UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

BUSCA EM VIZINHANÇA GRANDE APLICADA AO PROBLEMA DE CLUSTERIZAÇÃO DE MÓDULOS DE SOFTWARE

Marlon da Costa Monçores

Orientadores

Adriana Cesário de Faria Alvim

Márcio de Oliveira Barros

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 2015

Busca em Vizinhança Grande aplicada ao
Problema de Clusterização de Módulos de Software

Marlon da Costa Monçores

DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovada por:

Prof. Adriana Cesário de Faria Alvim, D.sc. - UNIRIO

Prof. Márcio de Oliveira Barros D.sc. - UNIRIO

Prof. Renata Araújo D.sc. - UNIRIO

Prof. Eduardo Uchoa Barbosa D.sc. - UFF

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO de 2015

Monçores, Marlon da Costa.
M738 Busca em vizinhança grande aplicada ao problema de clusterização de módulos de software / Marlon da Costa Monçores, 2015.
xii, 86 f. ; 30 cm

Orientadora: Adriana Cesário de Faria Alvim.
Coorientador: Márcio de Oliveira Barros.
Dissertação (Mestrado em Informática) - Universidade Federal do Estado do Rio de Janeiro, Rio de Janeiro, 2015.

1. Engenharia de software. 2. Clusterização de módulos de software. 3. Algoritmos computacionais. I. Alvim, Adriana Cesário de Faria. II. Barros, Márcio de Oliveira. III. Universidade Federal do Estado do Rio de Janeiro. Centro de Ciências Exatas e Tecnológicas. Curso de Mestrado em Informática. IV. Título.

CDD - 005.1

A meus pais, esposa, irmãos e amigos. Por todas as alegrias, tristezas e dores compartilhas.

Agradecimentos

Primeiramente eu agradeço a Deus, por ter me dado forças e por ter me sustentado, quando precisei. Obrigado Senhor.

Também agradeço aos meus pais por toda a dedicação de sempre. Ademais, a minha esposa Aline Sardow por todo o apoio e carinho a mim dedicado.

Agradeço a toda a equipe da Tamboro educacional, em especial à diretoria, pelo apoio e compreensão para que eu pudesse me dedicar as tarefas do mestrado. Deixo um agradecimento especial ao amigo Jerry Medeiros, que me ajudou muito, sobretudo nos momentos finais.

Agradeço também aos meus orientadores Adriana Alvim e Márcio Barros, que estiveram sempre presentes e disponíveis, compartilhando conhecimento e ideias a qualquer dia e hora. A dedicação de vocês é um exemplo que levarei comigo.

Além dos orientadores, eu deixo um agradecimento aos demais professores do PPGI e colegas de classe.

Monçores, Marlon da Costa. **Busca em Vizinhança Grande aplicada ao problema de Clusterização de Módulos de Software**. UNIRIO, 2015. 86 páginas. Dissertação de Mestrado. Departamento de Informática Aplicada, UNIRIO.

RESUMO

Softwares são compostos por um conjunto de módulos que desempenham funções distintas. Agrupar módulos com funções semelhantes em estruturas maiores (*clusters*) é importante para o bom entendimento e manutenção do software. A principal contribuição deste trabalho de dissertação de mestrado consiste no desenvolvimento de uma heurística robusta baseada na meta-heurística Busca em Vizinhança Grande aplicada ao problema de Clusterização de Módulos de Software (CMS). A heurística proposta utiliza os seguintes componentes: algoritmo gerador de solução inicial, algoritmo de destruição de parte da solução, algoritmo de reparação da solução e procedimento de redução do tamanho da instância. Com o objetivo de melhor caracterizar o algoritmo proposto e avaliar sua eficiência e eficácia, diversos experimentos computacionais foram realizados em um conjunto de 124 instâncias da literatura, a maior quantidade conhecida de instâncias reunidas em um experimento. A heurística proposta conseguiu melhorar a qualidade da solução de 58 instâncias (46,8% dos casos) com menor custo computacional quando comparando com os melhores resultados da literatura.

Palavras-chave: Busca em Vizinhança Grande, Clusterização de Módulos de Software, Engenharia de Software Baseada em Busca.

ABSTRACT

A set of modules playing their functions make a software functional. Clusters are groups of similar modules within bigger structures, and grouping them is important for software understanding and maintenance. The major contribution of this master's dissertation is developing a Large Neighborhood Search meta-heuristic based robust heuristic applied to the Software Module Clustering (SMC) problem. The heuristic proposed has the following components: initial solution generation algorithm, partial destruction algorithm, repair algorithm and instance size reduction procedure. The proposed algorithm efficiency and accuracy were analyzed by several computational experiments performed over 124 instances (the largest instance amount known for a single experiment) to improve algorithm characterization. The proposed heuristic did improve quality solution for 58 instances (46,8% of all) at lower machine cost even if compared to the best known results.

Keywords: Large Neighborhood Search, Software Module Clustering, Search-Based Software Engineering.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Nomenclaturas

CMS	Clusterização de Módulos de Software
CA	Algoritmo Construtivo Aleatório
CAMQ	Algoritmo Construtivo Aglomerativo MQ
CF	*Cluster Factor*
DA	Destrutivo Aleatório
DCA	Destrutivo Cluster Aleatório
DDMS	Destrutivo Diferença para a Melhor Solução
ECA	*Equal-Size Cluster Approach*
ES	*Effect-size*
EVM	*Evaluation Metric*
G	Grande
HC	*Hill Climbing*
ILS	*Iterated Local Search*
ILS_CMS	*Iterated Local Search* para o problema CMS
ISBSE	*Incremental Search-based Software Engineering*
LNS	*Large Neighborhood Search*
LNS_CMS	*Large Neighborhood Search* para o problema CMS
M	Média
MCA	*Maximizing Cluster Approach*
MDG	*Module Dependency Graph*
MG	Muito Grande
MHypEA	*Multi-objective Hyper-heuristic Evolutionary Algorithm*
MILP	*Mixed-Integer Linear Programming*
MQ	*Modularization Quality*
P	Pequena
RGMM	Reparativo Guloso Melhor Melhora
RGMMA	Guloso Melhor Melhora Aleatório
RGPMA	Guloso Pior Melhora Aleatório
RPMA	Primeira Melhora Aleatório
SMC	*Software Module Clustering*
SOLF	*Sum of Linear Fractional Functions*
VDN	*Variable-depth Neighborhood*
VLSN	*Very Large Scale Neighborhood*

1. Introdução

O processo de construção de um software envolve a criação de códigos-fonte em arquivos texto, chamados de módulos de software. Um software é composto por um ou mais módulos criados para executar tarefas específicas, podendo haver interação entre eles. Sempre que um módulo interage com outro, diz-se que existe uma dependência entre estes módulos [21].

Os requisitos de um software são as funcionalidades que um software deve desempenhar ou possuir [11]. Módulos de software são criados para implementar as funcionalidades necessárias para que o software atenda a todos os requisitos. Segundo Mancoridis et al. [20], os requisitos de um software tendem a mudar ao longo do tempo. Sempre que ocorre alguma alteração em algum requisito, os responsáveis pela manutenção do software têm que alterar o código dos módulos, e, eventualmente, quem fará a alteração pode não possuir total conhecimento da organização do código-fonte. Se as alterações forem feitas de forma *ad hoc*, ou seja, pontualmente para resolver um problema específico ou emergencial, esses módulos poderão ficar desorganizados ao ponto de impossibilitar a compreensão e, consequentemente, a execução de alterações futuras.

Uma alternativa para amenizar este problema é agrupar os módulos de um software em estruturas maiores, chamadas *cluster*. Esta organização pode, por exemplo, agrupar módulos de acordo com suas dependências ou de acordo com suas funcionalidades. Com esse nível de organização pode-se criar um modelo visual do software, que ajudará na compreensão do mesmo e em futuras manutenções. A organização de módulos de software em *clusters* pode ser chamada de clusterização.

Segundo Mancoridis et al. [20], técnicas de clusterização automática criam a organização dos módulos de software em *clusters* baseada nas dependências existentes entre os módulos. Os autores afirmam que estas técnicas podem ser usadas por programadores que não estão familiarizados com o sistema ou por arquitetos que queiram comparar a

clusterização existente com a automática para analisar as diferenças e propor melhorias.

A clusterização automática pode ser muito eficiente no custo de produção do software, visto que um software bem organizado é mais facilmente compreendido pelos programadores, o que implica na redução do número de falhas e em um tempo de desenvolvimento menor. Esta área tem sido, portanto, de grande interesse para a Engenharia de Software Baseada em Busca, levando ao desenvolvimento de ferramentas mais rápidas e mais eficientes para clusterização automática de software [10].

1.1 Definição do problema

O problema de Clusterização de Módulos de Software (CMS) (em inglês, *Software Module Clustering* - SMC) é o nome dado à tarefa de agrupar módulos de um software em estruturas maiores chamadas *clusters*, baseada nas dependências entre os módulos. O objetivo é unir em um mesmo *cluster* módulos que possuem muitas dependências entre si. Ao mesmo tempo, módulos que possuem poucas ou nenhuma dependência entre si deverão ficar em *clusters* separados.

Os módulos de software mais comuns incluem classe, macro e função. As dependências comuns incluem importação, exportação, herança, chamada de função e acesso a variável [20]. Por sua vez, *clusters* são agrupamentos de módulos, podendo representar pacotes, *namespaces* ou subsistemas do software, por exemplo. Segundo Pinto [22], *clusters* são estruturas com nível de abstração mais alto que os módulos do software.

O Problema de clusterização de módulos de software pode ser definido formalmente como a seguir: seja $N = \{M_1, M_2, \ldots, M_n\}$ o conjunto de n módulos, deseja-se encontrar um particionamento $P = \{C_1, C_2, \ldots, C_k\}$ de N em k *clusters*, com $\bigcup_{i=1}^{k} C_i = N$; $C_i \cap C_j = \emptyset$ para $1 \leq i, j \leq k$; $i \neq j$ e $C_i \neq \emptyset$ para $1 \leq i \leq k$. Isto é, a união de todos os *clusters* resulta no conjunto de módulos N, cada módulo pertence a exatamente um *cluster* e todos os *clusters* possuem ao menos um módulo. No caso do problema de clusterização de módulos a quantidade ideal de *clusters* não é um dado do problema e deve ser descoberta pelo método de solução proposto. Este tipo de problema de clusterização é conhecido como Problema de Clusterização Automática, em oposição ao Problema de k-Clusterização, em que o número k de *clusters* é um dado do problema, conhecido a *priori* [27].

Qualquer partição de módulos em *clusters* que obedeça às condições anteriores é uma solução válida para o problema CMS. Para se avaliar a qualidade da solução é necessária

a utilização de uma função objetivo.

Segundo Praditwong [24], coesão denota as dependências entre módulos dentro de um mesmo *cluster*, enquanto acoplamento é a medida do grau das dependências entre *clusters* distintos. Do ponto de vista do autor, encontrar uma boa solução para o problema CMS é procurar um equilíbrio entre acoplamento e coesão, recompensando *clusters* com muitas dependências internas e penalizando-os pelas dependências externas (com outros *clusters*). Uma boa estrutura é considerada aquela com alto grau de coesão e baixo grau de acoplamento [25]. Então, a função objetivo responsável por avaliar a qualidade de uma solução deve considerar, principalmente, os fatores coesão e acoplamento.

O problema CMS pode ser trabalhado apenas como um problema de grafos, desconsiderando as idiossincrasias relacionadas à linguagem de programação original do software. Usualmente é utilizado um grafo chamado de Grafo de Dependência de Módulos (em inglês, *Module Dependency Graph* - MDG) [20] para representar o problema CMS como um problema de particionamento de grafos. O MDG é um grafo direcionado onde os módulos são representados por vértices, dependências entre módulos são representados por arestas e *clusters* são as partições do grafo.

Considerando um MDG como entrada para o problema CMS, este também pode ser formalizado como mostrado a seguir [12]. Seja um MDG $G = (V, E)$, onde V é o conjunto de n nós (módulos) e $E = \{(u, v) | u, v \in V\}$ é o conjunto de dependências entre os módulos. O problema CMS consiste em encontrar uma partição de G em k *clusters* C_1,C_2,...,C_k, dividindo o conjunto V em componentes V_1,V_2,...,V_k com $\cup_{i=1}^{k} V_i = V$; $V_i \cap V_j = \emptyset$ para $1 \leq i, j \leq k$; $i \neq j$ e $V_i \neq \emptyset$ para $1 \leq i \leq k$. Cada nó em V_i é associado ao *cluster* C_i para $1 \leq i \leq k$.

Cada aresta no MDG pode ter um peso associado, representando o peso existente na dependência entre dois módulos. O peso das dependências entre os módulos no MDG é um guia importante para o comportamento dos algoritmos de clusterização. A atribuição de pesos pode ocorrer definindo-se um coeficiente relativo a cada tipo de dependência e, então, agregando-se o total de dependências com peso entre dois módulos [21]. Embora o presente trabalho utilize os pesos como parte da descrição das dependências, a definição de como estes pesos podem ser atribuídos aos diferentes tipos de dependência está fora do escopo deste trabalho.

1.2 Complexidade do problema

A complexidade do problema CMS está relacionada à quantidade de partições distintas em um MDG. Para se calcular o total de partições em um MDG considere n o número de módulos no MDG e k o número de partições. Tem-se que $1 \leq k \leq n$. Seja $S(n,k)$ o número total de k-$parties$ distintas para um MDG com n elementos. O número total de k-partições distintas satisfaz a equação:

$$S(n,k) = \begin{cases} 1 & \text{se } k = 1 \text{ ou } k = n \\ S_{n-1,k-1} + kS_{n-1,k} & \text{caso contrário} \end{cases} \quad (1.1)$$

Os elementos em $S_{n,k}$ são chamados de números de *Stirling* de segundo tipo (em inglês, *Stirling numbers of the second kind*) e crescem exponencialmente em função de n. Por exemplo, para calcular o número total de partições distintas de um MDG com $n = 5$ módulos é necessário calcular o somatório do número de k-partições distintas para $n = 5$ e $k = 1, 2 \ldots, n$. Neste caso, $\sum_{k=1}^{n} S(n,k) = S(5,1)+S(5,2)+S(5,3)+S(5,4)+S(5,5) = 52$. Seguindo o mesmo raciocínio, um MDG com 15 módulos possui 1.382.958.545 partições distintas. O crescimento de $S(n,k)$ pode ser limitado por $O(N!)$ [20, 21].

O problema genérico de particionamento de grafos é NP-Difícil. Tipicamente, outras interessantes funções guia para a clusterização, como pode ser considerada o MQ, por exemplo, igualmente tendem a corresponder a problemas NP-Difíceis [12]. Para tal classe de problemas, métodos heurísticos podem ser usados para encontrar soluções de boa qualidade e em um tempo razoável.

1.3 Objetivos

A literatura apresenta diversas propostas para resolução do problema CMS, incluindo buscas locais, algoritmos genéticos (mono-objetivo e multiobjetivo), busca local iterada e também técnicas de programação linear [3, 10, 12, 18, 19, 20, 22, 24, 25]. O HC com multipartidas aleatórias, como demonstrado no trabalho de Mitchell [21], é um algoritmo simples e alcança excelentes resultados, tanto em qualidade de solução quanto em tempo de processamento. Praditiwong et al. [25] concluíram que uma abordagem baseada em um algoritmo genético multiobjetivo consegue superar em qualidade as soluções geradas com o método HC, porém com tempo de processamento significativamente maior. No trabalho de Pinto [22] é utilizada uma busca local iterada mono-objetivo, obtendo bons resultados.

