Evaluating the Effectiveness of Multi-level Greedy Modularity Clustering for Software Architecture Recovery

Hasan Sözer(✉)

Ozyegin University, Istanbul, Turkey
hasan.sozer@ozyegin.edu.tr

Abstract. Software architecture recovery approaches mainly analyze various types of dependencies among software modules to group them and reason about the high-level structural decomposition of a system. These approaches employ a variety of clustering techniques. In this paper, we present an empirical evaluation of a modularity clustering technique used for software architecture recovery. We use five open source projects as subject systems for which the ground-truth architectures were known. This dataset was previously prepared and used in an empirical study for evaluating four state-of-the-art architecture recovery approaches and their variants as well as two baseline clustering algorithms. We used the same dataset for an evaluation of multi-level greedy modularity clustering. Results showed that MGMC outperforms all the other SAR approaches in terms of accuracy and modularization quality for most of the studied systems. In addition, it scales better to very large systems for which it runs orders-of-magnitude faster than all the other algorithms.

Keywords: Software architecture recovery · Software architecture reconstruction · Reverse engineering · Modularity clustering · Empirical evaluation

1 Introduction

Software architecture documentation is an important asset for supporting program comprehension, communication and maintenance [16]. This documentation turns out to be usually incorrect or incomplete, especially for old legacy systems [10,24]. It is also very effort-intensive to recover such a documentation manually [14], which can quickly become infeasible as the software size and complexity increases.

Software architecture reconstruction [9] or recovery [21] (SAR) approaches have been introduced to recover software architecture documentation. These approaches essentially analyze dependencies among software modules to group them and reason about the high-level structure of a system. Inter-dependencies among software modules are usually represented with design structure matrices [11] or (un)weighted (un)directed graphs [9,23]. In addition to these different representations, SAR approaches mainly vary with respect to the types of dependencies considered and the types of clustering techniques employed.

In this work, we focus on recovering the high-level structural decomposition of a system based on code dependencies. In that respect, a recent empirical study [21] evaluated the effectiveness of four state-of-the-art SAR approaches and their variants as well as two baseline clustering algorithms. The study was conducted on five open source projects as subject systems, for which the "ground-truth" software architectures were manually recovered. Various types of dependencies extracted from the subject systems were used as input to evaluate their impact on the accuracy of SAR approaches. We used the same dataset for an evaluation of modularity clustering [4,28] as an alternative SAR approach.

Modularity clustering aims at decomposing a graph into cohesive components that are loosely coupled. This aim is aligned with the very basic modularity principle [26] followed in software design. Hence, it makes sense to apply this approach for SAR. In fact, there have been clustering techniques [23] introduced for balancing the tradeoff between coupling and cohesion. However, it was shown that the accuracy of these techniques is low and the utilized modularity metrics are subject to flaws [21]. In this study, we employ the Multi-level Greedy Modularity Clustering (MGMC) approach [25], which borrows metrics and heuristics from the physics literature [7,31]. MGMC combines two heuristics, namely *greedy coarsening* [7] and *fast greedy refinement* [31] to maximize a modularity measure. We evaluate the accuracy of MGMC and compare it with respect to those achieved with other SAR approaches. It was shown that some of these approaches scale to very large systems that contain 10 MLOC, whereas others not [21]. Therefore, runtime performance of MGMC is another important aspect to investiage. We defined the following two research questions based on these concerns:

– *RQ1*: How does the accuracy of MGMC compare to those of other SAR approaches when various types of dependencies are considered?
– *RQ2*: How does the runtime performance of MGMC compare to those of other SAR approaches?

We applied MGMC on dependency graphs regarding five open source projects. These graphs represent different types of dependencies extracted from the source code such as file inclusions and function calls. Then, we measured the quality of the clustering using the corresponding ground-truth architectures and two different metrics proposed before [23,36]. We compared these measurements with respect to the measurements previously reported [21] for the same projects, input files and metrics but for different SAR approaches. Results showed that MGMC outperforms all the other SAR approaches in terms of accuracy and modularization quality [23] for most of the studied systems. In addition, it scales better to very large systems for which it runs orders-of-magnitude faster than all the other algorithms.

This paper is organized as follows. We summarize the related studies on SAR and position our work in the following section. We introduce MGMC in Sect. 3. We explain the experimental setup in Sect. 4. We present and discuss the results in Sect. 5. Finally, in Sect. 6, we conclude the paper.

2 Background and Related Work

There exist many approaches [9] proposed for SAR, some of which are manual or semi-automated. In this study, we focus on approaches introduced for automatically recovering an architecture. The recovered architecture can be in various forms for representing various architectural views [16]. The majority of the existing techniques [21,29,30,33] aim at recovering a *module view* that depicts the structural design-time decomposition of a system [16]. Some of them focus on analyzing the runtime behavior for reconstructing execution scenarios [5] and behavioral views [27]. There are also tools that construct both structural and behavioral views [17,34]. In this work, we focus on SAR approaches that are used for recovering a high-level module view of the system.

