ILS for the Software Module Clustering Problem

Alexandre Fernandes Pinto
Federal University of the State of Rio de Janeiro
Avenida Pasteur 458, Rio de Janeiro, RJ, 22.290-240, Brazil
alexandre.pinto@uniriotec.br

Adriana Cesário de Faria Alvim
Federal University of the State of Rio de Janeiro
Avenida Pasteur 458, Rio de Janeiro, RJ, 22.290-240, Brazil
adriana@uniriotec.br

Márcio de Oliveira Barros
Federal University of the State of Rio de Janeiro
Avenida Pasteur 458, Rio de Janeiro, RJ, 22.290-240, Brazil
marcio.barros@uniriotec.br

ABSTRACT

In this work we present an Iterated Local Search (ILS) based heuristic for the Software Module Clustering Problem (SMCP). When designing an ILS algorithm there are four free components left for the algorithm designer: the initial solution, a local search procedure, the perturbation strategy and an acceptance criterion. An extensive experiment was conducted to find the best choices for the initial solution and the perturbation components of the proposed ILS algorithm. Another study compared our best ILS with several configurations of Genetic Algorithms. Although the ILS metaheuristic is not extensively used to address software engineering problems, it was proven very effective for our selected problem, outperforming the best configuration for the Genetic Algorithm in 24 out of 40 instances and using a fraction of the latter's computing effort.

KEYWORDS. Iterated Local Search, Software Clustering, benchmarks.

Main Area: Metaheuristics.

1. Introduction

Software systems contain a finite set of software components, for example *modules*, along with a finite set of relationships between these components (import, export, inherit, procedure invocation, and variable access). As a software system grows in size, the distribution of its modules in larger, container-like structures becomes a relevant design decision. A proper module distribution aids in the identification of the modules responsible for a given functionality, provides easier navigation among software parts and enhances source code comprehension. Therefore, it supports the development and maintenance of a software system. Apart from that, experiments have shown a strong correlation between bad module distributions and the presence of faults in software systems.

The Software Module Clustering Problem (SMCP) can be defined as the problem of partitioning modules into *clusters* so as to optimize some *criterion*. Formally, given a set $N = \{1, \ldots, n\}$ of modules and a cost $c(i, j)$ which indicates the relation between each pair $i, j \in N$, the SMCP consists of finding a partition $P = \{C_1, C_2, \ldots, C_m\}$ of N into m clusters, $1 \leq m \leq n$, with $C_i \neq \emptyset, i = 1, \ldots, m$; $C_i \cap C_j = \emptyset, i, j = 1, \ldots, m, i \neq j$ and $\bigcup_{i=1}^{m} C_i = N$ so as to maximize/minimize some criterion. The relation $c(i, j)$ indicates the dependency between modules i and j, meaning that if $c(i, j) \geq 1$ then module i is dependent of module j. Two criterions widely used in literature are: (i) minimize connections between modules of two distinct clusters (inter-connectivity) and (ii) maximize connections between modules of the same cluster (intra-connectivity). A proper module distribution aids in the identification of the modules responsible for a given functionality [Briand (1999)], provides easier navigation among software parts and enhances source code comprehension [Larman (2002)]. The SMCP is essentially a Graph Partitioning Problem, which is known to be NP-hard [Garey and Johnson (1979)], for which heuristic algorithms provide reasonable solutions in acceptable computing times.

In this work we describe an Iterated Local Search (ILS) [Loureno et al. (2002)] based heuristic for the Software Module Clustering Problem. The contributions of our work are twofold: (i) to evaluate the efficiency and effectiveness of our proposed heuristic we report on an extensive empirical study to address the best configuration for the ILS heuristic and compare it to different configurations of Genetic Algorithms and (ii) we present 40 instances to serve as benchmarks for experimental evaluations related to the Software Module Clustering Problem.

The paper is organized as follows: Section 2 introduces the SMC problem. In Section 3, we briefly review the General ILS frame and present our adaptation of ILS for the SMCP. The effectiveness of our approach is tested through extensive computational experiments in Section 4. Concluding remarks are made in the last section.

2. The Software Module Clustering Problem

To evaluate the quality of its module distribution, a software system is usually represented as a Module Dependency Graph, or MDG [Mancoridis et al. (1999)]. The MDG is a directed graph in which modules are shown as nodes, dependencies are shown as edges, and clusters are partitions. Weights may be assigned to edges, denoting the strength of the dependency between the modules represented by the edge's source and target nodes. These weights may represent, for instance, the number of calls on the module represented by the source node referring to methods declared in the target node. In unweighted MDG, all edge weights are set to 1. The coupling of a cluster can be calculated by summing the weights of edges leaving or entering the partition (inter-edges), while its cohesion is calculated by summing the weights of edges whose source and target modules pertain to the partition (intra-edges) [Praditwong et al. (2011)].

Mancoridis et al. (1999) were the first to present a search-based assessment of the SMCP. They propose a Hill Climbing search to find the best module distribution for a system. The search is guided by a fitness function called Modularization Quality (MQ). Let m be the number of clusters, i the sum of intra-edge weights and j the sum of inter-edge weights for cluster $C_k, k = 1, \ldots, m$. MQ is calculated as the sum of the modularization factors (MF) of all clusters. MF is defined as:

$$MF(C_k) = \begin{cases} 0 & \text{if } i = 0 \\ \frac{i}{i+j/2} & \text{if } i > 0. \end{cases} \quad (1)$$

