stirling number of the second kind (2002). https://mathworld.wolfram.com/
37. Yang, K., Wang, J., Fang, Z., Wu, P., Song, Z.: Enhancing software modularization via semantic outliers filtration and label propagation. Inf. Softw. Technol. **145**, 106818 (2022)