Desta forma, o presente trabalho propõe desenvolver e implementar uma heurística baseada na meta-heurística Busca em Vizinhança Grande [28] (em inglês, *Large Neighborhood Search* - LNS) para encontrar boas soluções para o problema CMS. Não foram encontrados trabalhos na literatura que utilizem esta meta-heurística para o problema CMS.

O presente trabalho utiliza uma avaliação mono-objetivo, com uma abordagem heurística baseada na meta heurística LNS. Os resultados obtidos pelo método proposto serão comparados com os melhores valores encontrados na literatura tanto em tempo de processamento quanto em qualidade de solução.

Os objetivos da pesquisa são:

(i) Propor uma heurística baseada na meta-heurística de Busca em Vizinhança Grande (LNS) [28] para o problema CMS;

(ii) Projetar, executar e analisar resultados de um estudo experimental utilizando uma heurística baseada na meta-heurística LNS, com diversas instâncias encontradas na literatura. Comparar os resultados obtidos pela heurística LNS desenvolvida com os resultados encontrados na literatura. Os fatores considerados no estudo comparativo são a eficacia (qualidade das soluções) e a eficiência (tempo de execução).

1.4 Organização da dissertação

O texto desta dissertação está distribuído em 5 capítulos. O Capítulo 1 contém a introdução, a definição do problema de clusterização de módulos de software (CMS) e os objetivos da pesquisa. No Capítulo 2 é apresentada uma revisão bibliográfica dos trabalhos relacionados ao problema CMS, assim como as principais funções objetivo utilizadas e também trabalhos relacionados à meta-heurística Busca em Vizinhança Grande. O Capítulo 3, por sua vez, contém a descrição detalhada da proposta de solução para o problema CMS, com as configurações e parâmetros utilizados na heurística construída baseada na meta-heurística Busca em Vizinhança Grande. O Capítulo 4 é dedicado à apresentação dos resultados obtidos nos experimentos computacionais realizados e comparação com resultados da literatura. Por fim, o Capítulo 5 contém as conclusões extraídas dos resultados dos experimentos, as contribuições da pesquisa em um contexto geral, limitações e propostas para trabalhos futuros.

2. Revisão Bibliográfica

Este capítulo apresenta os principais trabalhos, suas abordagens aplicadas ao problema CMS e os resultados obtidos. A primeira seção do capítulo apresenta as funções objetivo previamente utilizadas na literatura e suas respectivas complexidades computacionais. A segunda seção apresenta uma técnica de pré-processamento para redução do tamanho do grafo MDG. A terceira seção apresenta diversas abordagens utilizadas para encontrar soluções para o problema CMS.

2.1 Funções objetivo para o problema CMS

Segundo Mancoridis et al. [20], intraconectividade é definida como uma medida da conexão entre módulos agrupados em um mesmo *cluster*. Interconectividade é definida como uma medida da conexão entre módulos que estão em *clusters* diferentes. Os conceitos de intraconectividade e interconectividade estão fortemente relacionados ao problema CMS. Todas as funções objetivo encontradas na literatura para o problema CMS utilizam estes conceitos.

Ainda segundo Mancoridis et al. [20], um alto grau de intraconectividade indica um bom particionamento, pois os módulos que estão agrupados em um mesmo *cluster* pertencem a um mesmo subsistema. Um baixo grau de intraconectividade indica um particionamento pobre, pois os módulos agrupados em um mesmo *cluster* não pertencem a um mesmo subsistema. Um baixo grau de interconectividade também pode ser desejável em um particionamento, pois indica que *clusters* individuais são independentes.

As Seções 2.1.1 a 2.1.4 exibem as funções mono-objetivo encontradas na literatura para o problema CMS. Segundo Harman et al. [10], nenhuma destas funções é uma métrica, pois seus valores não são normalizados. Não ser normalizado significa que uma solução que possua o dobro do valor em relação a uma outra solução não possui o dobro

da qualidade. Além de não serem normalizadas, o valor das funções objetivo é um somatório sobre valores individuais de cada *cluster* e, consequentemente, não existe limite superior para o valor da função.

2.1.1 EVM

Tucker et al. [30] definem uma função objetivo chamada de EVM (em inglês, *Evaluation Metric*). O seu principal objetivo é mostrar a similaridade entre dois *clusters* distintos. Em um trabalho posterior, Harman et al. [10] definem formalmente o EVM(C) para a clusterização C como: seja uma clusterização C definida por um conjunto de m *clusters* $\{c_1, ..., c_m\}$, onde cada *cluster* c_i é expresso por um conjunto de módulos, tal que $c_i = \{m_1, ..., m_k\}$. Seja k_i o tamanho $|c_i|$ do i-simo *cluster* de C. Seja c_{ij} o j-simo módulo do i-simo *cluster* de C. Tem-se que:

$$EVM(C) = \sum_{i=1}^{m} h(c_i) \tag{2.1}$$

onde $h(c_i)$ é o valor do *cluster* c_i definido como:

$$h(c_i) = \begin{cases} \sum_{a=1}^{k_i-1} \sum_{b=a+1}^{k_i} L(c_{ia}, c_{ib}), & \text{se } k_i > 1 \\ 0, & \text{caso contrário} \end{cases} \tag{2.2}$$

onde $L(c_{xy}, c_{pq})$ é definido como

$$L(c_{xy}, c_{pq}) = \begin{cases} 1, & \text{se existir uma dependência de } c_{xy} \text{ para } c_{pq} \text{ ou vice-versa} \\ -1, & \text{caso contrário} \end{cases} \tag{2.3}$$

O Algoritmo 2.1 detalha o pseudocódigo de cálculo do EVM conforme as ideias apresentadas por Harman [10], adaptado para receber como entrada o grafo $MDG = G(V = \{1, 2, \ldots, n\}, E)$ e uma clusterização $C = \{c_1, ..., c_m\}$. Considere a função $g(i)$ que, para cada módulo $i \in V$, retorna o identificador do *cluster* onde o módulo i se encontra.

2.1.2 MQ Básico

Mancoridis et al. [20] definem três funções objetivo para se calcular a qualidade da clusterização de um grafo. As funções objetivo definidas têm diversas similaridades e são chamadas de Qualidade da Modularização (em inglês, *Modularization Quality* - MQ).

Algoritmo 2.1 Algoritmo de cálculo do EVM ($G(V = \{1, 2, \ldots, n\}, E); C = \{c_1, \ldots, c_m\}$)

```
Seja EVM = 0
para u = 1 até n − 1 faça
    para v = u + 1 até n faça
        se g(v) = g(u) então
            se ∃((u, v) ou (v, u)) ∈ E então
                EVM ← EVM + 1
            senão
                EVM ← EVM − 1
            fim se
        fim se
    fim para
fim para
```

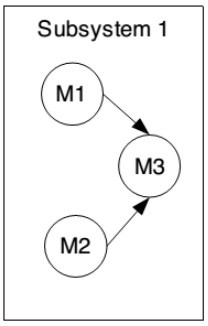
Diversos autores [6, 18, 19, 20, 21] utilizam a função objetivo MQ Básico (em inglês, *BasicMQ*), que pode ser formalmente descrita por: seja A_i a intraconectividade de um *cluster* i que possui N_i módulos e μ_i dependências internas.

$$A_i = \frac{\mu_i}{N_i^2} \tag{2.4}$$

Seja E_{ij} a interconectividade entre os *clusters* i e j consistindo dos módulos N_i e N_j, respectivamente e ε_{ij} o total de dependências entre os *clusters*.

$$E_{ij} = \begin{cases} 0 & \text{se } i = j \\ \frac{\varepsilon_{ij}}{2N_iN_j} & \text{se } i \neq j \end{cases} \tag{2.5}$$

A Figura 2.1 ilustra um exemplo de cálculo de intraconectividade para um *cluster*, enquanto a Figura 2.2 ilustra um exemplo de cálculo de interconectividade para um par de *clusters*.



$N_i = 3$

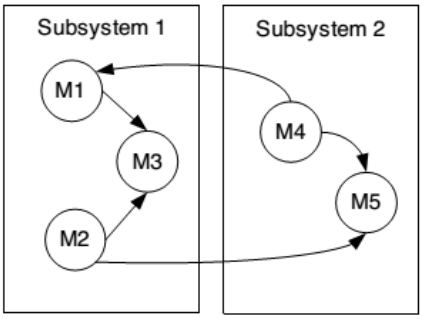
$\mu_i = 2$

$N_i^2 = 9$

$A_1 = \frac{\mu_i}{N_i^2} = \frac{2}{9} = 0.2222...$

Figura 2.1: Exemplo de cálculo de intraconectividade utilizando o MQ Básico [21].

Com os valores de interconectividade e intraconectividade definidos é possível calcular a qualidade da modularização. O MQ Básico de um grafo particionado em k *clusters*, onde A_i é a intraconectividade do i-simo *cluster* e E_{ij} é a interconectividade entre o i-



$$N_1 = 3$$
$$N_2 = 2$$
$$\varepsilon_{1,2} = 2$$
$$E_{1,2} = \frac{\varepsilon_{1,2}}{2N_1N_2} = \frac{2}{12} = 0.1666....$$

Figura 2.2: Exemplo de interconectividade utilizando o MQ Básico [21].

simo e o j-simo *clusters*, é definido por:

$$\text{MQ Básico} = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{ij} & \text{se } k > 1 \\ A_1 & \text{se } k = 1 \end{cases} \tag{2.6}$$

A função objetivo MQ demonstra um balanceamento entre interconectividade e intraconectividade por favorecer a criação de *clusters* muito coesos, enquanto penaliza a criação de muitas interconexões. Este balanceamento é estabelecido pela subtração da média da interconectividade pela média da intraconectividade [20].

2.1.3 MQ Turbo

A função MQ Turbo (em inglês, *TurboMQ*) criada por Mitchell tem como objetivo superar duas limitações do MQ Básico: o MQ Turbo suporta MDGs com peso e tem um desempenho mais rápido que o MQ Básico [21].

Para calcular o MQ Turbo é necessário somar os fatores de clusterização (em inglês, *Cluster Factor* - CF) para cada *cluster* no MDG particionado. O CF_i para um *cluster* i, tal que $1 \leq i \leq k$, onde k é a quantidade de *clusters*, é definido como uma razão entre o peso total das arestas internas e metade do peso total das arestas externas. O peso da aresta externa é dividido por dois para penalizar igualmente ambos os *clusters* conectados por cada dependência externa [21]. Um exemplo de cálculo do MQ Turbo é visto na Figura 2.3, onde cada uma das arestas possui peso equivalente a 1.

Seja μ_i o número de arestas internas de um *cluster* i e $\varepsilon_{i,j}$ e $\varepsilon_{j,i}$ o número de arestas externas entre dois *clusters* distintos i e j, respectivamente, se os pesos das dependências não forem fornecidos, todos os pesos são tratados com o valor 1 e assume-se que não existem autorrelacionamentos. As equações 2.7 e 2.8 demonstram o cálculo do MQ Turbo.

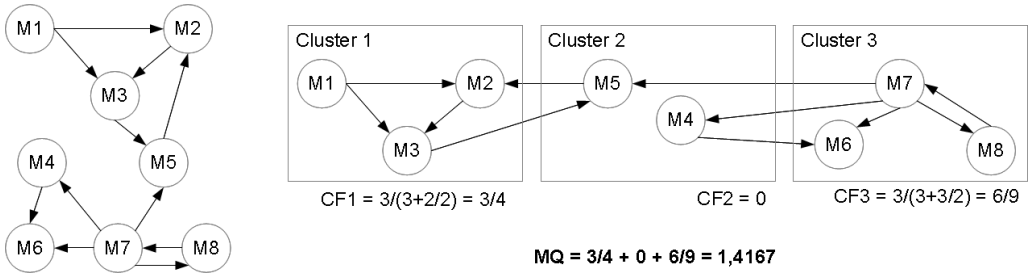


Figura 2.3: Exemplo de cálculo de MQ Turbo.

$$\text{MQ Turbo} = \sum_{i=1}^{k} CF_i \tag{2.7}$$

$$CF_i = \begin{cases} 0 & \text{se } \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \sum_{j=1, j\neq i}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & \text{caso contrário} \end{cases} \tag{2.8}$$

De acordo com Köhler et al. [12], o fator de clusterização CF pode ser visto como uma generalização de uma medida conhecida como densidade relativa que mede a qualidade de um *cluster* em um grafo. Síma and Schaeffer [29] provam que o problema de decisão associado ao problema de otimização que busca encontrar uma clusterização ótima com respeito à medida de densidade relativa é NP-Completo.

2.1.4 MQ Turbo Incremental

A função MQ Turbo Incremental (*ITurboMQ*) é uma otimização da função MQ Turbo com base na observação de que o valor da função objetivo é a soma dos fatores de clusterização. Quando um módulo é transferido de um *cluster* para outro, apenas os dois *clusters* envolvidos têm os seus valores alterados. Todos os outros *clusters* permanecem iguais quanto aos valores de seus respectivos μ e ε. Com isso não é necessário que o MQ seja totalmente recalculado a cada transferência: o MQ pode ser calculado de forma incremental, apenas atualizando os valores dos *clusters* que foram alterados [21].

Diversos autores [3, 12, 14, 24, 25, 27] também utilizam o MQ Turbo Incremental em seus trabalhos.

2.1.5 Complexidade das funções objetivo

Um fator relevante na comparação entre as funções objetivo é a complexidade computacional de cada uma delas.

Considerando $|V|$ a quantidade de módulos e $|E|$ a quantidade de dependências em um MDG, tem-se que, em grafos direcionados, o $|E|$ pode ser tão grande quanto $|V|^2$. Porém, sabe-se que para grafos de software $|E|$ é muito mais próximo de $|V|$ do que de $|V|^2$ [21]. Assumindo que a complexidade para se descobrir em qual *cluster* está um módulo é $O(1)$ e que para saber se dois módulos estão no mesmo *cluster* é $O(1)$, pode-se calcular a complexidade das funções objetivo EVM e MQ.

A complexidade da função objetivo *EVM* é definida pelo somatório dos valores obtidos em cada *cluster*. Para se somar os valores dos k *clusters* tem-se $O(k) = O(V)$. Para se calcular o valor de cada *cluster* é necessário percorrer cada par de módulos existente dentro do *cluster*, ou seja $O(|V^2|)$. Juntando-se os dois processamentos, tem-se $O(|V| + |V|^2) = O(|V^2|)$.

A complexidade da função MQ Básico é composta pela média das intraconectividades menos a média das interconectividades. O número máximo de k *clusters* em um MDG é $|V|$. Para se calcular a intraconectividade é necessário percorrer todos os *clusters*. Essa operação custa $O(k) = O(|V|)$. Após, todas as dependências tem que ser percorridas para verificar se é uma intraconectividade. Essa operação custa $O(|E|)$. Então, a complexidade para se calcular a intraconectividade é $O(|V| + |E|) = O(|E|)$. Por sua vez, para a interconectividade, é necessário somar o valor de cada par de *clusters*. Esta operação tem complexidade $O(|V|^2 \times |E|)$. Sendo assim, a complexidade da função é tida por $O(|E| + (|V|^2 \times |E|)) = O(|V|^2 \times |E|)$. Supondo, na prática, que $O(|E|) \approx O(|V|)$ então a complexidade é dada por $O(|V^3|)$ [21].