SAR approaches also vary with respect to types of inputs they consume [9]. Some of them rely on textual information extracted from source code [8,15]. Many others use dependencies among modules, which are usually represented with design structure matrices [11] or (un)weighted (un)directed graphs [23]. These dependencies can be extracted from a variety of sources as well. For instance, a call graph extracted from the source code can be interpreted as a dependency graph, where each vertex represents a module (e.g., class) and each directed edge represents a dependency (e.g., method call) from the source vertex to the target vertex [23]. As another example, commonly accessed database tables (or other external resources) can be interpreted as (indirect) module interdependencies [2]. The goal of a recent empirical study [21] was to measure the impact of various code dependencies on the accuracy of SAR approaches. These dependencies were represented in the form of unweighted directed graphs, which were extracted based on variable accesses, function calls and file inclusions. We use the same types of dependencies in this work to extend that study with an evaluation of MGMC.

Finally, the employed clustering algorithm/technique is a major variation point among SAR approaches. There are many techniques proposed so far and these techniques have been compared with each other as well. However, an analysis of existing evaluations [21] show that results are not always consistent. In a recent study [13], nine variants of six SAR approaches were compared based on eight subject systems. The overall accuracy of all the evaluated approaches turned out to be low based on their consistency with respect to the ground-truth architectures collected for the subject systems. In that study, ACDC [35] was pointed out as one of the best approaches. In another study, the performance of LIMBO (Scalable Information Bottleneck) [3] was shown to be comparable to that of ACDC. There also exist a study [38] indicating that WCA (Weighted Combined Algorithm) [22] performs better than ACDC. However, in the most recent studies [13,21], ACDC turns out to be superior than others. Results may differ due to the use of different subject systems and assessment measures/criteria.

Bunch [23] employs a hill-climbing algorithm for maximizing modularization quality, while clustering a dependency graph. Its objective function is defined to balance the tradeoff between the cohesion of clusters and coupling among them. However, the best objective function value can be achieved by grouping all the modules in a single cluster [21]. Also, the accuracy of Bunch was shown to be low in recent empirical studies [21]. We adopt a different formulation of modularity in this study and also a different algorithm to maximize it. We previously used another variant of modularity clustering [12] for recovering software architectures of PL/SQL programs. In that approach, dependencies among PL/SQL procedures are extracted based on their common use of database tables. These dependencies are represented in the form of a hypergraph. This representation is converted to a weighted undirected graph, which is then partitioned to maximize modularity. However, that approach was dedicated for PL/SQL programs and its evaluation was based on a single case study. Moreover, it employed a different algorithm [6] to maximize modularity. The effectiveness of MGMC that we introduce in the following section has not been empirically evaluated as a SAR approach.

3 Multi-level Greedy Modularity Clustering

Given a graph $G(V, E)$, modularity clustering aims at grouping the set of vertices $V = \{v_1, v_2, ..., v_n\}$ into a set of k disjoint clusters $C_1, C_2, ..., C_k$ such that the modularity is maximized. The modularity is calculated based on Eq. 1 [28].

$$\mathcal{M} = \frac{1}{2m} \sum_{l=1}^{k} \sum_{i,j | v_i, v_j \in C_l} (w_{ij} - \frac{d_i d_j}{2m}) \tag{1}$$

In this equation, w_{ij} represents the weight of the edge between v_i and v_j, $d_i = \sum_{j \neq i} w_{ij}$ and $m = \frac{1}{2} \sum_i d_i$. In our dataset, the extracted dependency graphs are not weighted. Hence, w_{ij} can be either 1 or 0, representing the existence of a dependency between v_i and v_j or lack thereof, respectively. However, the objective function and the employed algorithms are generic and they can work on weighted graphs as well. We should also note that graphs are considered as undirected in this formulation. Two vertices, v_i and v_j are adjacent ($w_{ij} = w_{ji} = 1$) if either of these vertices depends on the other.

$\mathcal{M}$ captures the inherent trade-off in maximizing the number of edges among the vertices that take place in the same cluster and minimizing the number of edges among the vertices that take place in different clusters. We can see in Eq. 1 that w_{ij} values are summed up only for pairs of vertices that are in the same cluster. Therefore, decreasing the number of clusters and as such, increasing the size of each cluster is rewarded by taking more pairs into account. On the other hand, the value of w_{ij} will be 0 for pairs of independent vertices that are in the same cluster. Nevertheless, the penalty $\frac{d_i d_j}{2m}$ is paid for each such pair as well. The amount of penalty is proportional to the number of dependencies of these vertices to all the other vertices in the graph.