MQ looks for a balance between coupling and cohesion, rewarding clusters with many intra-edges and penalizing them for dependencies with other clusters. According to Köhler et al. (2013), MF can be seen as a generalization of the fitness function known as relative density that measure the quality of a cluster within a graph. Síma and Schaeffer (2005) prove that the decision problem associated with the optimization task of finding the clusters that are optimal with respect to the relative density measure is NP-complete. The Hill Climbing search proposed in [Mancoridis et al. (1999)] aims to find partitions of a MDG with the highest MQ possible. Doval et al. (1999) present a genetic algorithm to address the `SMCP`, using MQ as a mono-objective fitness function. The genetic algorithm, however, has been found less effective and less efficient than the Hill Climbing search. Mahdavi et al. (2003) address the `SMCP` by running 23 independent Hill Climbing searches upon a MDG, building a partial solution consisting of modules sharing the same cluster in more than a predetermined number of solutions produced by the initial searches, and then running a second round of Hill Climbing searches to distribute the remaining modules. The authors saw improvements in solutions found by the second run of local searches when compared to those produced in the first round, particularly for large instances.

Semaan et al. (2011) used a ILS based heuristic to tackle the problem of clustering object-oriented information systems, represented as weighted MDG graph using as objective function the Basic MQ [Doval et al., (1999)]. The `SMCP` has also been addressed using multi-objective optimization [Abdeen et al. (2013), Praditwong et al. (2011), Barros (2012)], interactive search [Abdeen et al. (2013), Bavota et al. (2012)], and different metrics. Köhler et al. (2013) presented a mathematical optimization model and a preprocessing technique for the `SMCP`. A broader survey on search-based applications for software design is provided in [Räihä (2007)].

3. An ILS for the Software Module Clustering Problem

The basic `ILS` frame is summarized below. Assume we have been given a procedure called `GenerateInitialSolution` (Line 1) which constructs an initial solution s_0 for the problem we are addressing. Also assume we have a local search procedure called `LocalSearch` (Line 2) that when applied to a give input, say s_0, always returns the same output s^* which costs less or equal to s_0 (considering a minimization problem), that is, a *local optima*. The key idea underlying the `ILS` method, for escaping from local optima, is to repeatedly perturb the current local optima solution s^* so as to find a "nearby" solution s' (Line 4), then apply a local search procedure (Line 5) to s' finding a new local optima $s^{*'}$ which may be better than s^*. In Line 6 an acceptance test is done to decide whether the new solution $s^{*'}$ will replace the current one when the search returns to solution s^*. This process is repeated (Lines 3–7) until a stopping criterion is met.

procedure `Iterated Local Search`;
1 $s_0 \leftarrow$ GenerateInitialSolution;
2 $s^* \leftarrow$ LocalSearch(s_0);
3 **repeat**
4 $s' \leftarrow$ Perturbation($s^*, history$);
5 $s^{*'} \leftarrow$ LocalSearch(s');
6 $s^* \leftarrow$ AcceptanceCriterion($s^*, s^{*\prime}, history$);
7 **until** termination condition met;
end

In `ILS`, the two common strategies used to escape from local optima – *diversification* and *intensification* – are implemented in the `Perturbation` and `AcceptanceCriterion` procedures, respectively. Let the *strength* of a perturbation be the number of different elements from one

solution to another. If a perturbation is too strong the search behaves more like a random restart. On the other hand, if the perturbation is too weak there is much more chance of falling back into the just visited solution. The `AcceptanceCriterion` procedure decides from which solution the search will continue. A strong intensification strategy is achieved when only better solutions are accepted. Instead, if we always accept the solution produced by the perturbation step, we favor diversification. Another feature from `ILS` is the memory mechanism called *history*. It keeps information about the search and can be used for deciding whether the search accepts a worst solution or not in the `AcceptanceCriterion` procedure. For many problems, a good compromise between *diversification* and *intensification* is not evident to find and is subject of investigation.

A metaheuristic just guides the designer of a specific algorithm for a specific problem. In general, all metaheuristics have free components for which choices are left for the algorithm designer. Next, we show our choices for the four `ILS` free components on regard of the `SMCP`. We called the resulting algorithm `ILS_SMC`. As stopping criterion, we used the number of iterations equal to 200 times n^2, where n is the number of modules, formerly used in Barros (2012).

- *Initial solution:* The Agglomerative Hierarchical Clustering Method [Hansen and Jaumard (1997)] is a classical clustering algorithm in which an initial partition of n single-entity (module) clusters is built and, at each step, two clusters are merged, according to a local criterion, until all entities (modules) belongs to one single cluster. To build an initial solution for the `SMCP` we used a variant of this method. First, n clusters are built, each with a single module of the software. Then, at each step, we evaluate the resulting cost (MQ) of merging each pair of clusters and merge the two cluster that yield the best MQ. The procedure stops in the first iteration that does not find an improvement for MQ.

- *Local Search:* We adopted a Hill Climbing algorithm guided by the MQ measure. The search departs from a solution built using the procedure described above. At each search step, the search evaluates all neighbors of the current solution and selects the best neighbor, that is, the neighbor leading to the larger increase in MQ (this strategy is called best-improvement). The neighborhood relation is defined as the solutions obtained by moving one module from one cluster to another cluster.

- *Perturbation:* We used a move operation as our perturbation strategy. Move randomly selects a module and moves it to a different, randomly-selected cluster.

- *Acceptance criterion:* We used the "strong" intensification strategy which accepts only better solutions.

4. Computational Results

In this section we report on two computational experiments performed with `ILS_SMC`. The first experiment aimed to find the best configuration for the free components that have been designed for `ILS_SMC`. The best configuration for these components has already been presented in Section 3, but here we report on how these settings were found. The second experiment compared results produced by `ILS_SMC` with several configurations of genetic algorithms designed for the `SMCP`. All algorithms were run on an Intel Core i7-2600 3.40 GHz with 4 GB of RAM memory. They were coded in `Java 7` and compiled with `JDK 1.7.0-b147`. Running times are reported in seconds and do not include the cost of reading the input data.