A complexidade da função MQ Turbo é composta pelo somatório dos valores de MF para cada k *cluster*. Para se somar os valores dos k *clusters* tem-se $O(k) = O(|V|)$. Para se calcular o valor do MF de cada *cluster* é necessário percorrer cada uma das dependências de cada um dos módulos. A complexidade para este procedimento é $O(|V| + |E|)$. Então, a complexidade final é $O(|V| + |E| + |V|) = O(|V| + |E|) = O(|E|)$. Supondo, na prática, que $O(|E|) \approx O(|V|)$, então a complexidade é dada por $O(|V|)$ [21].

A complexidade da função MQ Turbo Incremental é composta apenas observando as dependências entre os dois módulos envolvidos em uma operação de movimentação. Cada módulo poderá ter no máximo $|E|$ e na média $|\frac{E}{V}|$ dependências. Então, a complexidade pode ser considerada inicialmente como $O(E)$. Porém, durante o processo de clusterização, a função será utilizada diversas vezes em diferentes partições distintas do grafo. Pode-se, então, considerar a complexidade do caso médio $O(|\frac{E}{V}|)$. Supondo, na prática, que $O(|E|) \approx O(|V|)$, a complexidade da função é $O(1)$.

A Tabela 2.1 exibe os valores de complexidade de cada uma das funções objetivo analisadas.

Tabela 2.1: Complexidade das funções objetivo. Pode-se perceber que o MQ Turbo Incremental possui a menor ordem de grandeza entre os métodos comparados.

Função	Complexidade
EVM	$O(\|V\|^2)$
MQ Básico	$O(\|V\|^3)$
MQ Turbo	$O(\|V\|)$
MQ Turbo Incremental	$O(1)$

2.2 Redução de grafos MDG

Redução de grafos MDG é uma forma de pré-processamento capaz de reduzir a quantidade de módulos e de dependências no grafo. Com a redução do tamanho do MDG, o número total de clusterizações possíveis também é reduzido e, com isso, o problema pode ser resolvido mais rapidamente. O método de redução criado por Köhler et al. [12] para o problema CMS garante que, dado um grafo MDG, o valor ótimo do MQ deste MDG é igual ao valor ótimo do MQ do MDG reduzido.

Para que seja efetuado o pré-processamento é necessário que o MDG seja analisado como um MDG não direcionado e com peso. Porém, o MDG é tipicamente um grafo direcionado. Assim, são necessários até dois passos para se chegar ao grafo não direcionado e com peso. O primeiro passo é adicionar peso para todas as dependências dos MDGs que não possuem peso. Isso é feito simplesmente considerando cada dependência com peso 1. Para se remover o direcionamento das arestas, basta juntar arestas que sejam opostas e somar os pesos de suas respectivas dependências, ou seja, se existe uma aresta que vai de um módulo m_1 até m_2 e outra aresta que vai de m_2 até m_1, estas arestas serão substituídas por uma aresta entre m_1 e m_2 sem direção, com peso igual à soma das arestas anteriores. A Figura 2.4 exibe um exemplo de transformação de um MDG direcionado em um MDG não direcionado.

Sejam um MDG $= \bar{G}(\bar{V}, \bar{E})$ e os pesos da aresta $\bar{c}_{uv}$, para cada $(u, v) \in \bar{E}$. Sem perda de generalidade, seja $\bar{c}_{uv} \geq 1$, $\forall (u, v) \in E$. Define-se um grafo não direcionado $G = (V, E)$ tal que $V = \bar{V}$ e $E = \{(u, v) \in \bar{E} | u < v\}$ e seja $c_{uv} = \bar{c}_{uv} + \bar{c}_{vu}$, para cada $(u, v) \in E$ (se $(v, u) \notin \bar{E}$, considere $\bar{c}_{vu} = 0$) [12].

Após a criação do MDG não direcionado e com pesos é possível executar o pré-processamento para redução do grafo MDG baseado no teorema apresentado a seguir:

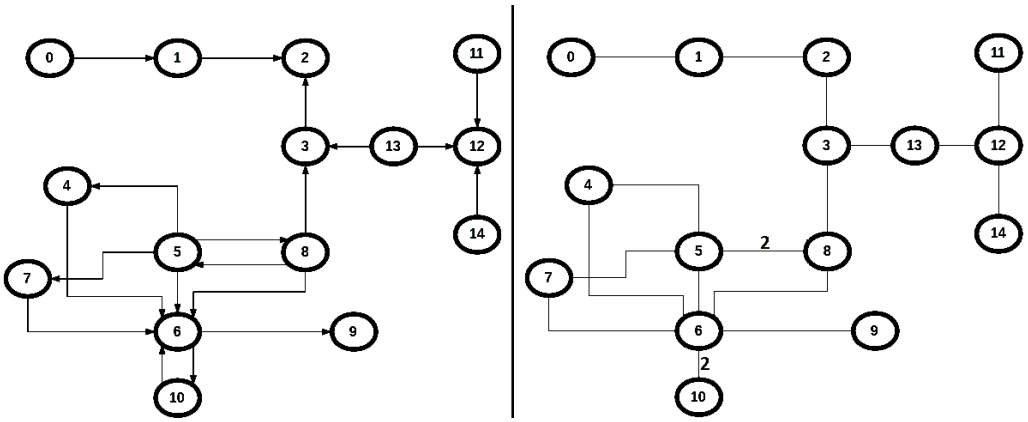


Figura 2.4: Exemplo de um MDG direcionado sendo transformado em um MDG não direcionado e com peso. Arestas sem valor possuem o peso = 1.

Teorema 1. *[12] Seja* $G(V, E)$ *o grafo não direcionado e com pesos dado como entrada para o problema CMS. Seja* $u \in V$ *um nó com grau igual a um e seja* $v \in V$ *adjacente à* u*. Então na solução ótima de CMS,* u *e* v *estão no mesmo cluster.*

Demonstração. cf. [12, pág. 121, Teorema 4.1] □

Para este pré-processamento, considere como entrada para o problema CMS um MDG não direcionado e com peso, chamado $G(V, E)$. Seja u um vértice (módulo) em V com grau igual a 1 (dependência com apenas um outro módulo) e seja v esse único vértice adjacente ao vértice u. O pré-processamento consiste em remover u de V e adicionar uma auto dependência (v, v) em E, com peso equivalente ao peso da dependência removida, ou seja, $c_{vv} = c_{uv}$. O MDG gerado pelo pré-processamento será utilizado como entrada para o procedimento, exato ou heurístico, usado para resolver o CMS. Uma vez encontrada a solução, ótima ou sub-ótima, é possível reverter o MDG reduzido e retornar ao MDG original, caso seja necessário. Entretanto, para cálculos de MF e MQ não existe tal necessidade, pois os seus respectivos valores são os mesmos tanto para os grafos reduzidos quanto para os grafos originais [12].

Note que o pré-processamento de redução de grafos MDG pode adicionar alguns autorrelacionamentos. Então é necessário alterar o cálculo do MQ descrito na equação 2.8 para que os autorrelacionamentos passem a ser avaliados. Para isso, basta remover a restrição $i \neq j$ do somatório.

A redução é baseada na observação de que módulos com grau 1 devem sempre ficar no mesmo *cluster* que seu módulo adjacente. Pois, quando juntos, a dependência existente é considerada como intraconectividade. Ao ser separado, a intraconectividade deixa de existir, sendo substituída por uma interconectividade. Então o *cluster* de origem terá o seu

valor de MF reduzido. Por sua vez o *cluster* que receber o módulo não terá acréscimo em seu valor de intraconectividade. Ao invés, o *cluster* terá um aumento na interconectividade e terá o seu valor de MF reduzido.

Quando o módulo possui grau 2 e um autorrelacionamento, não é possível aplicar a simplificação novamente, uma vez que o *cluster* que receber o módulo terá um aumento no seu valor de intraconectividade, o que poderá levar a um aumento no MF. Como um *cluster* terá diminuição no valor de MF e o outro poderá ter um aumento, o valor de MQ poderá ser maior ao separar o módulo.

A Figura 2.5 exibe um exemplo de um MDG não direcionado e com peso antes e depois do pré-processamento de redução.

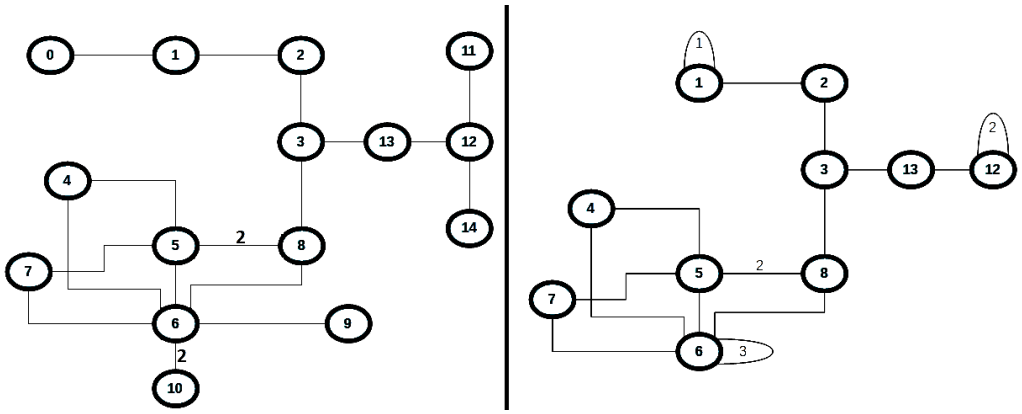


Figura 2.5: Exemplo de um MDG não direcionado e com peso antes e depois de sua redução. Nota-se uma redução no número de módulos e de dependências.

2.3 Algoritmos para o problema CMS

Existem dois tipos distintos de algoritmos possíveis para um problema que seja da classe de problemas NP-Difícil, inclusive para o problema CMS: algoritmos ótimos (ou exatos) e algoritmos heurísticos (ou sub-ótimos).

Os algoritmos ótimos encontram a melhor solução possível para um dado problema. Entretanto, para problemas NP-Difíceis existe uma limitação do tamanho do problema que estes algoritmos podem tratar considerando tempo computacional razoável. Alternativamente, métodos heurísticos são capazes de encontrar solução de qualidade, sem garantia de otimalidade, e em tempo computacional razoável.

A seguir é apresentado um algoritmo ótimo para o problema CMS. Todos os demais algoritmos apresentados são sub-ótimos, embora alcancem a solução ótima, eventualmente.

2.3.1 Algoritmos exatos

Em seu trabalho Mancoridis et al. [20] apresentam um algoritmo capaz de resolver de forma exata o problema CMS. A técnica utilizada é de enumeração explícita e compara todas as possíveis formas de particionar os módulos. Como visto na Seção 1.2, o número total de partições possui crescimento exponencial em função da quantidade de módulos. Assim, apenas problemas com poucos módulos podem ser resolvidos usando esta técnica. Ainda em seu trabalho, Mancoridis et al. [20] afirmam conseguir resolver MDGs de no máximo 15 módulos. O Algoritmo 2.2 apresenta o pseudo-código de um algoritmo exato para o problema CMS.

Algoritmo 2.2 Algoritmo exato para o problema CMS

```
Seja S = {M_1, M_2, ..., M_n}, onde cada M_i é um módulo
Gerar todas as partições do conjunto S
Avaliar o MQ para cada partição
A partição com o maior MQ é a solução ótima
```

Muitos problemas de otimização compartilham o mesmo subproblema: otimizar o valor de uma função objetivo que é o composta pelo somatório de frações lineares (frações com numerador e denominador sem expoentes) [5]. Observando-se a função objetivo MQ, percebe-se que ela é composta pelo somatório dos valores obtidos por cada *cluster* e cada um desses valores é uma fração linear. Então, é possível tratar o problema CMS como uma soma de funções fracionárias lineares (em inglês, *Sum of Linear Fractional Functions* - SOLF).

Outra abordagem exata é a programação linear inteira mista (em inglês, *Mixed-Integer Linear Programming* - MILP). Nessa abordagem, o problema é dividido em uma definição e uma série de equações com valores binários (verdadeiro ou falso). Resolver um problema com a abordagem MILP consiste em resolver todas as equações para valor verdadeiro. Quando isso ocorre, a definição terá um valor. O objetivo para o problema CMS é encontrar o maior valor possível. No trabalho de Li é apresentada também uma técnica de *branch and bound* capaz de resolver o problema MILP para a solução ótima [16].

Köhler et al. [12] definem três formulações para o problema CMS. A primeira formulação trabalha o problema CMS como SOLF e as outras duas formulações trabalham o problema como MILP. As técnicas exatas são aplicadas a 45 MDGs distintos, com número de módulos variando de 6 até 62. A segunda formulação MILP proposta, $\mathrm{MILP}^{+}_{2,1}$, consegue resolver para a solução ótima todos os MDGs analisados com até 29 módulos. Além disso, o maior MDG resolvido para a solução ótima possui 62 módulos. Ao total, dos 45 MDGs analisados, 34 foram resolvidos para a solução ótima.

Kramer et al. [13] utilizam uma abordagem MILP similar a utilizada no trabalho de Köhler et al. [12] e avaliam o desempenho desta abordagem para os mesmos 45 MDGs. A principal diferença entre os trabalhos é a técnica de geração de colunas utilizada, chamada de *Staged Column Generation*. O objetivo desta técnica é diminuir a carga computacional sobre o resolvedor MILP. O algoritmo proposto consegue resolver de forma ótima todos os 45 MDGs avaliados.

2.3.2 Busca Local

A busca local do tipo subida de encosta (em inglês, *Hill Climbing* - HC) é um algoritmo que começa com uma solução inicial e iterativamente procura por soluções vizinhas que sejam melhores, até que não seja mais possível encontrar melhores soluções. Uma busca local do tipo HC proposta por Mancoridis [20] possui vários inícios com soluções aleatórias. A vizinhança utilizada pelos autores é definida por todas as soluções alcançáveis trocando-se um módulo de *cluster*. Seja para um *cluster* existente ou para um novo.

Existem diversas abordagens para determinar a forma como uma busca local escolhe um vizinho melhor: escolher o primeiro vizinho melhor; escolher o melhor entre todos os vizinhos; escolher o pior vizinho melhor, dentre outras. Mitchell [21] utiliza uma abordagem onde apenas uma quantidade percentual do total de vizinhos são visitados. O melhor vizinho visitado será escolhido. Caso não encontre um vizinho melhor, os demais vizinhos serão visitados e o primeiro vizinho melhor será escolhido. Por exemplo, caso o valor percentual escolhido seja 0%, o primeiro vizinho melhor será retornado. Caso seja escolhido um valor de 100%, todos os vizinhos serão visitados e comparados e o melhor vizinho será o escolhido.