It was shown that finding a clustering of a given graph with maximum $\mathcal{M}$ is an $\mathcal{NP}$-hard problem [4]. Exact methods can not scale beyond graphs with a few hundred vertices [1,39]. Therefore, many heuristic algorithms have been proposed to address this problem. These are mainly proposed and elaborated in the physics literature [7,31]. MGMC is one of these and it combines two heuristics [25].

The first heuristic is *greedy coarsening* [7], which starts with singleton clusters and iteratively merges cluster pairs as long as the merge operation increases modularity. Hereby, a *merge priority* is assigned to each cluster pair, which determines the order of pairs to be merged at each step. It was empirically shown that the *Significance (Sig)* measure is an effective metric to quantify *merge priority* [25]. *Sig* for a cluster pair *(A,B)* is defined as follows.

$$Sig = \frac{\Delta\mathcal{M}_{A,B}}{\sqrt{deg(A) \times deg(B)}} \tag{2}$$

Hereby, $\Delta\mathcal{M}_{A,B}$ defines the amount of increase in modularity as a result of merging clusters A and B. The *deg* function provides the total weight of edges inside a given cluster.

The second heuristic is called *fast greedy refinement* [31]. This heuristic basically iterates over all the vertices in the graph and finds the *best* target cluster to move for each vertex. The *best* cluster is the one that leads to the largest modularity increase by moving the vertex to this cluster. Iteration stops when the modularity can not be improved further with any vertex movement.

The *coarsening* and *refinement* heuristics do not have to be applied in separate, sequential phases. Moving individual vertices after the completion of coarsening can lead to sub-optimal results. A densely connected group of vertices may not have a chance to move to another cluster because this would involve a series of vertex movements that degrade modularity. However, *refinement* can be applied at any level of the coarsening hierarchy in principle. An entire cluster can be moved rather than an individual vertex. This is the idea behind *multi-level refinement* [18,19], where the application of *coarsening* and *refinement* heuristics are interleaved. Intermediate coarsening results are saved as a *coarsening level* whenever the number of clusters is decreased by a certain percentage called the *reduction factor*. These intermediate results are embodied as a graph where vertices represent clusters obtained at the corresponding coarsening level. The refinement heuristic is applied to every level. It was empirically shown that modularity improves as *reduction factor* decreases; however, the amount of improvement becomes less significant when *reduction factor* incline below 50% [25].

The algorithm [28] we used in this study follows the steps and recommendations described above. The implementation of the overall greedy algorithm is discussed in [25]. Further details of the implementation together with pseudo codes of its various steps are provided in [28].

4 Experimental Setup

In this section, we describe our experimental setup including the properties of our dataset, SAR approaches being compared with MGMC and the evaluation criteria.

4.1 Subject Systems and the Dataset

Table 1 lists information about five open source projects, which were used as subject systems for a previous empirical study [21]. We used the same set of projects because their ground-truth architectures and module dependency information were available.

Table 1. Subject systems.

System	Version	LOC	# of files	Description
Chromium	svn-171054	10 M	18,698	Web Browser
ITK	4.5.2	1 M	7,310	Image Segmentation Toolkit
Bash	4.2	115 K	373	Unix Shell Command Processor
Hadoop	0.19.0	87 K	591	Data Processing Framework
ArchStudio	4	55 K	604	Architecture Development Tool

Table 2 lists the properties of our dataset. Hereby, the second column lists the number of clusters in the ground-truth architecture of each system. The following 3 columns list the numbers of dependencies extracted for 3 basic types of dependencies considered: *(i) Include* dependencies are established between two files if one of them declares that it includes the other. *(ii) Symbol* dependencies are established between two files if one of them makes use of a symbol that is defined in the other. A symbol can be a function or a variable name. *(iii) Function* dependencies constitute a subset of *Symbol* dependencies, just focusing on function calls between modules.

Types of *symbol* dependencies were further varied to observe their impact on the accuracy of SAR approaches. *(i) F-GV* captures function calls and global variables together. *(ii) S-NoDYB* represents *symbol* dependencies extracted by ignoring dynamic bindings. The values listed in Table 2 reflect this type of *symbol* dependencies. *(iii) S-CHA* takes dynamic bindings into account by analyzing the class hierarchies. *(iv) S-Int* is extracted by resolving dynamic bindings based on interfaces only. We used these dependency types in our evaluation. There are two other dependency types that were utilized in the previous empirical study [21], namely *transitive* and *module level* dependencies. We have not used these two since the corresponding dependency information was not available for most of the projects. Information regarding *Include*, *S-CHA*, *S-Int*, *S-NoDyB*, *Function* and *F-GV* dependencies was available for all the projects. One exception to this was the *Bash* project implemented in C, for which information regarding dynamic bindings could not be extracted. So, dependency information regarding *S-CHA*, *S-Int* and *S-NoDyB* variants is not available for this project. Dependency information regarding each type of dependency is represented in the form of an unweighted directed graph, so-called a *dependency graph*.