4.1. Test problems

Table 1 describes the 40 test problems instances considered in this work. These instances are available at `https://github.com/smcp/smcp_instances` and can serve as an extensive benchmark for further studies on the `SMCP`. Java code that reads these instances to memory is also available. A subset of these instances were firstly presented in the work of Barros (2012). The

first column of Table 1 indicates the name of the instance, which acts as an identifier to be used in further data tables. Next, we show a short description for the instance, including its source software system along with its version if applicable. The next two columns show, respectively, the number of modules (# Mod.) and the number of module dependencies for each instance (# Dep.).

	Instance name	Description	# Mod.	# Dep.
	JSTL	JSP Standard Tag Library	18	20
*	JNANOXML	XML parser for Java	25	64
	JODAMONEY	Money management library	26	102
	JXLSREADER	Library for reading Excel files	27	73
	SEEMP	Small information system	31	61
*	APACHE	file compression utility	36	86
	UDTJAVA	Native impl. for the UDT protocol v0.5	56	227
	JAVAOCR	Written text recognition library	59	155
	SERVLET	Java Servlets API	63	131
	PFCDA_BASE	Source code analisys (model) v1.1.1	67	197
	FORMS	GUI form handling library v1.3.0	68	270
*	JSCATTERPLOT	Scatter plot library (JTreeview) v1.1.6	74	232
	JFLUID	Java Profiler v1.7.0	82	315
	JXLSCORE	Library to represent Excel files	83	330
	JPASSWORD	Password management program	96	361
*	JUNIT	Unit test support library	100	276
	XMLDOM	Java XML DOM Classes	119	209
*	TINYTIM	Tiny TIM: Topic Maps Engine	134	564
	JKARYOSCOPE	Karyoscope charts (JTreeview) v1.1.6	136	460
*	GAE_PLUGIN	Eclipse Google Plugin	140	375
	JAVACC	Java CC: Yacc for Java	154	722
	JAVAGEOM	Java geometry library v0.11.0	172	1445
*	JDENDOGRAM	Dendogram plot (JTreeview) v.1.1.6	177	583
	XMLAPI	Java XML API	184	413
	JMETAL	JMetal: heuristic search algorithms	190	1137
	DOM4J	DOM4J: alternative XML API for Java	195	930
*	PDF_RENDERER	Java PDF renderer v0.2.1	199	629
	JUNG_GRAPH_MODEL	Jung Graph - model classes v2.0.1	207	603
	JCONSOLE	Java Console (part of JDK) v1.7.0	220	859
*	JUNG_VISUALIZATION	Jung Graph - visualization classes v2.0	221	919
*	PFCDA_SWING	Source code analisys - GUI v1.1.1	252	885
	JPASSWORD_FULL	Password management program full v0.5	269	1348
*	JML	MSN Messenger library for Java v1.0	270	1745
*	NOTEPAD_FULL	Tablet editor v0.2.1	299	1349
	POORMANS CMS	Poor Man's CMS	304	1118
	LOG4J	Logging library for Java 1.2	308	1078
	JTREEVIEW	Stanford Treeview for Java	329	1057
	JACE	Java Apple computer emulator	340	1524
	JAVAWS	Java Web Start from JRE v7	378	1403
	RES_COBOL	COBOL translator to Java	483	7163

Table 1: Test instances used in our experimental studies. All instances were used in our second study, while only instances marked with an asterisk were used in the first one.

4.2. Experimental Design

Heuristic search algorithms contains random components and, thus, may return different solutions for each independent run. To account for this stochastic behavior, each algorithm involved in the studies described in this section was run 30 times for each instance. For each trial, we obtained the MQ of its best solution and the execution time of the algorithm.

MQ values were used to compare different algorithms or different configurations of the same algorithm in terms of effectiveness, that is, its ability to produce good solutions. In this sense, algorithms were compared in a per instance basis and a solution A in considered better than a solution B if A has higher MQ than B. Execution times were used for efficiency comparisons. In this context, an algorithm is considered more efficient if it executes in shorter time.

Statistical tests were applied to determine whether results obtained by distinct algorithms or different configurations of the same algorithm can be considered significantly different from each

other. The Wilcoxon-Mann-Whitney non-parametric statistical inference test was used to compare the means of two independent samples. This test does not require the underlying dataset to be normally distributed or two samples to have equal variance. The test was applied at 95% significance level, but p-values are presented whenever possible.

Effect-size measures denote how often an algorithm is better than another in a set of pairwise comparisons. We have used the A_{12} non-parametric effect-size measure [Vargha and Delaney (2000)] in our studies. This measure yields a value between 0 and 100%. A value of 80% means that algorithm 1 is expected to produce better solutions than algorithm 2 in 8 out of 10 independent runs. Values closer to 50% denote that the algorithms tend to produce similar results.

The R Statistical System v.2.15.2 was used to perform inference tests, as well as to calculate the means, standard deviations, and effect-sizes reported in the following sections.

4.3. Comparing different ILS_CMS configurations

To select the best configuration for ILS_CMS, we designed and executed an experimental study on which 228 different configurations of the algorithm were examined using the 12 instances marked with an asterisk in Table 1.

The configurations selected for the experimental study differed according to (a) the strategy for creating the initial solution; (b) the perturbation strategy; and (c) the number of solutions produced by the selected perturbation strategy which were examined before restarting the local search. On regard of the initial solution, two alternatives were tested: a solution created by the Agglomerative Hierarchical Clustering Method (see Section 3) or a randomly-created solution. On regard of the perturbation, 57 distinct strategies were examined. Finally, concerning the local search, we have evaluated two settings: accepting the first solution produced by perturbation as the new starting point for the local search or testing up to 5 solutions produced by perturbation and selecting a solution only if it improves the best MQ known so far (local search was started from a random solution in case none of the 5 solutions improved the objective function).