Algoritmo 2.3 Algoritmo subida de encosta para o problema CMS

```
Seja S = {M1, M2, ..., Mn}, onde cada Mi é um módulo
Gerar uma partição inicial P aleatória do conjunto S
repetir
    Aleatoriamente escolher uma partição vizinha melhor bNP (partição com: MQ(bNP) > MQ(P) )
    se bNP for encontrado então
        P ← bNP
    fim se
até não encontrar partições vizinhas melhores
Partição P é a solução sub-ótima
```

A técnica de construção de blocos (em inglês, *building blocks*) é utilizada no trabalho de Mahdavi et al. [18]. Essa técnica consiste em executar múltiplas subidas de encosta e identificar similaridades entre as soluções encontradas, mais especificamente, encontrar módulos que foram associados aos mesmos *clusters* um percentual de vezes. Esse percentual é chamado de corte. Se o corte for, por exemplo, 10 %, os módulos pertencerão

ao mesmo bloco se aparecerem nos mesmos *clusters* em pelo menos 10% das soluções encontradas pelas subidas de encosta. Após essa identificação, uma nova série de subida de encosta é executada utilizando os blocos construídos anteriormente.

A utilização dessa técnica permite a diminuição do espaço de busca, uma vez que alguns módulos passam a ter posição fixa na segunda rodada de otimização. Os autores comparam a técnica de construção de blocos com a técnica de subida de encosta com múltiplas partidas e constatam que houve melhora em todos os MDGs estudados, tanto para os MDGs com peso quanto para os sem peso. Outra conclusão é que a técnica de construção de blocos funciona melhor para MDGs maiores [18].

Os autores testaram valores de corte entre 10% e 100%, com intervalos de 10%, ou seja, 10 valores de corte foram testados. Para todos os valores de corte testados houve melhoria no valor do MQ alcançado. Em especial, para os valores de corte 10% e 20%, onde, todos os MQs gerados na segunda rodada foram melhores que os obtidos na primeira [18].

Barros [4] introduz o conceito de engenharia de software baseada em busca incremental (em inglês, *incremental search-based software engineering* - ISBSE). Nesse cenário, a clusterização manual, feita durante o desenvolvimento do software, é considerada como solução inicial e um grau de perturbação é aplicado. Com isso, as soluções geradas são mais facilmente compreendidas pelos desenvolvedores, pois serão parecidas com a organização do software criada inicialmente.

A otimização é feita de forma incremental, ou seja, são executadas varias rodadas de otimização. Após cada rodada, os desenvolvedores podem analisar a solução gerada e descartar algumas alterações. A próxima iteração não efetuará alterações que tenham sidos restringidas anteriormente. Com essa abordagem é possível organizar apenas uma parte do software e, ao mesmo tempo, não deixar os desenvolvedores confusos com uma arrumação dos módulos em *clusters* totalmente diferente da original.

Neste cenário, a busca utilizando o MQ como função objetivo precisou de menos rodadas e menos restrições (proibições manuais) para alcançar a mesma performance que uma busca utilizando a função objetivo EVM em aproximadamente 80% dos grafos analisados pelo autor. Além disso a busca guiada pelo MQ gera, aproximadamente, 12 vezes menos retrabalho (um módulo é movido de um *cluster* A para B e então é movido novamente de B para C, sendo $A \neq B \neq C$, em iterações seguidas) que o EVM.

2.3.3 Algoritmos Genéticos

Algoritmos genéticos mimetizam o processo de seleção natural. Soluções são geradas utilizando técnicas inspiradas em processos biológicos como herança, *crossover* e mutação.

O funcionamento de um algoritmo genético pode ser descrito em alguns passos. Primeiramente, são criadas diversas soluções (população inicial). Então, essas soluções são combinadas (evoluídas) e selecionadas de forma a gerar um outro conjunto de soluções (seleção). A geração da população inicial pode ser feita de forma aleatória. Após, as soluções são combinadas duas a duas (reprodução), gerando novas soluções. As soluções geradas nesse processo de reprodução contêm apenas informações vindas das duas soluções que foram combinadas. A escolha de como será feita a combinação das soluções é feita por um componente chamado de operador de *crossover*.

Para evitar que a combinação gere soluções muito parecidas ou até idênticas, é adicionado também um operador de mutação, que é responsável por realizar alterações aleatórias na solução. A cada geração, para garantir uma maior diversidade nas soluções, pode-se também gerar algumas soluções aleatórias. Então, algumas das novas soluções são escolhidas (seleção) e levadas para a próxima iteração. O Algoritmo 2.4 exibe um pseudo-código de uma heurística genética.

Algoritmo 2.4 Algoritmo genético para o problema CMS

Seja $S \leftarrow \{M_1, M_2, ..., M_n\}$, onde cada M_i é um módulo em um software
Seja $g \leftarrow 0$
Gerar uma população $P(g)$ com partições aleatórias de S
repetir
    Calcular o valor da função objetivo de cada solução em $P(g)$
    Adicionar em $P(g+1)$ a melhor solução de $P(g)$
    Adicionar em $P(g+1)$ soluções geradas através da combinação de elementos de $P(g)$ utilizando crossover e mutação
    Adicionar em $P(g+1)$ algumas soluções aleatórias
    $g \leftarrow g+1$
até alcançar algum critério de parada
Calcular o valor da função objetivo de cada solução em $P(g)$
retornar A melhor solução em $P(g)$

No trabalho de Mancoridis [20], os autores trabalham o problema CMS como um problema de otimização e apresentam uma ferramenta chamada *Bunch*, que possui três algoritmos distintos para buscar soluções para o problema CMS: uma busca exaustiva, um algoritmo de subida de encosta e um algoritmo genético. Doval et al. [6] fizeram uma continuação do trabalho anterior, onde o problema CMS é tratado apenas com um algoritmo genético. Por fim, Mancoridis et al. [19] adicionam três novas funcionalidades à ferramenta *Bunch*: detecção e atribuição de módulos omnipresentes (atribui módulos a

uma lista ou duas listas; estes módulos não participarão do processo de clusterização; cada lista será considerada um *cluster*); clusterização dirigida pelo usuário (possibilita ao usuário clusterizar alguns módulos manualmente) e manutenção incremental (utiliza a última clusterização como entrada e permite adicionar novos módulos e suas dependências). A ferramenta encontra-se disponível em https://www.cs.drexel.edu/~spiros/.

Algoritmos genéticos trabalham com uma forma de codificação da solução. A escolha da codificação é extremamente importante para o desempenho do algoritmo. Uma codificação ruim poderá gerar uma busca muito demorada e que não consegue alcançar soluções de boa qualidade [21]. Segundo Praditwong [24], existem duas codificações distintas para o problema CMS.

- ***Group Number Encoding* - GNE**: É a representação de solução mais conhecida de algoritmo genético para problemas de clusterização. Nessa representação, uma solução é um vetor com n elementos, e, no caso, cada elemento representa um módulo. Cada módulo contém uma identificação especificando a qual *cluster* o módulo pertence;
- ***Grouping Genetic Algorithms* - GGA**: Essa representação consiste de dois componentes: O primeiro componente mapeia cada módulo para um grupo e o segundo faz a listagem dos grupos.

Praditwong [24] fez um comparativo entre as abordagens GNE e GGA utilizando 17 MDGs construídos a partir de softwares existentes. A informação de peso entre as dependências só esteve presente em 10 dos MDGs. A abordagem genética GGA consegue melhores resultados em 9 dos 10 MDGs com peso estudados. Por outro lado, para os MDGs sem peso a abordagem GNE alcança melhores resultados em 5 dos 7 MDGs avaliados. Em outro trabalho, Praditwong et al. [25] concluem ainda que o método de subida de encosta é mais simples e mais rápido que a abordagem genética. Os tempos de processamento do método de subida de encosta chegam a ser duas ordens de grandeza menor do que os tempos de processamento do algoritmo genético.

2.3.4 Algoritmos Genéticos multiobjetivo

Algoritmos genéticos multiobjetivo utilizam diversas funções objetivo para qualificar as soluções. Assim, a comparação entre duas soluções não é tão simples quanto em um algoritmo mono-objetivo. De acordo com Praditwong et al. [25], o valor de uma função objetivo $F(\overline{x_1})$ de uma solução candidata $\overline{x_1}$ é definido em termos de cada uma das funções

f_i constituintes. Para que uma solução $\overline{x_1}$ seja considerada melhor que uma outra solução $\overline{x_2}$, é necessário que $\overline{x_1}$ seja melhor que $\overline{x_2}$ em pelo menos uma das funções objetivo constituintes. Ao mesmo tempo, $\overline{x_1}$ não pode ser pior que $\overline{x_2}$ em nenhuma das outras funções constituintes. Quando isso ocorre, diz-se que a solução $\overline{x_1}$ domina a solução $\overline{x_2}$. Essa relação de domínio pode ser expressa pela Equação 2.9, chamada de ótimo de Pareto:

$$F(\overline{x_1}) > F(\overline{x_2}) \Leftrightarrow \forall i \cdot f_i(\overline{x_1}) \geq f_i(\overline{x_2}) \wedge \exists i \cdot f_i(\overline{x_1}) > f_i(\overline{x_2}). \tag{2.9}$$

Praditwong et al. [25] definem duas abordagens multiobjetivo para o problema CMS. Cada uma das abordagens possui cinco funções objetivo.

- **Abordagem maximizadora de *clusters***: A abordagem maximizadora de *clusters* (em inglês, *Maximizing Cluster Approach* - MCA) é composta pelas seguintes funções objetivo: maximizar a intraconectividade de todos os *clusters*; minimizar a interconectividade de todos os *clusters*; maximizar o número de *clusters*; maximizar o MQ e minimizar o número de *clusters* isolados, ou seja, *clusters* com apenas um módulo.
- **Abordagem *clusters* de mesmo tamanho** A abordagem multiobjetivo *clusters* de mesmo tamanho (em inglês, *Equal-Size Cluster Approach* - ECA) é composta pelas seguintes funções objetivo: maximizar a intraconectividade de todos os *clusters*; minimizar a interconectividade de todos os *clusters*; maximizar o número de *clusters*; maximizar o MQ e minimizar a diferença de tamanho entre o *cluster* com mais módulos e o *cluster* com menos módulos.

Os resultados obtidos por Praditwong et al. [25] indicam que para MDGs sem peso o método de subida de encosta supera o método MCA, porém para MDGs com peso o MCA supera o subida de encosta. Por sua vez o ECA supera o método de subida de encosta em ambos os tipos de MDGs. Comparando-se as duas abordagens multiobjetivo, o ECA também supera o MCA. Por fim, o método ECA é capaz de produz soluções melhores do que as soluções geradas pelos métodos genéticos mono-objetivo. Contudo, o tempo de processamento é significativamente maior.

2.3.5 Hiper-heurística

Hiper-heurística é uma abordagem que consiste em utilizar várias heurísticas diferentes em um mesmo processo de busca, com o intuito de alcançar boas soluções, fugindo de

ótimos locais. Assim, é possível percorrer o espaço de busca de maneiras diferentes, e aproveitar as características de cada heurística. O funcionamento de uma hiper-heurística consiste em escolher qual heurística será utilizada em um momento específico da busca, favorecendo as heurísticas que conseguem obter bons resultados, e desfavorecendo as heurísticas que não conseguem gerar boas soluções.

Kumari e Srinivas [14] utilizam uma hiper-heurística multiobjetivo baseada em algoritmos evolutivos (em inglês, *Multi-objective Hyper-heuristic Evolutionary Algorithm* - MHypEA). Os autores criam 12 heurísticas, todas com a utilização de algoritmos evolutivos. Como critério multiobjetivo, os autores utilizam o ECA e o MCA, como descritos na Seção 2.3.4. Por sua vez, o critério de escolha da heurística a ser utilizada é definido com base em pesos. Inicialmente, todas as 12 heurísticas possuem o mesmo peso. A cada iteração uma heurística é utilizada. Após, o valor do peso da heurística será aumentado se houve melhora na solução, ou decrescido caso tenha ocorrido piora na solução.

Segundo os autores, o MHypEA produz soluções com melhor qualidade em um tempo 20 vezes menor quando comparado ao algoritmo genético multiobjetivo proposto por Praditwong et al. [25].

2.3.6 Algoritmos Híbridos

Algoritmos híbridos utilizam mais de uma técnica durante o processo de busca. Um exemplo dessa aplicação é o trabalho de Semaan e Ochi [27], onde é apresentado um algoritmo evolutivo híbrido. Neste trabalho foi utilizada uma heurística genética do tipo GNE, que trabalha em conjunto com uma busca local do tipo subida de encosta e com uma técnica chamada reconexão de caminhos (em inglês, *Path Relinking*) [8].

Na reconexão de caminhos, duas soluções são comparadas (em geral, a melhor solução já visitada e a solução corrente). O objetivo é transformar uma das soluções na outra por meio da geração de soluções intermediárias. Eventualmente uma das soluções intermediárias poderá ser melhor que as soluções utilizadas na entrada [27].

Os autores utilizam três variáveis para controlar quais componentes estarão ativados ou não durante o processo de busca. Uma variável controla a busca local, outra a reconexão de caminhos e outra o particionamento da população (técnica para acelerar a convergência da população). São executados oito testes (todas as combinações possíveis dos três métodos) com configurações distintas. Os melhores resultados para todos os MDGs analisados foram encontrados na configuração que utilizou os três métodos propostos. Embora a utilização das três técnicas em conjunto guie a busca a uma melhor solução, o tempo

de processamento foi o maior. Por outro lado, a configuração utilizando apenas o método de subida de encosta alcançou resultados próximos, com o tempo de processamento menor [27].

2.3.7 Busca Local Iterada

A meta-heurística Busca Local Iterada (em inglês, *Iterated Local Search* - ILS) [17] consiste em aplicar uma busca local sobre uma solução inicial. Uma vez que um ótimo local é atingido, um método de perturbação é aplicado sobre a solução ótima local. Assim, é gerada uma nova solução e esta passa a ser a solução inicial da próxima iteração da busca local.

Pinto [22] utiliza a busca ILS para o problema CMS. Como geração de solução inicial o autor propõe dois métodos: criar uma solução de forma aleatória e usar um algoritmo aglomerativo guloso, semelhante ao descrito na Seção 3.2.1. Para a busca local é utilizado um método de subida de encosta. Por sua vez, são criados cinco critérios de perturbação. Todos consistem em trocar módulos de *cluster*, utilizando uma estratégia fixa e com componente aleatório. Outro fator importante a ser observado é que sempre que ocorrem 5 perturbações seguidas sem melhoria no valor da função objetivo, a busca é reiniciada com uma solução aleatória.

O trabalho também apresenta um método de subida de encosta e dois algoritmos genéticos, um utilizando a codificação GGA e outro utilizando GNE. Os resultados obtidos pelo método ILS são comparados com os resultados obtidos pelos genéticos e também com o método de subida de encosta com múltiplos reinícios. Os resultados obtidos pelo ILS mostram que o método supera o algoritmo de subida de encosta com múltiplos reinícios com um tamanho de efeito médio de 87%, enquanto que, na comparação com as abordagens genéticas o ILS vence com efeito médio de 57%. Analisando os tempos de processamento, o ILS consegue tempos dentro da mesma ordem de magnitude que o algoritmo de subida de encosta. Por sua vez, quando comparado aos genéticos, o ILS possui tempos de processamento muito menores, chegando a ser 1.000 vezes mais rápido em um dos MDGs estudados [22].