Table 2. Properties of the dataset [21].

System	# of clusters in the ground-truth architecture	# of various types of dependencies		
		Include	Symbol	Function
Chromium	67	1,183,799	297,530	123,422
ITK	11	169,017	75,588	16,844
Bash	14	2,512	2,481	1,025
Hadoop	67	1,772	11,162	2,953
ArchStudio	57	866	5,359	1,411

4.2 Architecture Recovery Approaches

We selected the same variants of SAR approaches, for which we took the results reported [21] regarding their accuracy on the same dataset we use. We only omitted two of these approaches, namely Architecture Recovery Using Concerns (ARC) [15] and Zone Based Recovery (ZBR) [8], which use textual information from source code as input. Results regarding these approaches were missing for dependency graphs that are used as input for MGMC. Most of the results were missing for ARC and ZBR also because they could not scale for large systems [21]. In particular, we included results regarding ACDC [35], two variants of Bunch [23], namely Bunch-NAHC and Bunch-SAHC, two variants of WCA [22], namely WCA-UE and WCA-UENM, and finally, LIMBO [3].

We also included results regarding K-means algorithm used as a baseline for comparison. There was a second baseline derived from the directory structure of the project [21]. However, we omitted that one since most of the corresponding results we missing, just like the case for ARC and ZBR.

4.3 Environment and Parameters

We used a laptop computer with Intel Core i7 1.80 GHz CPU and 16 GB RAM to run the experiments. We used the implementation of MGMC provided by Rossi [28], which is available online[1]. This implementation works on weighted

[1] http://apiacoa.org/research/software/graph/index.en.html.

undirected graphs. Hence, in our dataset directions are ignored and all the edge weights are assumed to be 1. We did not provide any of the optional parameters and as such, used the algorithm with its default parameter settings (i.e., *reduction factor* = 25%, *merge priority* = *Sig*).

Input files that store dependency graphs [21] conform to the Rigi Standard Format (RSF) format [32,37]. The clustering results should also be saved in this format to be provided to the implementations of metrics described in the following subsection. However, the input and output formats of the MGMC implementation do not conform to RSF. Hence, we developed programs to preprocess the input and postprocess the output. We did not include the time spent for input/output transformations in our measurements and just report the time elapsed during clustering. We run the algorithm 100 times to observe the variation in running time although the results do not change in these runs.

The reported results for Bunch variants and ACDC are calculated as the average of five runs due to the non-determinism of the employed clustering algorithms [21]. On the other hand, WCA variants, LIMBO and K-means take the number of clusters, k as input. Results reported for these approaches are averages of results obtained from multiple executions, where k is varied in each run. The values of k range from 20 clusters below to 20 clusters above the number of clusters in the ground-truth architecture with step size 5 [21].

4.4 Evaluation Criteria

We used two different metrics to evaluate MGMC and compare it with the other SAR approaches. The first one is the *MoJoFM* metric [36], of which the implementation is available online[2]. This metric is used for measuring the similarity between the recovered architecture and the ground-truth architecture. It has been shown to be more accurate than other representative measures and consistently been used in empirical studies on SAR [13,20,21]. The MoJoFM value for given two clusterings A and B is calculated as follows:

$$MoJoFM = (1 - \frac{mno(A,B)}{max(mno(\forall A, B))}) \times 100\% \tag{3}$$

Hereby, $mno(A,B)$ calculates the minimum number of *move* or *join* operations needed to transform A to B. On the other hand, $max(mno(\forall A, B))$ calculates the maximum $mno(A,B)$ possible for any A. High and low *MoJoFM* values indicate high similarity and high disparity between A and B, respectively.

There might be a lack of consensus on the ground-truth architecture by the domain experts. Hence, there might be multiple such architectures derived [21]. Moreover, the recovery process is by-and-large manual, and as such, error-prone. For these reasons, we used a second metric, namely *normalized TurboMQ* [21], which measures the quality of a clustering independent of any ground-truth architecture. This metric is defined based on the *Cluster Factor (CF)* that is calculated for each cluster, i as follows:

[2] http://www.cse.yorku.ca/~bil/downloads/.

Table 3. MoJoFM results for *Bash*.

Method	**Include**	**Symbol**	**Function**	**F-GV**
MGMC	64	52	57	54
ACDC	52	57	49	50
Bunch-NAHC	53	43	49	46
Bunch-SAHC	57	52	43	49
WCA-UE	34	24	29	30
WCA-UENM	34	24	31	30
LIMBO	34	27	22	22
K-means	59	55	47	46

$$CF_i = \frac{\mu_i}{\mu_i + 0.5 \times \sum_j(\epsilon_{ij} + \epsilon_{ji})} \tag{4}$$

Hereby, μ_i is the number of dependencies among the elements in cluster i. The term $\sum_j(\epsilon_{ij} + \epsilon_{ji})$ defines the sum of dependencies between elements in cluster i and all the elements residing in other clusters. *TurboMQ* measure basically adds up the CF values for all the clusters as shown in Eq. 5.

$$TurboMQ = \sum_{i=1}^{k} CF_i \tag{5}$$

It was observed that *TurboMQ* measure is biased towards architectures with large numbers of clusters [21]. Therefore, it is normalized with respect to the total number of clusters in the recovered architecture. This leads to the *normalized TurboMQ* metric, which we used in our study. The implementation of this metric is available online[3] as well.