To build the 57 perturbation strategies, five operations were examined: (i) moving 10% of the modules of a given solution from their clusters to different, randomly-selected clusters; (ii) exchanging the position of 10% pairs of modules, each module in a pair coming from a different cluster; (iii) merging 10% of the clusters; (iv) selecting 10% of the clusters and dividing each of them into two new clusters; and (v) exploding 10% of the clusters into many cluster, each conveying a single module. Each of these operations was tested in isolation (5 configurations), was combined in groups of two (10 configurations), three (10 configurations), four (5 configurations), or five operations (1 configuration). For each configuration with at least two operations, they were either executed in a predefined order (move, exchange, merge, division, and split) or one operation was randomly-selected whenever a perturbation was required.

Each ILS_CMS configuration was executed 30 times for each instance. Results were compared in a per instance basis. For each instance, we calculated the mean MQ produced by each configuration, selected the larger mean, and marked all configurations whose mean was equal to the larger one as winners for that instance (means were considered equal if their difference was less than 0.01). Finally, we selected the configurations that won more often as the best settings for ILS_CMS.

The best configurations have used the greedy algorithm to create their initial solutions and their local search started from the first solution produced by the selected perturbation strategy. Configuration ILS1 won on 7 out of 12 instances using only the move operation in perturbations. Configurations ILS2 (sequence of move, exchange, and merge for perturbation), ILS3 (exchange for perturbation), and ILS4 (random selection of move or exchange for perturbation) scored 6 victories each. These configurations were selected for a final comparison using statistic inference tests and effect-sizes. The means and standard deviations for MQ found by each configuration are shown in Table 2.

Instances	ILS 1	ILS 2	ILS 3	ILS 4
JNANOXML	$\mathbf{3.82 \pm 0.00}$	$\mathbf{3.82 \pm 0.00}$	$\mathbf{3.82 \pm 0.00}$	$\mathbf{3.82 \pm 0.00}$
APACHE	$\mathbf{5.77 \pm 0.00}$	$\mathbf{5.77 \pm 0.00}$	$\mathbf{5.77 \pm 0.00}$	$\mathbf{5.77 \pm 0.00}$
JSCATTERPLOT	$\mathbf{10.74 \pm 0.02}$	$\mathbf{10.74 \pm 0.02}$	$\mathbf{10.74 \pm 0.00}$	$\mathbf{10.74 \pm 0.01}$
JUNIT	$\mathbf{11.09 \pm 0.00}$	$\mathbf{11.09 \pm 0.00}$	$\mathbf{11.09 \pm 0.00}$	$\mathbf{11.09 \pm 0.00}$
TINYTIM	$\mathbf{12.51 \pm 0.02}$	12.47 ± 0.02	$\mathbf{12.51 \pm 0.02}$	12.50 ± 0.02
GAE_PLUGIN	17.28 ± 0.02	$\mathbf{17.33 \pm 0.02}$	17.29 ± 0.02	17.29 ± 0.02
JDENDOGRAM	$\mathbf{26.07 \pm 0.01}$	$\mathbf{26.07 \pm 0.01}$	$\mathbf{26.07 \pm 0.01}$	$\mathbf{26.07 \pm 0.01}$
PDF_RENDERER	21.85 ± 0.14	21.72 ± 0.14	21.78 ± 0.20	21.82 ± 0.11
JUNG_VISUALIZ	20.88 ± 0.21	20.79 ± 0.20	20.76 ± 0.16	20.81 ± 0.21
PFCDA_SWING	$\mathbf{28.99 \pm 0.03}$	28.94 ± 0.02	28.97 ± 0.03	28.98 ± 0.03
JML	17.40 ± 0.05	17.36 ± 0.03	17.39 ± 0.04	17.40 ± 0.03
NOTEPAD_FULL	29.48 ± 0.05	29.41 ± 0.06	29.48 ± 0.05	$\mathbf{29.49 \pm 0.06}$

Table 2: Means and standard deviations of MQ for the best ILS_CMS configurations.

	$ILS1 > ILS2$		$ILS1 > ILS3$		$ILS4 > ILS2$		$ILS4 > ILS3$		$ILS1 > ILS4$	
Instances	PV	ES	PV	ES	PV	ES	PV	ES	PV	ES
JNANOXML	-	50%	-	50%	-	50%	-	50%	-	50%
APACHE	1.000	50%	0.161	47%	0.161	53%	-	50%	0.161	47%
JSCATTERPLOT	0.452	53%	0.161	47%	0.371	54%	0.161	47%	0.959	50%
JUNIT	-	50%	-	50%	-	50%	-	50%	-	50%
TINYTIM	< 0.001	93%	0.684	53%	< 0.001	90%	0.399	44%	0.217	59%
GAE_PLUGIN	< 0.001	8%	0.435	44%	< 0.001	8%	0.510	45%	0.833	48%
JDENDOGRAM	0.797	52%	0.881	49%	0.884	51%	0.477	45%	0.858	51%
PDF_RENDERER	< 0.001	77%	0.208	60%	0.001	76%	0.662	53%	0.208	60%
JUNG_VISUALIZ	0.088	63%	0.015	68%	0.719	53%	0.343	57%	0.224	59%
PFCDA_SWING	< 0.001	91%	0.004	71%	< 0.001	83%	0.234	59%	0.115	62%
JML	< 0.001	79%	0.483	55%	< 0.001	83%	0.181	60%	0.483	45%
NOTEPAD_FULL	< 0.001	82%	0.965	50%	< 0.001	84%	0.356	57%	0.356	43%

Table 3: P-values and effect-sizes for pair-wise comparisons of the best ILS_CMS configurations.