2.4 Considerações finais

O problema CMS é um problema atual, com diversas publicações nos últimos cinco anos. Diferentes métodos já foram utilizados para o problema. O método de subida

de encosta com reinícios aleatórios obtêm bons resultados. Estudos baseados em metaheurísticas têm tentado superar os valores obtidos pelo método de subida de encosta. Por exemplo, no trabalho de Doval et al. [6] o algoritmo genético é utilizado, porém não supera o método subida de encosta com reinícios aleatórios. Contudo, outro trabalho mais recente [25] mostra resultados favoráveis para heurísticas genéticas. Além dos algoritmos genéticos, recentemente foi utilizada uma hiper-heurística [14] e um ILS [22] para o problema CMS, ambas apresentando bons resultados. Uma formulação matemática [12] também já foi utilizada. Essa formulação, conseguiu alcançar o resultado ótimo e comprová-lo em alguns MDGs. Outras técnicas, como redução do MDG [12], identificação e construção de blocos [18] e utilização de algoritmos híbridos [27] também já foram aplicadas ao problema.

Além da escolha do método de busca, outro fator importante é a escolha da função objetivo. A maioria dos trabalhos encontrados utiliza o MQ como critério de avaliação. Existem outros trabalhos que utilizam o EVM ou até mesmo métodos multiobjetivo. Embora o MQ seja mais utilizado, estudos comparativos feitos por Harman et al. [10] e Barros [3] mostram que a função objetivo EVM é mais robusta para softwares reais.

O próximo capítulo é dedicado à apresentação do algoritmo proposto, baseado na metaheurística de Busca em Vizinhança Grande para tratar o problema CMS.

3. Busca em Vizinhança Grande Aplicada ao Problema de Clusterização de Módulos de Software

Este capítulo está dividido em duas partes. A primeira parte introduz conceitos básicos relacionados à meta-heurística Busca em Vizinhança Grande [28] e a segunda apresenta a proposta de solução desta dissertação que é uma heurística baseada em Busca em Vizinhança Grande aplicada ao Problema de Clusterização de Módulos de Software (CMS).

3.1 A meta-heurística Busca em Vizinhança Grande

Nesta seção, inicialmente apresentam-se os conceitos de busca em vizinhança e busca em vizinhança em larga-escala, para então descrever a meta-heurística Busca em Vizinhança Grande, conforme Pisinger e Ropke [23].

3.1.1 Busca em vizinhança

Busca em vizinhança é um importante conceito quando desenvolvendo métodos heurísticos para resolver problemas de otimização combinatória. De acordo com Pisinger e Ropke [23], a definição formal de busca em vizinhança é: Seja um problema de otimização combinatória em que X é o conjunto finito e extremamente grande de soluções possíveis e, $c : X \rightarrow \mathbb{R}$ é uma função que mapeia uma solução x para o seu custo, assume-se que o problema de otimização combinatória é um problema de minimização, isto é, deseja-se encontrar uma solução x^* tal que $c(x^*) \leq c(x) \forall x \in X$.

Define-se a vizinhança de uma solução $x \in X$ como $V(x) \subseteq X$. Isto é, V é uma função que mapeia uma solução para um conjunto de soluções. Uma solução x é chamada de um ótimo local, respeitando a vizinhança V, se $c(x) \leq c(x') \forall x' \in V(x)$. O tamanho de uma vizinhança para um problema é definido por $\mathtt{maximo}\{|N(x)| : x \in X\}$. Dessa forma, é possível definir um algoritmo de busca em vizinhança. O algoritmo recebe uma solu-

ção inicial x como entrada. Então, o algoritmo verifica soluções $x' \in V(x)$. Caso exista alguma solução $c(x') < c(x)$, a solução corrente x será atualizada para $x = x'$. Sequencialmente, a vizinhança da nova solução x será verificada. O algoritmo para quando um ótimo local for alcançado.

3.1.2 Busca em vizinhança de larga-escala

Segundo Ahuja et al. [1], para que um algoritmo de busca em vizinhança seja considerado da classe de Algoritmos de Busca em Vizinhança de Larga Escala (em inglês, *Very Large Scale Neighborhood* - VLSN) é necessário que o crescimento da sua vizinhança seja exponencial em função do tamanho do problema, ou simplesmente que a vizinhança seja muito grande para ser totalmente percorrida.

Intuitivamente, uma busca em uma vizinhança muito grande deveria caminhar em direção a soluções com qualidade melhor do que uma busca em uma vizinhança pequena. Contudo, na prática, vizinhanças pequenas podem fornecer soluções similares ou até superiores quando embutidas em heurísticas baseadas em meta-heurísticas, simplesmente porque podem ser executadas mais rapidamente. Sendo assim, o sucesso de uma heurística do tipo VLSN não é garantido apenas pelo tamanho de sua vizinhança [23].

Um exemplo de vizinhança de crescimento exponencial pode ser encontrado nos métodos de profundidade variável (em inglês, *Variable-Depth Neighborhood* - VDN) que definem um conjunto de vizinhanças cada vez mais amplo de forma heurística. Seja um conjunto de vizinhanças $\{V_1, V_2, V_3, ..., V_k\}$, onde k é o total de vizinhanças criadas, tem-se que $|V_i| < |V_j|$ e $V_i \in V_j$ se $i < j$. Um exemplo desse tipo de vizinhança é exibido na Figura 3.1. A principal ideia dessa abordagem é a de se iniciar o algoritmo com uma vizinhança pequena para o algoritmo ser rápido e aumentar progressivamente a vizinhança na medida em que a busca fica estagnada, tornando o algoritmo mais lento, porém permitindo que sejam encontradas novas soluções vizinhas.

3.1.3 Busca em vizinhança grande

A meta-heurística Busca em Vizinhança Grande (em inglês, *Large Neighbourhood Search* - LNS), primeiramente proposta por Shaw [28], pertence à classe de algoritmos do tipo VLSN. LNS é baseada em um processo de constante destruição (relaxamento) e reparação (re-otimização) da solução corrente de um procedimento iterativo. A cada iteração o processo de destruição utiliza uma solução válida e remove alguns componentes da solução, gerando uma solução temporária e inválida. Em seguida, o processo de reparação

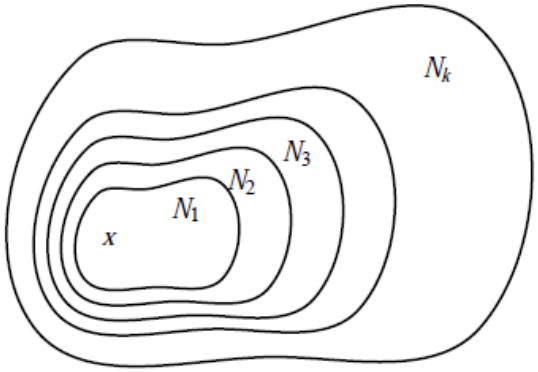


Figura 3.1: Exemplo de uma vizinhança de profundidade variável. A solução corrente é identificada por x [23].

utiliza a solução temporária e reinsere os elementos que foram removidos, gerando outra solução válida [23]. Caso a solução gerada pelo processo de reparação seja a melhor solução já encontrada, a solução é armazenada. Ao final de todas as iterações, a melhor solução encontrada é retornada.

A meta-heurística Busca em Vizinhança Grande possui uma estrutura simples e seu pseudo-código está demonstrado no Algoritmo 3.1. Para o funcionamento desse algoritmo são necessárias três variáveis de controle e quatro funções. As variáveis de controle são: x, usada para armazenar a solução corrente; x^t, usada para armazenar uma solução temporária obtida após a execução do processo de destruição e subsequente reparação e x^m, que armazena a melhor solução já visitada. Por sua vez, as funções são: $\texttt{destroi}(x)$, responsável pelo processo de destruição da solução x; $\texttt{repara}(x)$, responsável pelo processo de reparação (recebe uma solução x parcialmente destruída e a reconstrói); $\texttt{aceita}(x^t, x)$, responsável por verificar se a solução solução temporária x^t será aceita em relação à solução x e $\texttt{custo}(x)$, responsável por calcular o valor da função objetivo da solução x.

Algoritmo 3.1 Algoritmo Busca em Vizinhança Grande

```
1:  x ← uma solução viável
2:  x^m ← x
3:  repetir
4:      x^t ← repara(destroi(x))
5:      se aceita(x^t, x) então
6:          x ← x^t
7:      fim se
8:      se custo(x^t) > custo(x^m) então
9:          x^m = x^t
10:     fim se
11: até atingir algum critério de parada
12: retornar x^m
```

Na meta-heurística LNS a vizinhança é implicitamente definida pelos métodos de destruição e reparação. Isto é, a vizinhança de uma solução é definida por todas as demais

soluções alcançáveis após a execução do método de destruição e subsequente execução do método de reparação. Uma vez que o método de destruição pode destruir uma parte grande da solução, a vizinhança possui uma grande quantidade de vizinhos, o que explica o nome da heurística. Considere, por exemplo, um problema cuja solução é uma permutação. Considere também um fator de destruição de 15% e uma instância do problema com 100 elementos na solução. Neste caso, existem $C(100, 15) = 100!/(15! \times 85!) = 2.5 \times 10^{17}$ maneiras diferentes de destruir a solução e para cada solução destruída existem outras tantas maneiras distintas de construir a solução [23].

Desta forma, o tamanho da vizinhança é um fator importante nos procedimentos de busca baseados em LNS. Quanto menos elementos da solução forem removidos, menor será a vizinhança. Com uma vizinhança menor, a busca poderá convergir rapidamente para um mínimo local, eventualmente de baixa qualidade. Entretanto, ao remover mais elementos da solução, a busca terá uma liberdade maior para encontrar soluções melhores, porém o tempo de processamento será maior. Outra possibilidade é utilizar uma abordagem na qual o tamanho da vizinhança pode ser alterado ao longo da busca. É natural que o tamanho da vizinhança aumente uma vez que a busca esteja estagnada.

Segundo Pisinger e Ropke [23], navegar em um espaço de busca pequeno pode ser mais difícil do que navegar em um espaço de busca grande. Desta forma, acredita-se que uma busca baseada em LNS consiga navegar mais facilmente pelo espaço de busca, encontrando soluções de boa qualidade.

3.2 Busca em Vizinhança Grande aplicada ao problema CMS

Ao desenvolver heurísticas baseadas em meta-heurísticas é necessário fazer escolhas específicas para os componentes livres da meta-heurística. No caso da Busca em Vizinhança Grande, seus componentes livres são: solução inicial, método de destruição, método de reparação, critério de aceitação e critério de parada. Nesta seção, apresentam-se as escolhas específicas dos componentes livres da Busca em Vizinhança Grande para o problema CMS.

3.2.1 Métodos de construção de solução inicial

Todo processo de busca precisa de uma solução inicial. Um algoritmo construtivo para o problema CMS usado para gerar uma solução inicial recebe como entrada um conjunto $N = \{1, 2, \ldots, n\}$ de módulos e retorna uma solução viável, isto é, um conjunto de *clusters*

ou agrupamentos. O número de *clusters* da solução não é um dado do problema e pode variar entre 1 e n. A seguir, descrevem-se dois algoritmos construtivos utilizados pelo procedimento de busca proposto nesta dissertação. Para ambos, assume-se que os *clusters* estão indexados $1, 2, \ldots$ na ordem em que são criados, isto é, em que recebem o primeiro módulo.

- **Algoritmo Construtivo Aleatório (CA)**: cria uma solução de forma aleatória, sorteando para cada módulo um número entre 1 e n, sendo n o número total de módulos. O número sorteado será o identificador do *cluster* onde o módulo será adicionado.

- **Algoritmo Construtivo Aglomerativo MQ (CAMQ)**: é um processo iterativo que se inicia com uma solução em que cada módulo está em um *cluster* diferente. O processo consiste em juntar dois *clusters* a cada iteração até que exista apenas um *cluster*. A cada iteração todas as possíveis uniões são analisadas e a melhor união é escolhida, de forma a maximizar o MQ. A primeira solução tem sempre um valor de MQ $= 0$, pois não existem dependências internas em nenhum dos *clusters*. A última solução possui um valor de MQ $= 1$ pois há apenas um *cluster* que não possui dependência externa. A melhor solução encontrada durante o processo é retornada. O Algoritmo 3.2, extraído de Hansen e Jaumard [9] exibe um exemplo de um algoritmo aglomerativo onde N é o total de módulos, C_i é o i-simo cluster, P_N é uma partição com N *clusters* e O_j é o conjunto dos módulos do j-simo *cluster*.

Algoritmo 3.2 Algoritmo de clusterização hierárquica aglomerativa

1: ▷ Inicialização:
2: $P_N \leftarrow \{C_1, C_2, ..., C_N\}$
3: $C_j \leftarrow \{O_j\}\ j = 1, 2, ..., N$
4: $k \leftarrow 1$
5: ▷ Iteração:
6: **enquanto** $N - k > 1$ **faça**
7: Selecionar $C_i, C_j \in P_{N-k+1}$ seguindo um critério local
8: $C_{N+k} \leftarrow C_i \cup C_j$
9: $P_{N-k} \leftarrow (P_{N-k+1} \cup \{C_{N+k}\}) \setminus \{C_i, Cj\}$
10: $k \leftarrow k + 1$
11: **fim enquanto**

3.2.2 Métodos de destruição

Métodos de destruição removem elementos de uma dada solução, transformando uma solução viável em uma solução parcial (inviável). Tais métodos contêm componentes estocásticos para permitir que elementos distintos sejam destruídos a cada chamada do método. O percentual de elementos que serão removidos da solução é controlado pelo grau

de destruição. Shaw [28] propõe um aumento gradual no grau de destruição enquanto Pisinger e Ropke [23] propõem escolher o grau de destruição aleatoriamente a cada iteração dentro de um conjunto de valores possíveis, dependentes do tamanho da instância.

Caso o grau de destruição seja pequeno, a busca ficará rapidamente presa em uma região do espaço de busca. Por outro lado, se o grau de destruição for muito grande, o algoritmo de reparação ficará sobrecarregado e acabará gerando diversas vezes a mesma solução [23]. O importante é que o grau de destruição permita que todo o espaço de busca seja percorrido ou, pelo menos, áreas onde a melhor solução possa estar. Entretanto, destruir a solução toda a cada iteração não é interessante.

A seguir apresentam-se três métodos de destruição para o problema CMS. Em todos eles o elemento da solução a ser removido é o módulo e o grau de destruição é dado pelo parâmetro `grau_d`. Este parâmetro configura o percentual da solução que será destruído pelo método. O valor mínimo para o parâmetro é 0, desta forma nenhum módulo é removido da solução. O valor máximo para o parâmetro é 1, onde ocorre a remoção de todos os módulos da solução.

- **Método Destrutivo Aleatório (DA)**: remove `grau_d` módulos de uma solução de forma aleatória. A cada iteração um módulo é sorteado e removido da solução até que `grau_d` módulos sejam removidos. Considerando que no máximo n módulos podem ser removidos, e que a cada iteração um módulo é removido, a complexidade deste método é $O(n)$.