We discuss the obtained results in the following section.

5 Results and Discussion

Results for each subject system are listed in Tables 3, 4, 5, 6, 7, 8, 9, 10, 11 and 12. The first and the latter five tables list results regarding the *MoJoFM* metric and the *normalized TurboMQ* metric, respectively. In the following section we first interpret these results to answer *RQ1*. Then, we evaluate the runtime performance as the focus of *RQ2*. We conclude the section with a discussion on threats to validity.

[3] https://github.com/hasansozer/Normalized-TurboMQ.

5.1 Accuracy of Modularity Clustering

Tables 3, 4, 5, 6 and 7 list the results for the *MoJoFM* metric. The first column lists the compared SAR approaches, which is followed by results regarding each type of dependency in the respective columns. The best score obtained by any of the SAR approaches for a particular type of dependency is highlighted in light gray. The best score overall is highlighted in dark gray. We can see from these results that the overall best scores are obtained with either ACDC or MGMC. We can also see that best scores per various dependency types are also attributed to these two techniques except a few cases. Overall, MGMC outperforms ACDC in approximately half of the cases.

Table 4. MoJoFM results for *ArchStudio*.

Method	**Include**	**S-CHA**	**S-Int**	**S-NoDyB**	**Function**	**F-GV**
MGMC	61	50	64	66	63	63
ACDC	60	60	77	78	74	74
Bunch-NAHC	48	40	49	47	53	46
Bunch-SAHC	54	39	53	40	53	54
WCA-UE	30	30	32	45	31	31
WCA-UENM	30	30	32	45	31	31
LIMBO	23	23	24	25	24	23
K-means	44	37	39	41	39	38

Table 5. MoJoFM results for *Chromium*.

Method	**Include**	**S-CHA**	**S-Int**	**S-NoDyB**	**Function**	**F-GV**
MGMC	59	56	55	64	67	67
ACDC	64	70	73	71	71	71
Bunch-NAHC	28	31	24	29	29	35
Bunch-SAHC	12	71	43	42	39	29
WCA-UE	23	23	23	27	29	29
WCA-UENM	23	23	23	27	29	29
LIMBO	N/A	23	3	26	27	27
K-means	40	42	43	43	45	45

Tables 8, 9, 10, 11 and 12 list the results for the *normalized TurboMQ* metric. We can see that MGMC is even much better than all the other SAR approaches for this metric. It also consistently outperforms ACDC. In fact, this result is expected because the *normalized TurboMQ* metric evaluates the modularity of the clusters and MGMC aims at maximizing this property although the metrics

Table 6. MoJoFM results for *Hadoop*.

Method	Include	S-CHA	S-Int	S-NoDyB	Function	F-GV
MGMC	27	24	40	42	37	39
ACDC	24	29	41	41	41	41
Bunch-NAHC	23	21	24	24	26	26
Bunch-SAHC	24	26	28	26	29	28
WCA-UE	13	12	15	28	17	17
WCA-UENM	13	12	15	28	17	17
LIMBO	15	13	14	14	13	14
K-means	30	25	29	28	29	29

Table 7. MoJoFM results for *ITK*.

Method	Include	S-CHA	S-Int	S-NoDyB	Function	F-GV
MGMC	50	57	56	54	62	62
ACDC	52	55	52	48	60	60
Bunch-NAHC	37	36	35	35	45	47
Bunch-SAHC	32	46	43	41	54	53
WCA-UE	30	31	44	45	36	36
WCA-UENM	30	31	44	45	36	36
LIMBO	30	31	44	38	36	35
K-means	38	42	39	43	60	61

used for assessing modularity are different. Bunch variants also aim at improving modularity. Hence, it is interesting to see Bunch variants lagging behind for this metric as well. There is one exception to this observation among the results, which is related to the *Archstudio* project (Table 9). Here, Bunch variants outperform all the other SAR approaches in general, although the best overall result is still obtained with MGMC.

We manually analyzed the clustering output provided by MGMC for the *S-CHA* dependency file regarding the *ArchStudio* project in detail. We noticed that there are many clusters in the output that contain a single item only. Then, we checked the occurrence of these items in the input dependency graph. We found out that they are subject to reflexive dependencies. For instance, the following file is specified to be dependent on itself only:

```
edu.uci.isr.archstudio4.comp.archipelago.ObjRefTransfer
```

The output of MGMC is reasonable for such cases. A cluster with no external dependencies may not be merged with other clusters. Also, an item that is dependent on itself only may not be moved to other clusters. These actions would not improve the modularity measure. Indeed, we observed that the TurboMQ value increases from 31 to 70 for MGMC after we remove reflexive dependencies.