The four configurations with greater MQ means were compared using the selected non-parametric inference test and effect-size measures. Table 3 shows the comparative results of statistical tests between these configurations. P-values shown as hyphens correspond to samples that produced identical results and thus cannot be statistically different.

The first column of Table 3 shows instance identifiers, these instances being listed in the increasing order of their size, measured in number of modules. The second and third columns respectively report the p-value of the inference test and the effect-size for comparing ILS1 to ILS2. A p-value under 0.05 indicates significantly different MQ means with 95% confidence. The comparison between ILS1 and ILS2 shows that sample means are statistically different for 7 out of 12 instances, with effect-size favoring ILS1 (particularly on the three largest instances).

The next columns work in pairs and follow the same pattern used in the second and third columns. The comparison between ILS1 and ILS3 shows that sample means are statistically different for only 2 out of 12 instances, but effect-sizes tend to favor ILS1 for instances with more than 199 modules (from PDF_RENDERER on). The comparison between ILS4 and ILS2 shows that sample means are statistically different in half of the instances, but ILS4 outperforms ILS2 while searching solutions for the largest ones. The comparison between ILS4 and ILS3 did not find any significant differences among means, but effect-size slightly favors ILS4 for large instances. Finally, no significant difference among means was found in the comparison between ILS1 and ILS4: only small effect-sizes were found, in average favoring ILS1 (52%).

These results indicate that ILS1 and ILS4 are the most promising settings for ILS_CMS. Although solutions found by these settings were not significantly different, ILS1 has shown a slightly greater effect-size and is selected for the next study, which compares ILS1 with genetic algorithms designed to address the SMCP. Furthermore, it is important to notice that we do not report on execution times for this study because execution times for all selected configurations were

very close for practical purposes.

4.4. Comparing ILS_CMS to genetic algorithms

After determining the best configuration for ILS_CMS, we designed and executed an experimental study to compare its effectiveness and efficiency with two GA-based approaches. To allow for a fair comparison, the same procedure used to select the best ILS configuration for the SMCP was used to select the best configuration for both GA-based approaches. Then, the best GA of each kind was compared to ILS_CMS in order to find the most appropriate algorithm for the mono-objective formulation of the SMCP.

4.4.1. Genetic Algorithms for SMCP

Many GA-based algorithms were built to address the SMCP, using both mono-objective [Doval et al. (1999), Bavota et al. (2012)] and multi-objective [Abdeen et al. (2013), Praditwong et al. (2011), Barros (2012)] formulations for the problem. Most of these works use the group-numbers encoding (GNE) representation. In this representation, a chromosome has one gene per module and each gene encodes the number of the cluster where the module belongs in the solution. One problem of this encoding schema is its high degree of redundancy, since different numbers may refer to the same cluster in distinct solutions.

To deal with this issue, Falkenauer (1996) presented the Grouping Genetic Algorithm (GGA) in which clusters, instead of modules, become genes in the chromosome. Since the number of clusters may vary from one solution to another, the size of the chromosome may change from solution to solution and special genetic operators are used for crossover and mutation. Falkenauer (1996) suggests dividing the chromosome into two parts: one to identify the modules pertaining to each clusters and the second to identify the cluster themselves. Praditwong (2011) applied the GGA approach to the SMCP and found that it outperformed the GNE approach for large-sized software projects represented as unweighted MDG.

In the present work, we implemented two GA-based algorithms for the SMCP: one using GNE (GA_GNE) and the other using GGA (GA_GGA). Both algorithms use MQ as their objective function. We have tested different configurations of each algorithm to find their best genetic operators using the same 12 instance that were used to find the best configuration for ILS_CMS. Next, we outline our specific choices for the best configuration for the GA_GNE algorithm:

- *Crossover operator*: two-point crossover, with random selection of the two cutting positions. The probability of applying the crossover operator is set to 80% for systems with less than 100 modules and 100% for larger systems. The two parents subjected to crossover are replaced by their two offspring in the new generation;
- *Parent selection*: binary tournament with 30% elitism (that is, the best 30% instances are selected before binary tournament is applied to select parents for crossover in order to complement the population);
- *Mutation operator*: uniform mutation with $0.04 * \log_2(n)$ probability.

A similar procedure was used to select the best configuration for GA_GGA:

- *Crossover operator*: same crossover proposed by Praditwong (2011). First, two parents are selected using binary tournament. For each parent, select one position in the list of clusters. Then, select a second position, necessarily after the first one, so that at most 10% of all clusters are contained in the region between the two positions. A similar procedure is applied to the second chromosome and the clusters in each selected region are exchanged between the chromosomes. In case any of the transferred clusters carries a module that already pertains to another cluster in the target chromosome, the module is removed from its original cluster and maintained in that received as part of the crossover. Finally, clusters with a single module are merged to a randomly-selected cluster;

- *Mutation operator*: the mutation is applied on the two offspring. It randomly chooses one of the following three options: a) join two clusters randomly selected; b) split a randomly selected cluster in two clusters; and c) move a module from one cluster to another, both randomly selected. The probability of applying mutation is $0.04 * \log_2(n)$.

- *Population diversity*: in order to introduce diversity in the population, the algorithm counts the number of individuals in a new generation having the same fitness. If this number exceeds 0.1% of the individuals, a uniform mutation is performed on the newly inserted individual (5% randomly-selected modules are moved to other clusters).