- **Método Destrutivo Diferença para a Melhor Solução (DDMS)**: enquanto a quantidade de módulos removidos for menor do que `grau_d`, remove módulos da solução comparando-a com a melhor solução já visitada. Cada solução é representada por um vetor onde o índice `i` representa o `i-simo` módulo e o valor associado ao índice é o *cluster* no qual o `i-simo` módulo está inserido. Esta comparação é feita verificando se os elementos associados aos índices do vetor possuem os mesmos valores. Primeiramente, são removidos os módulos que estão em *clusters* diferentes dos *clusters* da melhor solução. Se em algum momento não houver mais diferenças, a remoção será feita de forma aleatória. Considerando que a quantidade de módulos a ser removida é no máximo n, descobrir o identificador do *cluster* para um determinado módulo pode ser implementado em $O(1)$ (acesso ao valor do índice de um vetor), e que a cada iteração um módulo é removido então a complexidade deste método é $O(n)$.

- **Método Destrutivo Cluster Aleatório (DCA)**: enquanto a quantidade de módulos

removidos for menor do que `grau_d`, sorteia-se um *cluster* e então remove módulos de forma aleatória apenas dentro do *cluster* sorteado. Quando um *cluster* tem o seu ultimo módulo removido, outro *cluster* é sorteado. Considerando que no máximo n módulos podem ser removidos, encontrar um módulo dentro de um *cluster* é $O(1)$ e que a cada iteração um módulo é removido, a complexidade deste método é $O(n)$.

3.2.3 Métodos de reparação

Métodos de reparação são responsáveis por transformar uma solução parcialmente destruída em uma solução viável. Para o problema CMS, basta que todos os módulos removidos sejam colocados novamente em algum *cluster*. Porém, para que a Busca em Vizinhança Grande seja eficaz e encontre boas soluções, é necessário que o método de reparação remonte a solução de forma a otimizá-la. Pisinger e Ropke [23] sugerem a utilização de um algoritmo guloso.

O método de reparação pode ser ótimo dentro de suas possibilidades, ou seja, encontrar a melhor solução possível dentre as alcançáveis, utilizando a solução parcial como entrada. Porém, como a vizinhança é muito grande, verificar todas as possibilidades é muito caro computacionalmente, então é comum o método de reparação utilizar alguma heurística para encontrar uma boa solução. Segundo Pisinger e Ropke [23], do ponto de vista de vista de diversificação, "um método de reparo exato pode não ser interessante", pois pode gerar muitas vezes a mesma solução, deixando a busca estagnada em uma determinada parte do espaço de busca.

Foram criados quatro métodos de reparação da solução, descritos a seguir. Em todos os casos, considere que a solução corrente tem m *clusters*.

- **Reparativo Guloso Melhor Melhora Aleatório (RGMMA)**: A cada iteração, escolhe, em uma ordem aleatória, um dos módulos removidos pelo algoritmo destrutivo. Verifica todas as possibilidades de inserção do módulo corrente, considerando também a abertura de um novo *cluster*. A inserção é feita de forma a manter o MQ o maior possível. Uma iteração deste método possui complexidade $O(m)$. O número de iterações depende da quantidade de módulos removidos. No pior caso, a complexidade do método é $O(mn)$.
- **Método Reparativo Guloso Pior Melhora Aleatório (RGPMA)**: A cada iteração, escolhe, em uma ordem aleatória, um dos módulos removidos pelo algoritmo destrutivo. Cada um dos módulos será testado em todos os *clusters* existentes na so-

lução parcial e também em um novo. A inserção será feita de forma a garantir a pior melhora, ou no caso de não haver possibilidade de se melhorar, a inserção ocorrerá de forma a piorar o menos possível a solução. Uma iteração deste método possui complexidade $O(m)$. O número de iterações depende da quantidade de módulos removidos. No pior caso, a complexidade do método é $O(mn)$.

– **Método Reparativo Primeira Melhora Aleatório (RPMA)**: A cada iteração, escolhe em uma ordem aleatória, um dos módulos removidos pelo algoritmo destrutivo. Para inserir o módulo escolhido, seleciona o primeiro *cluster* em que a inserção do módulo melhora o valor da função objetivo. Caso não haja inserção que melhore o resultado, a inserção ocorrerá de forma a piorar o menos possível a qualidade da solução. Uma iteração deste método possui complexidade $O(m)$. O número de iterações depende da quantidade de módulos removidos. No pior caso, a complexidade do método é $O(mn)$.

– **Método Reparativo Guloso Melhor Melhora (RGMM)**: A cada iteração, verifica todas as possibilidades de inserir os módulos removidos em todos os *clusters*, sempre considerando a abertura de um novo *cluster*. A inserção que gerar o melhor resultado entre todas as demais será escolhida. Uma iteração deste método possui complexidade $O(mn)$. O número de iterações depende da quantidade de módulos removidos. No pior caso, a complexidade do método é $O(mn^2)$.

3.2.4 Critério de aceitação

Como pode ser observado no Algoritmo 3.1, a solução x^t gerada na linha 4 após os processos de destruição e reparação passa por um critério de aceitação, na linha 5. Escolheu-se a utilização de um critério de aceitação padrão, em que apenas soluções com valores de MQ maiores que a melhor solução já visitada são aceitas.

3.2.5 Critério de parada

Os algoritmos desenvolvidos usam dois critérios de parada clássicos: (i) aquele que pára a busca após um determinado número de iterações, controlado pelo parâmetro `max_itereções` e (ii) aquele que pára a busca após um determinado número de iterações sem melhoria no valor da melhor solução conhecida, controlado pelo parâmetro `max_itereções_sem_sucesso`.

3.3 Considerações finais

Este capítulo apresentou a meta-heurística Busca em Vizinhança Grande como proposta de solução para o problema CMS. Além da descrição da meta-heurística, foram detalhados as escolhas específicas para a implementação da heurística utilizada no trabalho. Cada um dos parâmetros de configuração da busca também foi exposto, assim como os algoritmos envolvidos nos processos de construção da solução inicial, destruição e reparação da solução a cada iteração.

O próximo capítulo descreverá os experimentos computacionais efetuados para avaliar a proposta. Os dados obtidos são, então, comparados com os dados da literatura.

4. Experimentos Computacionais

Apresentam-se, neste capítulo, os experimentos computacionais realizados com o procedimento de busca baseado na meta-heurística Busca em Vizinhança Grande, proposto para resolver heuristicamente o CMS. Inicialmente, com o objetivo de fazer as escolhas ideais para os componentes do LNS e para os valores dos parâmetros utilizados pela busca, foram realizados dois estudos experimentais descritos nas Seções 4.3 e 4.4. Na Seção 4.4.2 exibe-se os tempos computacionais do método proposto. Com o objetivo de avaliar a eficiência e eficácia do método proposto, foram realizados experimentos computacionais descritos nas Seções 4.5 e 4.6. A Seção 4.5 exibe um estudo experimental comparando o desempenho do método LNS com o método ILS proposto por Pinto [22]. A Seção 4.6 exibe um outro estudo experimental, desta vez comparando os resultados do método LNS com os melhores resultados encontrados na literatura.

Todos os experimentos foram realizados em um computador com processador Intel Core i7-2600 CPU 3.40 GHz, com memória 4GB DDR3 e dedicação exclusiva. Os algoritmos implementados nesta dissertação foram codificados na linguagem Java 7 e compilados utilizando-se a JDK 1.8.0_20-b26.

4.1 Instâncias de teste

Para o propósito dos experimentos foram escolhidas instâncias previamente utilizadas em outros trabalhos [3, 12, 14, 20, 21, 22, 24, 25]. As instâncias obtidas de Pinto [22] estavam no formato *.odem*, um arquivo XML que contém informações sobre a clusterização do sistema, a descrição dos módulos e seus relacionamentos. As instâncias obtidas de Mitchell [21] estavam em um formato mais simples, um arquivo-texto contendo em cada linha as informações sobre uma dependência. Por questões de simplicidade, optou-se por converter as instâncias que estavam no formato *.odem* para o formato texto.

A estrutura do arquivo-texto que representa as instâncias possui dois identificadores em cada linha e, em algumas instâncias, um número associado a ela. Cada linha representa uma dependência e os identificadores representam o nome dos módulos interdependentes. O peso da dependência é representado pelo número existente na mesma linha. Quando não existe um número associado, a dependência é tratada como tendo peso 1. Algumas instâncias não possuem o número com o peso da dependência em cada linha, porém, possuem a mesma dependência repetida em linhas diferentes. Nesse caso, o peso considerado para o experimento foi igual a quantidade de repetições de cada dependência. As informações dos nomes dos módulos, suas dependências e os seus respectivos pesos, são informações suficientes para a montagem de um MDG para a instância.

Foram reunidas 124 instâncias para estudo. Estas instâncias foram classificadas em categorias para facilitar a análise e as conclusões. A seguir, apresenta-se o método utilizado para classificar estas 124 instâncias. Considere o método de clusterização *k-means* que recebe como entrada um conjunto de elementos, associado a cada elemento um valor, e tem como objetivo particionar os elementos do conjunto em k grupos de forma que o valor de cada elemento do grupo seja o mais próximo possível da média da soma dos valores de todos os elementos do seu grupo. Seja a seguinte entrada para o procedimento *k-means*: o valor de k fixado em 5 e o conjunto de 124 instâncias, tendo cada uma delas a quantidade de módulos como seu valor associado. O *k-means* classificou as instâncias em cinco *clusters* e a quantidade de instâncias em cada *cluster* é : 64, 29, 18, 9 e 4. Por conterem poucos elementos, os últimos dois *clusters* foram unidos. A Tabela 4.1 exibe o nome das categorias, a quantidade mínima e máxima de módulos em cada categoria e quantas instâncias são trabalhadas por categoria. A maior instância avaliada possui 1.161 módulos e mais de 5.700 dependências.

Tabela 4.1: Instâncias por quantidade de módulos.

Categoria	Número de Módulos	Número de instâncias
Pequena (P)	Entre 2 e 74	64
Média (M)	Entre 79 e 182	29
Grande (G)	Entre 190 e 377	18
Muito Grande (MG)	Entre 413 e 1161	13

Uma listagem de todas as instâncias estudadas é exibida na Tabela 4.2. Além do nome (identificador) da instância, a tabela contém a categoria e a quantidade de módulos e dependências de cada instância. São exibidos os valores originais de cada instância e os valores obtidos após a execução do pré-processamento de redução, descrito na Seção 2.2. Algumas instâncias também estão marcadas com um asterisco após o nome. Isto significa

que foram utilizadas no primeiro e segundo estudos dos componentes do LNS (Seções 4.3 e 4.4), estudos executados para escolher a melhor configuração da busca. Escolheu-se um total de 18 instâncias para estes dois estudos. Este subgrupo possui instâncias das categorias P, M e G, sendo sete pertencentes à categoria P, seis pertencentes à categoria M e cinco pertencentes à categoria G.

Tabela 4.2: Conjunto de 124 instâncias com informações sobre a categoria, módulos e dependências antes e após o pré-processamento de redução de instância. Instâncias com (*) foram utilizadas nos dois primeiros estudos experimentais.

		Antes da Redução		Pós Redução	
Instância	**Categoria**	**Módulos**	**Dependências**	**Módulos**	**Dependências**
squid	P	2	2	1	1
small	P	6	5	3	5
compiler	P	13	32	13	32
random	P	13	32	12	23
regexp	P	14	20	9	18
jstl	P	15	20	14	15
lab4	P	15	18	10	14
netkit-ping	P	15	15	1	1
nss_ldap	P	15	16	3	4
nos	P	16	52	15	50
lslayout	P	17	43	17	43
boxer	P	18	29	12	29
netkit-tftpd	P	18	23	6	11
sharutils	P	19	36	14	30
mtunis *	P	20	57	20	57
spdb	P	21	33	7	8
xtell	P	22	57	14	44
bunch	P	23	62	15	45
Ispell *	P	24	103	23	97
netkit-inetd	P	24	25	4	6
nanoxml *	P	25	64	23	62
ciald	P	26	64	22	62
jodamoney	P	26	102	26	85
Modulizer	P	26	66	18	57
bootp	P	27	75	19	55
jxlsreader	P	27	73	25	73
sysklogd-1	P	28	74	22	65

Continua na próxima página

Continuação da página anterior

		Antes da Redução		Pós Redução	
Instância	**Categoria**	**Módulos**	**Dependências**	**Módulos**	**Dependências**
telnetd	P	28	81	19	62
crond	P	29	112	25	90
netkit-ftp	P	29	95	24	76
Rcs *	P	29	163	28	155
seemp	P	30	61	21	43
dhcpd-2	P	31	122	26	104
cyrus-sasl	P	32	100	21	77
tcsh	P	32	105	23	86
micq	P	33	156	26	116
apache_zip *	P	36	86	30	74
star	P	36	89	36	89
bison *	P	37	179	36	167
cia	P	38	636	34	167
stunnel	P	38	97	25	78
minicom	P	40	257	35	198
mailx	P	41	331	38	244
dot	P	42	255	40	248
screen	P	42	292	35	208
slang	P	45	242	42	184
slrn	P	45	323	37	231
net-tools	P	48	183	44	157
graph10up49	P	49	1650	49	1054
wu-ftpd-1	P	50	230	44	187
joe	P	51	540	44	318
hw	P	53	51	15	28
imapd-1	P	53	298	40	211
wu-ftpd-3	P	54	278	48	222
udt-java	P	56	227	54	210
javaocr	P	58	155	43	126
dhcpd-1	P	59	571	55	398
icecast	P	60	650	51	419
pfcda_base	P	60	197	54	168
servletapi	P	61	131	47	122
php	P	62	191	39	148
bunch2	P	65	151	42	111
forms	P	68	270	64	224

Continua na próxima página

Continuação da página anterior

		Antes da Redução		Pós Redução	
Instância	**Categoria**	**Módulos**	**Dependências**	**Módulos**	**Dependências**
jscatterplot *	P	74	232	68	171
jxlscore	M	79	330	67	305
elm-2	M	81	683	75	541
jfluid	M	81	315	75	276
grappa *	M	86	295	76	249
elm-1	M	88	941	81	736
gnupg	M	88	601	74	420
inn	M	90	624	80	485
bash	M	92	901	86	662
jpassword	M	96	361	87	332
bitchx	M	97	1653	92	1075
junit *	M	99	276	61	193
xntp	M	111	729	95	534
acqCIGNA	M	114	188	75	157
bunch_2	M	116	364	98	355
exim	M	118	1255	105	891
xmldom	M	118	209	67	159
cia++	M	124	369	63	309
tinytim *	M	129	564	122	499
mod_ssl	M	135	1095	123	752
jkaryoscope	M	136	460	127	345
ncurses	M	138	682	120	487
gae_plugin_core *	M	139	375	87	259
lynx	M	148	1745	100	1211
javacc	M	153	722	145	663
lucent	M	153	103	66	74
javaGeom	M	171	1445	160	1339
incl *	M	174	360	122	329
jdendogram *	M	177	583	148	447
xmlapi	M	182	413	125	358
jmetal	G	190	1137	178	1123
graph10up193	G	193	9190	193	7552
dom4j	G	195	930	181	876
nmh	G	198	3262	190	2473
pdf_renderer *	G	199	629	172	497
jung_graph_model	G	207	603	187	532

Continua na próxima página

Continuação da página anterior

		Antes da Redução		Pós Redução	
Instância	**Categoria**	**Módulos**	**Dependências**	**Módulos**	**Dependências**
jung_visualization *	G	208	919	192	837
jconsole	G	220	859	187	663
pfcda_swing *	G	248	885	237	801
jml-1.0b4 *	G	267	1745	262	1671
jpassword2	G	269	1348	248	1171
notelab-full *	G	293	1349	273	1233
poormans CMS	G	301	1118	253	1009
log4j	G	305	1078	258	944
jtreeview	G	320	1057	268	830
bunchall	G	324	1339	243	1250
jace	G	338	1524	271	1209
javaws	G	377	1403	293	1107
swing	MG	413	1513	377	1478
lwjgl-2.8.4	MG	453	1976	392	1790
res_cobol	MG	470	7163	461	5690
ping_libc	MG	481	2854	395	1802
y_base	MG	556	2510	479	2166
krb5	MG	558	3793	508	2494
apache_ant_taskdef	MG	626	2421	538	2108
itextpdf	MG	650	3898	603	3555
apache_lucene_core	MG	738	3726	712	3330
eclipse_jgit	MG	909	5452	872	4904
linux	MG	919	13493	879	11059
apache_ant	MG	1085	5329	999	4881
ylayout	MG	1161	5770	1004	4990

As Figuras 4.1 a 4.4 exibem gráficos para cada uma das categorias de instância. Para cada instância são exibidos dois pontos, um com o número original de módulos e outro com o número de módulos após o processo de redução. A Tabela 4.3 exibe o percentual de redução do número de módulos e de dependências de cada categoria de instâncias e a média geral. A categoria de instâncias que alcança o maior percentual de redução é a categoria Pequena, tanto para a quantidade percentual de módulos quanto para a quantidade percentual de dependências. Em contrapartida, a categoria Grande é a que obteve menor percentual de redução.