5.2 Runtime Performance of Modularity Clustering

Figure 1 depicts a box-plot regarding the execution times of MGMC for the largest set of input files. Hereby, the x-axis lists the four largest dependency graphs in the dataset that are provided as input for clustering. These are all extracted from the *Chromium* project. The total completion time of clustering is indicated by the y-axis in seconds. Recall that we used a laptop computer with Intel Core i7 1.80 GHz CPU and 16 GB RAM to run the experiments. Yet, the execution time do not exceed half a minute even for the largest input file. However, ACDC, which was reported as the most scalable technique, took 70–120 min to run for the same input file on a 3.3 GHz E5-1660 server with 32 GB RAM [21]. Results for the other SAR approaches obtained only after 8 to 24 h of running or a timeout error [21]. Therefore, we conclude that MGMC runs orders-of-magnitude faster than all the other algorithms.

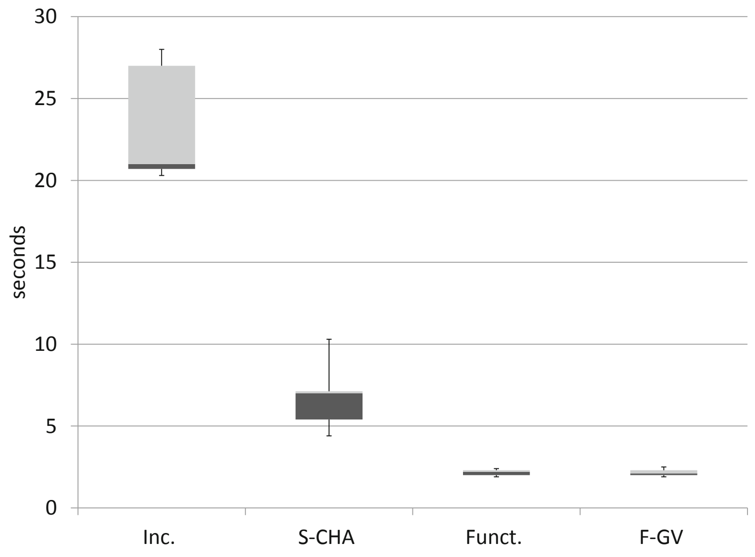

Fig. 1. Runtime performance of MGMC on the largest dependency graphs extracted from the *Chromium* project.

5.3 Threats to Validity

There are several validity threats to our evaluation. First, our evaluation is based on the commonly used *MoJoFM* metric. It was shown that this metric was preferable to other alternatives when the architectures being compared contain the same files [21]. The validity of the ground-truth archtiectures poses another threat for the study. However, actual developers and architects of the projects

Table 8. Normalized TurboMQ results for *Bash*.

Method	**Include**	**Symbol**	**Function**	**F-GV**
MGMC	74	64	63	63
ACDC	9	22	29	29
Bunch-NAHC	25	31	33	28
Bunch-SAHC	30	30	28	28
WCA-UE	0	7	10	10
WCA-UENM	0	7	5	10
LIMBO	6	13	7	7
K-means	0	17	14	16

Table 9. Normalized TurboMQ results for *ArchStudio*.

Method	**Include**	**S-CHA**	**S-Int**	**S-NoDyB**	**Function**	**F-GV**
MGMC	89	31	50	50	54	37
ACDC	66	41	76	84	72	74
Bunch-NAHC	72	42	74	85	74	75
Bunch-SAHC	71	41	76	85	72	74
WCA-UE	1	11	22	65	10	19
WCA-UENM	1	11	22	65	10	19
LIMBO	2	12	31	38	24	27
K-means	13	21	38	51	35	39

were involved in the extraction of this information [21]. To mitigate these threats, we used a second measure, *normalized TurboMQ*, which measures the quality of a clustering independent of any ground-truth architecture. This measure is based on the modularity metric utilized by the Bunch tool [23] and it is subject to flaws, i.e., it is possible to obtain the maximum score by grouping all the modules in a single cluster. We manually checked results for such cases. Our evaluation is based on five subject systems, which limits the generalizability of conclusions. These systems were selected to be of different size, functionality and design/implementation paradigms to mitigate this threat. It is not easy to extend the dataset due to difficulties in obtaining ground-truth architectures [14].

Table 10. Normalized TurboMQ results for *Chromium*.