Both GA_GNE and GA_GGA used a population of $10n$ individuals, where n is the number of modules of the instance under analysis. The initial population was randomly-generated. For the stopping criterion, we used the same maximum number of evaluations as for ILS_CMS.

4.4.2. Analysis of Solution Quality

In this section the algorithms are compared on regard of the quality of the solution they produced, that is, on the mean values of their best solution's MQ over the course of their 30 independent runs. Means and standard deviations for MQ for each of the 40 instances are shown in Table 4. The table is divided into three sections, separating small, medium, and large instances. The best means for each instance are shown in bold.

The first column in Table 4 contains instance identifiers. The second column shows the means and standard deviations of the best MQ found by ILS_CMS. The next column shows means and standard deviations found by GA_GNE and the following column shows similar results for GA_GNE. As can be seen, GA_GNE is outperformed by GA_GGA or ILS_CMS in all but one instance, on which all algorithms find solutions with similar MQ. Thus, from this point on GA_GNE is left out of further comparisons. The fifth column in Table 4 shows the p-value for comparing ILS_CMS and GA_GGA with the statistical inference test and the last column shows the effect-size for the pair-wise comparison of these algorithms.

Considering small instances (with up to 100 modules), ILS_CMS produced the best means for 13 out of 16 instances, while GA_GGA produced the best solutions for 9 instances. ILS_CMS significantly outperformed GA_GGA in 7 out of 16 instances. Overall, ILS_CMS proved slightly superior to GA_GGA for small instances, showing an average effect-size of 52%.

For mid-sized instances (from 101 to 200 modules), ILS_CMS produced the best mean MQ in 8 out of 11 instances, while GA_GGA produced the best average in 3 instances. The differences between the algorithms were considered significant in 8 out of 11 instance and again ILS_CMS has shown a slightly greater effect-size (61%) when compared to GA_GGA.

In large instances (with more than 200 modules), ILS_CMS obtained the best average in 9 out of 13 instances, while GA_GGA outperformed the former in only 4 instances. 8 comparisons were statistically significant, 7 of them favoring ILS_CMS. Finally, as it happened with small and mid-sized instances ILS_CMS showed a slightly higher effect-size than GA_GGA for large instances (67%).

ILS_CMS was superior in terms of solution quality to the selected genetic algorithms while addressing the SMCP. ILS_CMS obtained the best values for MQ in most instances, regardless of their sizes and number of dependencies, demonstrating its scalability as instance size grows. The second best performing heuristic was GA_GGA, with average effect-size for all instances of 59% favoring ILS_CMS. Comparing the two genetic algorithm approaches, GA_GGA obtained a mean effect-size of 88% in comparisons with GA_GNE. Thus, we can conclude that there is evidence that ILS_CMS can find better solutions for the mono-objective formulation of the SMCP than genetic algorithms based on GNE and GGA representation.