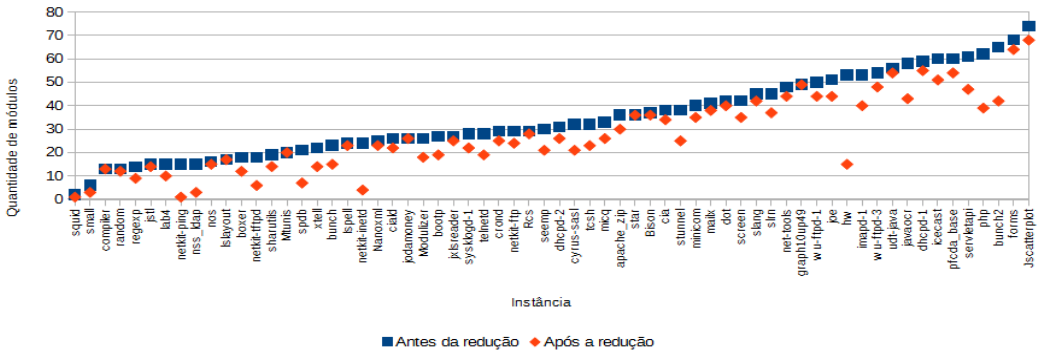


Figura 4.1: Número de módulos das instâncias da categoria Pequena.

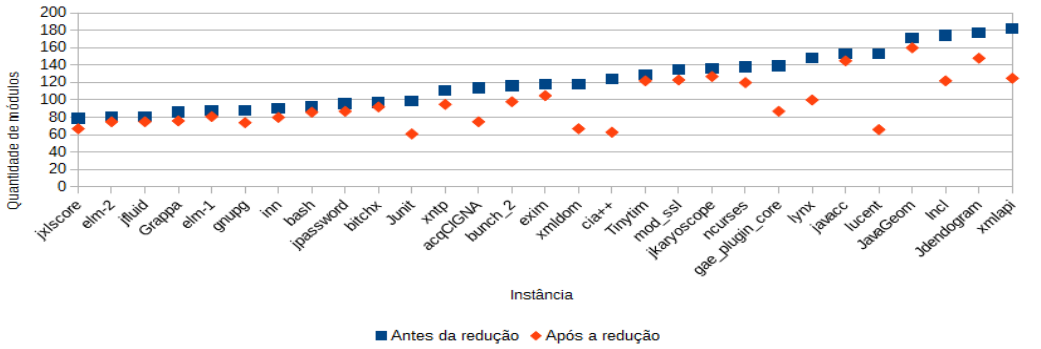


Figura 4.2: Número de módulos das instâncias da categoria Média.

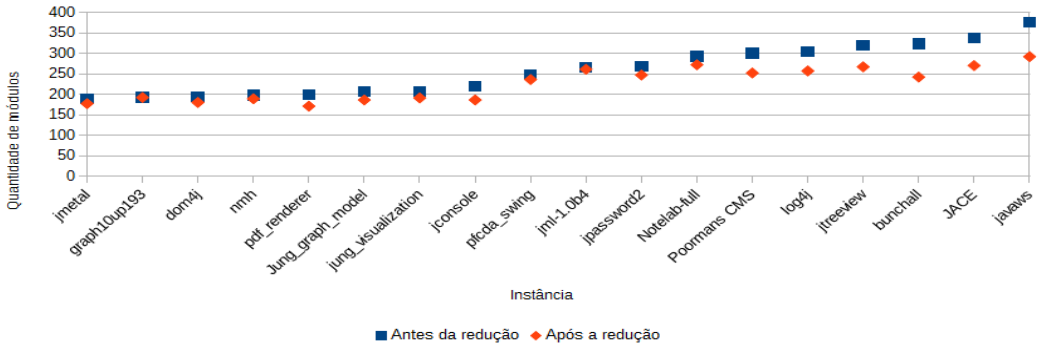


Figura 4.3: Número de módulos das instâncias da categoria Grande.

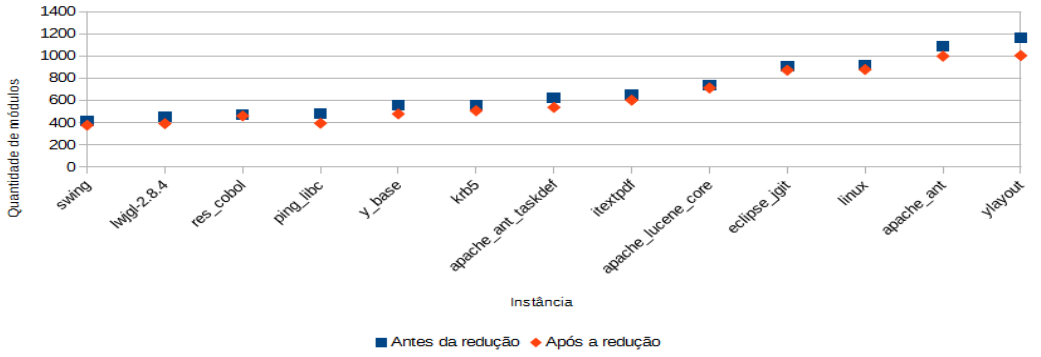


Figura 4.4: Número de módulos das instâncias da categoria Muito Grande.

Tabela 4.3: Percentual de redução do número de módulos e dependências das instâncias em cada categoria.

Categoria	Percentual de redução de módulos	Percentual de redução de dependências
Pequena	22,90%	23,38%
Média	19,09%	20,41%
Grande	11,06%	13,39%
Muito Grande	9,19%	15,34%
Média total	**18,85%**	**20,39%**

4.2 Questões de pesquisa

Os experimentos computacionais foram realizados com o objetivo de responder a três questões de pesquisa, apresentadas a seguir:

- **QP1 - Comparativo com a heurística ILS_CMS proposta por Pinto [22] : a heurística baseada em LNS produz soluções com maior MQ em um tempo menor que e a heurística ILS_CMS em um comparativo utilizando todas as instancias?**

  A heurística baseada em LNS e a heurística ILS_CMS foram executadas para todas as instâncias. As médias dos valores alcançados pela função objetivo MQ e a média dos tempos gastos em cada instância foram analisadas e comparadas.

- **QP2 - Eficácia como critério de avaliação: A heurística baseada em LNS produz soluções com maior valor de MQ que as soluções encontradas na literatura?**

  A média dos valores alcançados pela função objetivo MQ utilizando o método LNS para cada uma da instâncias analisadas foi comparada aos valores obtidos na literatura.

- **QP3: Eficiência como critério de avaliação: A heurística baseada em LNS produz soluções consumindo menos tempo de execução que as soluções encontradas na literatura?**

  O tempo de processamento, ou seja, o custo computacional do método LNS foi comparado ao custo computacional das melhores soluções encontradas na literatura para cada instância.

4.3 Definição dos componentes do LNS

O primeiro estudo experimental teve como objetivo avaliar diferentes escolhas para os componentes livres da meta-heurística LNS, apresentados na Seção 3.2, e diferentes valores para o parâmetro referente ao grau de destruição, chamado de `grau_d`. Os componentes avaliados do método LNS são: o método de construção de solução inicial (Seção 3.2.1), o método de reparação (Seção 3.2.3), o método de destruição de solução (Seção 3.2.2) e o parâmetro `grau_d` usado pelo método de destruição.

Chama-se de "parâmetro do experimento" um dos componentes do LNS ou um parâmetro dos algoritmos utilizados, e chama-se de "valor do parâmetro" as possíveis escolhas para os componentes e o conjunto de possíveis valores de um parâmetro. As escolhas possíveis para os componentes do método LNS são os algoritmos descritos na Seção 3.2 e as escolhas possíveis para o parâmetro `grau_d` são valores numéricos dentro do intervalo contínuo e infinito $[0, 1]$. Para a realização dos experimentos, discretizou-se o conjunto de valores possíveis para o parâmetro `grau_d`, como demonstrado na Seção 4.3.4.

O experimento foi realizado da seguinte forma: escolhe-se um parâmetro do experimento para ser estudado. Todos os demais parâmetros têm os seus valores sorteados (valores pertencentes aos conjuntos de valores disponíveis para cada um deles). Após todos os sorteios, a busca é executada uma vez para cada valor possível do parâmetro sendo estudado. Com isso, é possível avaliar para cada valor do parâmetro a qualidade da solução gerada pela busca. Para amenizar o efeito das demais escolhas, este procedimento é repetido 2.000 vezes, sempre utilizando valores sorteados para os demais parâmetros. Esse procedimento foi repetido para cada uma das 18 instâncias (marcadas com $*$ na Tabela 4.2) utilizadas no primeiro estudo dos componentes do LNS.

Após todas as execuções, os resultados obtidos foram comparados. A comparação foi feita observando a média dos MQs gerados e o tempo gasto em cada uma das execuções. Sempre que uma escolha para o valor do parâmetro gera um valor de MQ mais alto que as demais, esta opção ganha um ponto no comparativo (vitória); quando ocorre empate no valor de MQ, todas as escolhas empatadas ganham ponto; porém quando todos os valores do parâmetro geram o mesmo MQ e este MQ é exatamente igual ao de entrada da busca (não houve melhora), nenhum valor recebe ponto. Sendo assim, a pontuação mínima de um valor do parâmetro analisado é 0, caso perca em todos os comparativos, e a pontuação máxima é 36.000 (2.000 execuções vezes 18 instâncias), caso vença sempre.

Após a análise do experimento para um determinado parâmetro, os valores que obtiveram melhores resultados continuam sendo utilizados nos experimentos dos próximos

parâmetros, enquanto que os valores que não obtiveram bons resultados são descartados. Para o primeiro estudo decidiu-se fixar o critério de parada (parâmetro `max_iterações`) em 500 iterações.

As Tabelas 4.4 a 4.7 possuem estruturas semelhantes. A coluna "Pontos" exibe a quantidade de vezes que um valor alcançou resultados melhores ou iguais aos demais valores do parâmetro, a coluna "Tempo médio(s)" exibe a média do tempo de processamento da busca para cada valor e, por fim, a coluna "% pontos" é a quantidade percentual de pontos que um valor recebeu. Um asterisco ao lado do valor do parâmetro significa que o mesmo foi mantido para os experimentos seguintes. Valores sem asterisco foram descartados.

4.3.1 Estudo dos métodos de construção de solução inicial

O primeiro parâmetro do experimento analisado foi o método de construção de solução inicial. Este parâmetro é responsável pela escolha do algoritmo utilizado no processo de construção da solução inicial. Foram desenvolvidos dois algoritmos construtivos, chamados CAMQ e CA. A descrição destes algoritmos está na Seção 3.2.1. A Tabela 4.4 apresenta os dados obtidos no experimento. Utilizando o algoritmo CAMQ a busca conseguiu encontrar melhores soluções do que utilizando o algoritmo CA em 78% das vezes. Por esse motivo, decidiu-se por manter o CAMQ fixo nos experimentos seguintes, enquanto o valor CA foi descartado.

Tabela 4.4: Comparativo entre os algoritmos de construção de solução inicial.

Valor	Pontos	Tempo médio(s)	% pontos
CAMQ *	7455	0,74	78,65%
CA	2024	0,58	21,35%

4.3.2 Estudo dos métodos de reparação da solução

O segundo parâmetro do experimento analisado foi o método de reparação da solução. Este parâmetro é responsável pela escolha do algoritmo utilizado no processo de reparação da solução, executado a cada iteração da busca. Foram desenvolvidos quatro algoritmos. São eles: RGMMA, RGPMA, RPMA e RGMM, descritos na Seção 3.2.3. A Tabela 4.5 apresenta os dados obtidos no experimento.

O algoritmo RGMM consegue cerca de 54% dos pontos e o algoritmo RGMMA obtém 40% dos pontos. Optou-se por manter os dois valores para o próximo experimento. O RGMM foi mantido por apresentar o maior número de vitórias e o RGMMA foi mantido por apresentar número de vitórias próximos do RGMM e também por apresentar um tempo

computacional cerca de 30 vezes menor.

Tabela 4.5: Comparativo entre os algoritmos de reparação da solução.

Valor	Pontos	Tempo médio(s)	% de pontos
RGMM *	15817	1,98	53,93%
RGMMA *	11853	0,07	40,41%
RPMA	1659	0,02	5,66%
RGPMA	0	0,06	0,00%

4.3.3 Estudo dos métodos de destruição da solução

O terceiro parâmetro do experimento analisado foi o método de destruição da solução. Este parâmetro é responsável pela escolha do algoritmo utilizado no processo de destruição da solução, que é executado a cada iteração da busca. Foram desenvolvidos três algoritmos. São eles: DA, DDMS e DCA, descritos na Seção 3.2.2. A Tabela 4.6 apresenta os dados obtidos no experimento. Os algoritmos DA e DDMS conseguem cerca de 50% dos pontos cada, com aproximadamente o mesmo tempo de processamento. O método DCA foi descartado por obter o número de vitórias muito inferior ao dos demais métodos. Os métodos DA e DDMS, por sua vez, obtiveram resultados muito próximos. Porém, por ser um algoritmo mais complexo, o DDMS foi descartado.

Tabela 4.6: Comparativo entre algoritmos de destruição da solução.

Valor	Pontos	Tempo médio(s)	% de vitoria
DA *	20283	1,77	50,10%
DDMS	19975	1,78	49,34%
DCA	226	1,30	0,56%

4.3.4 Estudo do grau de destruição

Por último, analisou-se o parâmetro grau de destruição. Este parâmetro é responsável pela quantidade de módulos que serão removidos da solução a cada iteração. Para funcionar independente do tamanho da instância, esse parâmetro foi estabelecido como um valor percentual do total de módulos existentes na instância. Sendo assim, o seu intervalo de dados é um conjunto infinito entre 0 e 1. Para que o parâmetro seja utilizado no experimento é necessário que o mesmo possua um conjunto discreto e finito de valores. Para criar este conjunto foram utilizados os mesmos valores utilizados no trabalho de Ropke et al. [26]. Assim, os valores variam de 0,05 até 0,5, com intervalo de 0,05. A Tabela 4.7 apresenta os dados obtidos no experimento.