Method	Include	S-CHA	S-Int	S-NoDyB	Function	F-GV
MGMC	94	90	80	94	93	93
ACDC	15	19	18	20	24	24
Bunch-NAHC	4	24	9	26	16	19
Bunch-SAHC	2	30	11	23	29	11
WCA-UE	0	2	2	2	2	2
WCA-UENM	0	2	2	2	2	3
LIMBO	N/A	2	2	2	2	2
K-means	0	17	13	19	22	22

Table 11. Normalized TurboMQ results for *Hadoop*.

Method	Include	S-CHA	S-Int	S-NoDyB	Function	F-GV
MGMC	89	45	48	52	54	45
ACDC	48	28	59	65	57	58
Bunch-NAHC	40	26	53	61	52	48
Bunch-SAHC	40	31	53	61	54	56
WCA-UE	1	5	8	34	6	8
WCA-UENM	1	5	8	33	6	8
LIMBO	2	7	19	25	17	17
K-means	11	13	29	34	26	27

Table 12. Normalized TurboMQ results for *ITK*.

Method	Include	S-CHA	S-Int	S-NoDyB	Function	F-GV
MGMC	95	92	80	90	94	94
ACDC	33	24	18	32	40	40
Bunch-NAHC	15	23	23	22	34	37
Bunch-SAHC	10	29	23	21	44	37
WCA-UE	3	9	3	2	10	9
WCA-UENM	3	9	3	2	10	19
LIMBO	7	11	5	1	9	9
K-means	13	24	15	13	31	25

6 Conclusion and Future Work

We introduced an empirical evaluation of MGMC used for SAR. We used five open source projects as subject systems for which the ground-truth architectures were known. Various types of dependencies extracted from these systems were previously used as input to evaluate their impact on the accuracy of state-of-the-art SAR techniques. We used the same dataset to evaluate the accuracy and runtime performance of MGMC and compared the results with respect those achieved with existing techniques. Results showed that the accuracy of MGMC is comparable to that of the best known algorithm so far, namely ACDC [35], outperforming it in approximately half of the cases. In addition, it scales better to very large systems for which it runs orders-of-magnitude faster than all the other algorithms.

As future work, additional metrics can be employed for evaluating the accuracy of clustering results. Other types/variants of greedy, heuristic-based approaches can be employed to maximize modularity. Exact methods can also be applied to obtain the optimal possible outcome as a reference point although they do not scale for large projects. The dataset used for experimentation can also be extended; however, ground-truth architectures are usually not available and it is very effort-consuming to recover them [14].

Acknowledgements. We thank Thibaud Lutellier for providing the extracted dependency data regarding the subject systems.

References

1. Agarwal, G., Kempe, D.: Modularity-maximizing graph communities via mathematical programming. Eur. Phys. J. B **66**(3), 409–418 (2008)
2. Altinisik, M., Sozer, H.: Automated procedure clustering for reverse engineering PL/SQL programs. In: Proceedings of the 31st ACM Symposium on Applied Computing, pp. 1440–1445 (2016)
3. Andritsos, P., Tsaparas, P., Miller, R.J., Sevcik, K.C.: LIMBO: scalable clustering of categorical data. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., Ferrari, E. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 123–146. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24741-8_9
4. Brandes, U., et al.: On modularity clustering. IEEE Trans. Knowl. Data Eng. **20**(2), 172–188 (2008)
5. Callo, T., America, P., Avgeriou, P.: A top-down approach to construct execution views of a large software-intensive system. J. Softw.: Evol. Process **25**(3), 233–260 (2013)
6. Çatalyürek, U., Kaya, K., Langguth, J., Uçar, B.: A partitioning-based divisive clustering technique for maximizing the modularity. In: Bader, D.A., Meyerhenke, H., Sanders, P., Wagner, D. (eds.) Graph Partitioning and Graph Clustering. Contemporary Mathematics. AMS, Providence (2012)
7. Clauset, A., Newman, M., Moore, C.: Finding community structure in very large networks. Phys. Rev. E **70**, 066111 (2004)
8. Corazza, A., Martino, S.D., Maggio, V., Scanniello, G.: Investigating the use of lexical information for software system clustering. In: Proceedings of the 15th European Conference on Software Maintenance and Reengineering, pp. 35–44 (2011)
9. Ducasse, S., Pollet, D.: Software architecture reconstruction: a process-oriented taxonomy. IEEE Trans. Softw. Eng. **35**(4), 573–591 (2009)