Instance	ILS_CMS	GA_GNE	GA_GGA	PV	ES
JSTL	$\mathbf{3.31 \pm 0.04}$	3.30 ± 0.04	3.29 ± 0.06	0.039	64%
JNANOXML	$\mathbf{3.82 \pm 0.00}$	3.77 ± 0.03	3.81 ± 0.01	0.006	62%
JODAMONEY	$\mathbf{2.75 \pm 0.00}$	2.73 ± 0.03	$\mathbf{2.75 \pm 0.00}$	-	50%
JXLSREADER	$\mathbf{3.60 \pm 0.00}$	3.58 ± 0.03	$\mathbf{3.60 \pm 0.00}$	-	50%
SEEMP	$\mathbf{4.65 \pm 0.00}$	4.64 ± 0.02	$\mathbf{4.65 \pm 0.00}$	-	50%
APACHE	$\mathbf{5.77 \pm 0.00}$	5.74 ± 0.03	$\mathbf{5.77 \pm 0.00}$	0.161	47%
UDTJAVA	$\mathbf{5.28 \pm 0.01}$	5.24 ± 0.03	$\mathbf{5.28 \pm 0.01}$	0.246	44%
JAVAOCR	9.00 ± 0.03	8.95 ± 0.04	$\mathbf{9.02 \pm 0.01}$	< 0.001	28%
SERVLET	9.47 ± 0.06	9.45 ± 0.11	$\mathbf{9.50 \pm 0.07}$	0.039	35%
PFCDA_BASE	$\mathbf{7.33 \pm 0.01}$	7.30 ± 0.03	7.32 ± 0.01	0.028	66%
FORMS	$\mathbf{8.33 \pm 0.00}$	8.30 ± 0.03	8.32 ± 0.01	0.011	60%
JSCATTERPLOT	$\mathbf{10.74 \pm 0.02}$	10.68 ± 0.05	$\mathbf{10.74 \pm 0.00}$	0.161	47%
JFLUID	$\mathbf{6.58 \pm 0.00}$	6.51 ± 0.05	6.54 ± 0.05	< 0.001	78%
JXLSCORE	$\mathbf{9.30 \pm 0.09}$	$\mathbf{9.30 \pm 0.04}$	9.29 ± 0.09	0.367	57%
JPASSWORD	10.29 ± 0.04	10.28 ± 0.06	$\mathbf{10.34 \pm 0.04}$	< 0.001	21%
JUNIT	$\mathbf{11.09 \pm 0.00}$	11.06 ± 0.05	11.08 ± 0.01	< 0.001	78%
XMLDOM	10.86 ± 0.03	10.84 ± 0.07	$\mathbf{10.88 \pm 0.05}$	0.083	38%
TINYTIM	$\mathbf{12.51 \pm 0.02}$	12.31 ± 0.07	12.45 ± 0.05	< 0.001	87%
JKARYOSCOPE	$\mathbf{18.98 \pm 0.02}$	18.90 ± 0.05	18.96 ± 0.04	0.014	64%
GAE_PLUGIN	17.28 ± 0.02	17.29 ± 0.04	$\mathbf{17.29 \pm 0.04}$	0.026	33%
JAVACC	10.60 ± 0.05	10.43 ± 0.07	$\mathbf{10.62 \pm 0.06}$	0.145	39%
JAVAGEOM	$\mathbf{14.07 \pm 0.02}$	13.67 ± 0.08	14.01 ± 0.04	< 0.001	93%
JDENDOGRAM	$\mathbf{26.07 \pm 0.01}$	25.58 ± 0.12	26.03 ± 0.03	< 0.001	87%
XMLAPI	$\mathbf{18.99 \pm 0.04}$	18.49 ± 0.11	18.97 ± 0.08	0.636	54%
JMETAL	$\mathbf{12.41 \pm 0.03}$	11.87 ± 0.11	12.39 ± 0.09	0.790	52%
DOM4J	$\mathbf{18.83 \pm 0.10}$	17.97 ± 0.19	18.77 ± 0.13	0.053	65%
PDF_RENDERER	$\mathbf{21.85 \pm 0.14}$	21.19 ± 0.19	21.82 ± 0.12	0.277	58%
JUNG_GRAPH	31.52 ± 0.14	30.16 ± 0.34	$\mathbf{31.55 \pm 0.11}$	0.636	46%
JCONSOLE	$\mathbf{26.51 \pm 0.01}$	24.99 ± 0.19	26.43 ± 0.06	< 0.001	94%
JUNG_VISUALIZ	20.88 ± 0.21	20.16 ± 0.16	$\mathbf{21.05 \pm 0.07}$	0.001	24%
PFCDA_SWING	$\mathbf{28.99 \pm 0.03}$	26.25 ± 0.30	28.85 ± 0.08	< 0.001	94%
JPASSWORD_FULL	$\mathbf{27.89 \pm 0.08}$	24.74 ± 0.23	27.87 ± 0.08	0.176	60%
JML	$\mathbf{17.40 \pm 0.05}$	15.28 ± 0.16	17.35 ± 0.09	0.005	71%
NOTEPAD_FULL	29.48 ± 0.05	25.47 ± 0.28	$\mathbf{29.49 \pm 0.10}$	0.941	51%
POORMANS	$\mathbf{34.13 \pm 0.03}$	28.79 ± 0.28	34.06 ± 0.12	0.006	71%
LOG4J	31.43 ± 0.21	26.71 ± 0.30	$\mathbf{31.49 \pm 0.18}$	0.371	43%
JTREEVIEW	$\mathbf{47.59 \pm 0.15}$	39.28 ± 0.47	47.55 ± 0.12	0.198	60%
JACE	$\mathbf{26.63 \pm 0.02}$	23.16 ± 0.17	26.59 ± 0.08	0.007	70%
JAVAWS	$\mathbf{38.27 \pm 0.02}$	29.73 ± 0.36	38.19 ± 0.09	< 0.001	89%
RES_COBOL	$\mathbf{15.97 \pm 0.01}$	12.86 ± 0.10	15.89 ± 0.06	< 0.001	97%

Table 4: Means and standard deviations for MQ found by the selected algorithms.

4.4.3. Analysis of Execution Time

On regard of efficiency, Figure 1 shows that the computational time required by GA_GGA increases much faster than the time required by ILS_CMS as instance size grows. The X axis in this chart shows instance sizes, while the Y axis denotes the time required to run the algorithms, measured in seconds. While GA_GGA requires ten times the computational effort required by ILS_CMS for the smallest instances, this ratio grows to more than a thousand times for the largest ones. Thus, we can conclude that ILS_CMS outperforms GA_GGA in terms of computational efficiency for all instances.

4.5. Threats to Validity

A fundamental question concerning experimental studies is how valid and general are their results. According to Wohlin et al. (2000) threats to validity can be divided into four categories: Construct, Internal, External and Conclusion.

Construct threats deals with the relationship between theory and observation. As a measure against the construct validity threats, the theoretical definition of the problem was presented in Section 2, while the experimental design was detailed in Section 4. Valid and well-known metrics were used to address efficiency (execution time) and effectiveness (MQ) of the selected algorithms while addressing the SMCP.

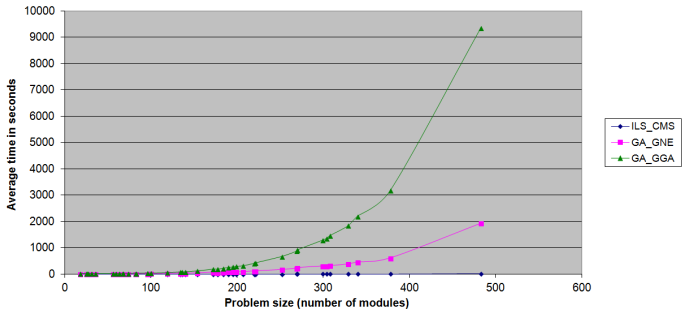

Figure 1: Average CPU time as a function of problem size (number of modules) for algorithms ILS_CMS, GA_GGA and GA_GNE.

Internal validity threats ascertain that if a relationship is observed between the treatment and the output, it is a causal relationship and not the result of a uncontrolled or unmeasurable factor introduced in the experimental design. These threats have been treated by reusing algorithm configurations and parameters used in previous studies addressing the SMCP. Also, a systematic procedure was devised to determine the best configuration for each algorithm. Finally, real open-source instances were used in the experiments and made available for replication purposes.

External validity relates to our ability to generalize the results of the study for other instances. Although instances have been collected mainly from software written in Java, the selected systems are real applications developed for distinct domains and presenting different sizes. We have also used a large number of instances and thus are confident that our results hold for other samples.

Finally, conclusion validity is related to the relationship between treatment and outcomes. Among the measures against such threats, we considered the random nature of the selected algorithms by running 30 independent trials for each algorithm and each instance. We also adopted non-parametric statistical inference tests and effect-size measures to make sure that the requirements of statistical tests would be satisfied by our samples.

5. Conclusions

We proposed a new ILS based heuristic to solve the Software Module Clustering Problem. To demonstrate the effectiveness of our approach we made two experiments. The first study examined 228 different configurations of the proposed algorithm with the purpose to characterize the algorithm's performance in isolation. The goal of the second experiment was to compare the performance of different algorithms for the same class of problem. To achieved this task we compared the best version of our ILS_CMS heuristic, obtained in the first study, with two GA-based algorithms on a set of 40 benchmark instances, up to 483 modules and 7163 dependencies. An extensive comparative empirical analysis showed that ILS_CMS heuristic outperformed the two other approximated algorithms, in terms of solution quality and computation times. A basic ILS is simple to implement and can be used to heuristically solve other problems in the Search Based Software Engineering (SBSE) area. Future work aims to apply the proposed heuristic on instances from literature and compare our results with those recently presented in the work of Köhler et al. (2013). To make our work comparable, we also introduced 40 benchmarks instances, available at https://github.com/smcp/smcp_instances. We hope that these problems will constitute a comparison base for future resolution methods.

Acknowledgments

The two first authors thanks FAPERJ (Project E-26/110. 552/2010). Márcio de O. Barros thanks CAPES and CNPq for the financial support given to this project.

References

Abdeen, H., Sahraoui, H., Shata, O., Anquetil, N. and Ducasse, S. (2013), Towards Automatically Improving Package Structure while Respecting Original Design Decisions, *Proceedings of the Working Conference on Reverse Engineering*, 212-221.

Barros, M. (2012), An Analysis of the Effects of Composite Objectives in Multiobjective Software Module Clustering, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2012)*, 1205–1212.

Bavota, G., Carnevale, F., De Lucia, A., Di Penta, M. and Oliveto, R. (2012), Putting the Developer in-the-loop: an Interactive GA for Software Re-Modularization, *Proceedings of the 4th Symposium on Search Based Software Engineering (SSBSE 2012)*, 75-89.

Briand, L., Morasca, S. and Basili, V. (1999), Defining and Validating Measures for Object-based High-Level Design, *IEEE Transactions on Software Engineering*, 25, 722-743.

Doval, D., Mancoridis, S. and Mitchell, B.S. (1999), Automatic clustering of software systems using a genetic algorithm, *Proceedings of the Software Technology and Engineering Practice*, 73–81.

Falkenauer, E. (1996), A Hybrid Grouping Genetic Algorithm for Bin Packing, *Journal of Heuristics*, 2, 5–30.

Garey, M.R. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979

Hansen, P. and Jaumard, B. (1997), Cluster Analysis and Mathematical Programming, *Journal of Math. Program.*, 79, 191–215.

Köhler, V., Fampa, M. and Olinto, A. (2013), Mixed-Integer Linear Programming Formulations for the Software Clustering Problem, *Computational Optimization and Applications*, 55, 113–135.

Larman, C., Applying UML and Patterns: An Introduction to Object-Oriented Analysis and the Unified Process, Prentice Hall, Upper Saddle River, NJ, 2002.

Loureno, H.R., Martin, O.C. and Stützle, T., Iterated Local Search, in Handbook of Metaheuristics, Glover, F. and Kochenberger, G. A. (Eds.) *International Series in Operations Research and Management Science*, Kluwer Academic Publishers, 321–353, 2002.

Mahdavi, K., Harman, M. and Hierons, R.M. (2003), A Multiple Hill Climbing Approach to Software Module Clustering, *Proceedings of the International Conference on Software Maintenance (ICSM '03)*, 315–324.

Mancoridis, S., Mitchell, B.S., Chen, Y. and Gansner, E.R. (1999), Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures, *Proceedings of the IEEE International Conference on Software Maintenance*, 50–59.

Praditwong, K. (2011), Solving Software Module Clustering Problem by Evolutionary Algorithms, *Eighth International Joint Conference on Computer Science and Software Engineering*, 154–159.

Praditwong, K., Harman, M. and Xin Yao (2011), Software Module Clustering as a Multi-Objective Search Problem, *IEEE Transactions on Software Engineering*, 37(2), 264–282.

Räihä, O., A Survey on Search-Based Software Design, Technical Report D-2009-1, Department of Computer Sciences University of Tampere, March 2007.

Semaan, G.S., Botelho, S.L.V. and OCHI, L.S. (2011), Heurística Baseada em Busca Local Iterada para a resolução do Problema de Agrupamento de Sistemas Orientados a Objetos, *Simpósio Brasileiro de Pesquisa Operacional* (XLIII SBPO).

Síma, J. and Schaeffer, S.E., On the NP-Completeness of Some Graph Cluster Measures, in Proc. of the 32th Conf. on Current Trends in Theory and Practice of Computer Science, Wiedermann, J., Tel, G., Pokorný, J., Bieliková, M, and Stuller, J. (Eds.), *LNCS* 3831, 530–537, 2006.

Vargha, A. and Delaney, H.D. (2000), A critique and improvement of the CL common language effect size statistics of McGraw and Wong, *Journal of Educational and Behavioral Statistics*, 25, 101–132.

Wohlin, C., Runeson, P. and Host, M., *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers Group, Massachusetts, 2000.