10. Eick, S., Graves, T., Karr, A., Marron, J., Mockus, A.: Does code decay? Assessing the evidence from change management data. IEEE Trans. Softw. Eng. **27**(1), 1–12 (2001)
11. Eppinger, S., Browning, T.: Design Structure Matrix Methods and Applications. MIT Press, Cambridge (2012)
12. Ersoy, E., Kaya, K., Altınışık, M., Sözer, H.: Using hypergraph clustering for software architecture reconstruction of data-tier software. In: Tekinerdogan, B., Zdun, U., Babar, A. (eds.) ECSA 2016. LNCS, vol. 9839, pp. 326–333. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48992-6_24
13. Garcia, J., Ivkovic, I., Medvidovic, N.: A comparative analysis of software architecture recovery techniques. In: Proceedings of the 28th International Conference on Automated Software Engineering, pp. 486–496 (2013)
14. Garcia, J., Krka, I., Mattmann, C., Medvidovic, N.: Obtaining ground-truth software architectures. In: Proceedings of the International Conference on Software Engineering, pp. 901–910 (2013)
15. Garcia, J., Popescu, D., Mattmann, C., Medvidovic, N., Cai, Y.: Enhancing architectural recovery using concerns. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, pp. 552–555 (2011)
16. Garlan, D., et al.: Documenting Software Architectures: Views and Beyond, 2nd edn. Addison-Wesley, Boston (2010)
17. Guo, G.Y., Atlee, J.M., Kazman, R.: A software architecture reconstruction method. In: Donohoe, P. (ed.) Software Architecture. ITIFIP, vol. 12, pp. 15–33. Springer, Boston (1999). https://doi.org/10.1007/978-0-387-35563-4_2
18. Hendrickson, B., Leland, R.: A multi-level algorithm for partitioning graphs. In: Proceedings of the ACM/IEEE Conference on Supercomputing, p. 28 (1995)
19. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
20. Kobayashi, K., Kamimura, M., Kato, K., Yano, K., Matsuo, A.: Feature-gathering dependency-based software clustering using dedication and modularity. In: Proceedings of the 28th IEEE International Conference on Software Maintenance, pp. 462–471 (2012)
21. Lutellier, T., et al.: Measuring the impact of code dependencies on software architecture recovery techniques. IEEE Trans. Softw. Eng. **44**(2), 159–181 (2018)
22. Maqbool, O., Babri, H.: The weighted combined algorithm: a linkage algorithm for software clustering. In: Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering, pp. 15–24 (2004)
23. Mitchell, B., Mancoridis, S.: On the automatic modularization of software systems using the Bunch tool. IEEE Trans. Softw. Eng. **32**(3), 193–208 (2006)
24. Murphy, G., Notkin, D., Sullivan, K.: Software reflexion models: bridging the gap between design and implementation. IEEE Trans. Softw. Eng. **27**(4), 364–380 (2001)
25. Noack, A., Rotta, R.: Multi-level algorithms for modularity clustering. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 257–268. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02011-7_24
26. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)
27. Qingshan, L., Hua, C., Ping, C., Yun, Z.: Architecture recovery and abstraction from the perspective of processes. In: Proceedings of the 12th Working Conference on Reverse Engineering, pp. 57–66 (2005)

28. Rossi, F., Villa-Vialaneix, N.: Représentation d'un grand réseau á partir d'une classification hiérarchique de ses sommets. Journal de la Société Française de Statistique **152**(3), 34–65 (2011)
29. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using dependency models to manage complex software architecture. In: Proceedings of the 20th Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 167–176 (2005)
30. Sangwan, R., Neill, C.: Characterizing essential and incidental complexity in software architectures. In: Proceedings of the 3rd European Conference on Software Architecture, pp. 265–268 (2009)
31. Schuetz, P., Caflisch, A.: Efficient modularity optimization by multistep greedy algorithm and vertex mover refinement. Phys. Rev. E **77**, 046112 (2008)
32. Storey, M.A., Wong, K., Muller, H.: Rigi: A visualization environment for reverse engineering. In: Proceedings of the 19th International Conference on Software Engineering, pp. 606–607 (1997)
33. Sullivan, K., Cai, Y., Hallen, B., Griswold, W.: The structure and value of modularity in software design. In: Proceedings of the 8th European Software Engineering Conference, pp. 99–108 (2001)
34. Sun, C., Zhou, J., Cao, J., Jin, M., Liu, C., Shen, Y.: ReArchJBs: a tool for automated software architecture recovery of JavaBeans-based applications. In: Proceedings of the 16th Australian Software Engineering Conference, pp. 270–280 (2005)
35. Tzerpos, V., Holt, R.: ACDC: an algorithm for comprehension-driven clustering. In: Proceedings of the 7th Working Conference on Reverse Engineering, pp. 258–267 (2000)
36. Wen, Z., Tzerpos, V.: An effectiveness measure for software clustering algorithms. In: Proceedings of the 12th IEEE International Workshop on Program Comprehension, pp. 194–203 (2004)
37. Wong, K.: RIGI User's Manual. University of Victoria (1996)
38. Wu, J., Hassan, A.E., Holt, R.C.: Comparison of clustering algorithms in the context of software evolution. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 525–535 (2005)
39. Xu, G., Tsoka, S., Papageorgiou, L.: Finding community structures in complex networks using mixed integer optimisation. Eur. Phys. J. B **60**(2), 231–239 (2007)