Os valores 0,15 e 0,10 conseguem um número de vitórias muito próximos e, embora o valor 0,15 consiga mais pontos que o valor 0,10, seu tempo de processamento é mais que o dobro do tempo de processamento do 0,10. Por este motivo, o valor 0,10 foi mantido e os demais foram descartados.

Tabela 4.7: Comparativo entre os valores do parâmetro grau de destruição.

Valor	Pontos	Tempo médio	% de pontos
0,15	14646	0,44	14,23%
0,10 *	14163	0,21	13,76%
0,20	13470	0,75	13,09%
0,25	11914	1,15	11,58%
0,30	10772	1,63	10,47%
0,35	9892	2,21	9,61%
0,40	8530	2,85	8,29%
0,45	7449	3,50	7,24%
0,50	6297	4,17	6,12%
0,05	5787	0,06	5,62%

Observa-se, após a execução do primeiro estudo, que os métodos de construção de solução inicial e reparação da solução alcançam melhores resultados quando não possuem componentes aleatórios (CAMQ e RGMM). O método de destruição, por sua vez, é totalmente dependente de componentes aleatórios. Por fim, o grau de destruição obtêm os melhores resultados quando seu valor está entre 0,10 e 0,20, mostrando que estes valores permitem que a busca navegue em uma área promissora da região de busca. O grau de destruição 0,05 obteve o pior desempenho. Isto mostra que destruir pouco a solução não permite que a busca navegue bem pelo espaço de busca e encontre soluções de qualidade.

4.4 Definição das configurações do LNS

O primeiro estudo experimental investigou diversas escolhas para os componentes método de construção de solução inicial, método de reparação, método de destruição e para o parâmetro `grau_d`. O segundo estudo experimental tem como objetivo aprofundar esta análise, incluindo a investigação do componente critério de parada. Diferentemente do primeiro estudo, que avaliou os parâmetros do experimento de forma isolada, o segundo estudo considera configurações específicas do algoritmo, sendo que cada configuração é um conjunto de valores estabelecidos para os parâmetros configuráveis do experimento. Assim, este estudo avalia o efeito conjunto das escolhas para os componentes do experi-

mento.

Os resultados do primeiro experimento foram utilizados como guia para a escolha das configurações utilizadas. Sendo assim, cada parâmetro teve seu valor escolhido de acordo com seu respectivo desempenho (qualidade das soluções alcançadas e custo computacional) nos experimentos da Seção 4.3. A Tabela 4.8 exibe os valores escolhidos para cada parâmetro.

Tabela 4.8: Parâmetros do segundo experimento e seus respectivos valores.

Parâmetro do experimento	**Valores**
Método de construção de solução inicial	CAMQ
Método de reparação	RGMMA, RGMM
Método de destruição	DA
Grau de destruição	0,1

Como método inicial foi escolhido o algoritmo CAMQ por ter apresentado um número de vitórias significativamente maior que o seu concorrente CA.

Como método reparador foram escolhidos os algoritmos RGMM e RGMMA, pois ambos obtiveram pontuações próximas. O RGMM fez 54% dos pontos e o RGMMA obteve 40% dos pontos. Embora tenha obtido menos pontos, o algoritmo RGMMA possui um custo computacional consideravelmente menor, gerando um tempo total 30 vezes menor que o seu rival.

Para o método destruidor, o valor escolhido foi o DA. Embora os resultados obtidos pelo método DDMS sejam muito próximos aos resultados obtidos pelo DA, tanto em número de vitórias quanto em custo computacional, optou-se por manter o valor DA por ser um método de destruição mais simples.

Por sua vez, o valor escolhido para o grau de destruição foi 0,10. O valor escolhido não obteve o maior número de pontos, ficando com um percentual total de pontos 0,5% menor que o valor 0,15. Porém, o tempo de processamento do valor 0,15 é duas vezes maior que o 0,10. Por ser mais rápido e por conseguir praticamente o mesmo percentual de pontos, o valor 0,10 foi mantido.

4.4.1 Estudo das configurações do LNS

No estudo das configurações do LNS foram criados dois tipos de configurações. O primeiro tipo de configuração, chamado de FIXA, utiliza todos os parâmetros do experi-

mento fixos e a busca é interrompida quando o limite máximo de iterações é alcançado. O segundo tipo de configuração, chamado de MISTA, utiliza dois métodos reparadores: o RGMMA e o RGMM. Neste segundo tipo de configuração a busca é interrompida apenas após utilizar todos os métodos reparadores disponíveis e não conseguir melhorar a solução em nenhum deles. Por outro lado, se houver alguma melhoria, todos os métodos são revisitados. A quantidade máxima de iterações sem melhoria disponível para cada método é definida pelo parâmetro chamado `max_iterações_sem_sucesso`.

Duas variações do tipo de configuração MISTA também foram experimentadas. Estas configurações foram chamadas de MISTA SEM REINICIO e MISTA REINICIO 0,5. A configuração MISTA SEM REINICIO utiliza os métodos reparadores RGMMA e RGMM e, diferentemente da abordagem MISTA, não ocorre o ciclo entre os métodos reparadores, ou seja, ao chegar ao final do segundo método reparador, a busca é interrompida. A segunda configuração estudada, chamada de MISTA REINICIO 0,5, possui um comportamento muito parecido com a MISTA, no qual é permitido à busca retornar a métodos reparadores já utilizados. Porém, em vez de ter novamente o mesmo limite de iterações sem melhoria, a busca terá o valor do limite anterior multiplicado por 0,5. Este processo só é interrompido quando os dois métodos de reparação disponíveis (RGMMA e RGMM) são executados e nenhum deles consegue melhorar a solução existente.

Cada configuração utilizada é nomeada de acordo com o tipo de configuração utilizado e com a quantidade de iterações disponível para a busca. Este valor é definido pelo parâmetro `max_iterações` ou `max_iterações_sem_sucesso`. Para cada configuração a busca foi executada 100 vezes em cada uma das 18 instâncias utilizadas.

A Tabela 4.9 exibe o nome de quatro configurações do tipo FIXA, além do valor de cada um dos parâmetros do experimento. Por sua vez, a Tabela 4.10 exibe as mesmas informações para as configurações do tipo MISTA.

Tabela 4.9: Quatro configurações da busca LNS do tipo FIXA.

Parâmetro do experimento	**FIXA RA 500**	**FIXA RM 500**	**FIXA RA 1000**	**FIXA RM 1000**
Construção de solução inicial	CAMQ	CAMQ	CAMQ	CAMQ
Método de reparação	RGMMA	RGMM	RGMMA	RGMM
Método de destruição	DA	DA	DA	DA
Grau de destruição	0,1	0,1	0,1	0,1
max_iterações	500	500	1000	1000
max_itereções_sem_sucesso	-	-	-	-

Tabela 4.10: Quatro configurações da busca LNS do tipo MISTA.

Parâmetro do experimento	**MISTA 500**	**MISTA 1000**	**MISTA 1000 SEM REINICIO**	**MISTA 1000 COM REINICIO 0,5**
Construção de solução inicial	CAMQ	CAMQ	CAMQ	CAMQ
Método de reparação	RGMMA RGMM	RGMMA RGMM	RGMMA	RGMMA RGMM
Método de destruição	DA	DA	DA	DA
Grau de destruição	0,1	0,1	0,1	0,1
max_iterações	-	-	-	1000
max_itereções_sem_sucesso	500	1000	1000	1000

Tabela 4.11: Resultados obtidos com as configurações da busca LNS do tipo FIXA.

Instância	**FIXA RA 500**	**FIXA RM 500**	**FIXA RA 1000**	**FIXA RM 1000**
mtunis	2,2668	2,2710	2,2755	2,2753
ispell	2,3312	2,3318	2,3319	**2,3323**
nanoxml	3,8161	3,8155	3,8169	3,8159
rcs	2,2201	2,2222	2,2224	2,2248
apache_zip	5,7602	5,7644	5,7655	5,7662
bison	2,6762	2,6729	2,6815	2,6789
jscatterplot	10,7407	10,7399	**10,7419**	10,7417
grappa	12,7047	12,7052	12,7053	**12,7054**
junit	11,0900	11,0900	**11,0901**	11,0900
tinytim	12,5122	12,4956	12,5177	12,4963
gae_plugin_core	17,3267	17,2929	**17,3358**	17,2929
incl	13,6002	13,6057	13,6022	13,6088
jdendogram	26,0614	26,0608	26,0649	26,0629
pdf_renderer	22,3548	22,3570	22,3568	22,3566
jung_visualization	21,8151	21,8167	21,8288	21,8225
pfcda_swing	29,0290	29,0630	29,0582	29,0700
jml-1.0b4	17,5045	17,5145	17,5307	17,5294
notelab-full	29,5523	29,5711	29,6032	29,5905

A Tabela 4.11 e Tabela 4.12 exibem as médias dos valores de MQ obtidos com cada uma das configurações do tipo FIXA e MISTA em cada uma das instâncias. Os resultados marcados em negrito são os melhores resultados obtidos no experimento para cada instância. Comparando-se os resultados obtidos, percebe-se que a configuração MISTA 1000 alcança a melhor média de MQ em 13 das 18 instâncias. Por outro lado, as configurações do tipo FIXA conseguem poucas vitórias. Isto demonstra a superioridade das configurações do tipo MISTA sobre as configurações do tipo FIXA.

Tabela 4.12: Resultados obtidos com as configurações da busca LNS do tipo MISTA.

Instância	**MISTA 500**	**MISTA 1000**	**MISTA 1000 SEM REINICIO**	**MISTA 1000 COM REINICIO 0,5**
mtunis	2,2747	**2,2856**	2,2797	2,2782
ispell	2,3320	**2,3323**	**2,3323**	**2,3323**
nanoxml	3,8164	**3,8171**	**3,8171**	**3,8171**
rcs	2,2253	**2,2261**	**2,2261**	2,2259
apache_zip	5,7660	**5,7663**	**5,7663**	**5,7663**
bison	2,6807	**2,6828**	2,6817	2,6815
jscatterplot	**10,7419**	10,7414	**10,7419**	**10,7419**
grappa	12,7053	**12,7054**	**12,7054**	**12,7054**
junit	11,0900	11,0900	11,0900	11,0900
tinytim	12,5157	**12,5200**	12,5171	12,5156
gae_plugin_core	17,3279	**17,3358**	17,3344	17,3352
incl	13,6073	**13,6122**	13,6113	13,6113
jdendogram	26,0649	26,0666	**26,0671**	26,0667
pdf_renderer	22,3573	22,3572	**22,3573**	**22,3573**
jung_visualization	21,8282	21,8283	21,8310	**21,8316**
pfcda_swing	29,0707	**29,0720**	29,0719	29,0710
jml-1.0b4	17,5443	**17,5554**	17,5504	17,5495
notelab-full	29,6267	**29,6458**	29,6433	29,6407

Tabela 4.13: Tempo de processamento em segundos das configurações da busca LNS do tipo FIXA.

Instância	FIXA RA 500	FIXA RM 500	FIXA RA 1000	FIXA RM 1000
mtunis	0,00	0,00	0,00	0,00
ispell	0,00	0,00	0,01	0,00
nanoxml	0,00	0,00	0,01	0,00
rcs	0,00	0,00	0,01	0,01
apache_zip	0,00	0,00	0,01	0,01
bison	0,01	0,01	0,01	0,01
jscatterplot	0,01	0,02	0,02	0,03
grappa	0,01	0,03	0,03	0,06
junit	0,01	0,02	0,02	0,04
tinytim	0,03	0,10	0,05	0,20
gae_plugin_core	0,02	0,04	0,03	0,07
incl	0,02	0,06	0,04	0,12
jdendogram	0,03	0,16	0,07	0,31
pdf_renderer	0,04	0,18	0,07	0,35
jung_visualization	0,06	0,42	0,12	0,83
pfcda_swing	0,07	0,64	0,15	1,28
jml-1.0b4	0,12	1,35	0,23	2,68
notelab-full	0,11	1,25	0,22	2,48
Tempo total	**54**	**429**	**107**	**849**

A Tabela 4.13 e Tabela 4.14 exibem o tempo médio de processamento gasto por cada uma das configurações em cada uma das instâncias. Ao final, é exibido o tempo total aproximado de processamento de cada uma das configurações (tempo de execução de 100 rodadas para as 18 instâncias). Observa-se que as configurações do tipo FIXA exigiram menor tempo computacional para serem executadas. A configuração MISTA 1000 foi a mais eficaz (alcançou maiores médias de MQ) e, ao mesmo tempo, foi a menos eficiente (exigiu maior tempo computacional).

Tabela 4.14: Tempo de processamento em segundos das configurações da busca LNS do tipo MISTA.

Instância	MISTA 500	MISTA 1000	MISTA 1000 SEM REINICIO	MISTA 1000 COM REINICIO 0,5
mtunis	0,00	0,01	0,01	0,01
ispell	0,01	0,01	0,01	0,01
nanoxml	0,01	0,01	0,01	0,01
rcs	0,01	0,02	0,02	0,02
apache_zip	0,01	0,02	0,02	0,02
bison	0,02	0,03	0,02	0,03
jscatterplot	0,03	0,06	0,05	0,06
grappa	0,06	0,10	0,10	0,10
junit	0,03	0,06	0,06	0,06
tinytim	0,26	0,50	0,31	0,40
gae_plugin_core	0,10	0,19	0,12	0,16
incl	0,11	0,22	0,20	0,21
jdendogram	0,30	0,58	0,49	0,54
pdf_renderer	0,32	0,61	0,58	0,59
jung_visualization	1,20	2,33	1,56	1,90
pfcda_swing	1,34	2,26	2,07	2,10
jml-1.0b4	4,58	8,74	5,83	7,11
notelab-full	4,22	7,57	5,50	6,52
Tempo total	**1.259**	**2.331**	**1.696**	**1.982**

A Tabela 4.15, em forma de matriz, exibe a comparação dos valores de MQ obtidos para cada par de configurações listadas na primeira linha e primeira coluna da matriz. A comparação foi feita calculando-se o somatório da diferença percentual entre as médias dos valores de MQ obtidos pelos métodos para cada instância. Cada célula da matriz representa o valor obtido pela comparação da configuração da linha correspondente com a coluna correspondente. Valores positivos indicam que a configuração da linha obteve melhores resultados enquanto que valores negativos indicam que a configuração da coluna obteve melhores resultados. Observa-se com base nos dados que a configuração MISTA 1000 foi a única que conseguiu resultados favoráveis em comparação com todas as outras configurações.

Tabela 4.15: Comparação feita par a par entre as configurações da busca LNS estudadas.

	FIXA RM 500	**FIXA RA 1000**	**FIXA RM 1000**	**MISTA 500**	**MISTA 1000**	**MISTA 1000 SEM REINICIO**	**MISTA 1000 COM REINICIO 0,5**
FIXA RA 500	-0,20%	-1,46%	-1,05%	-1,70%	-2,59%	-2,23%	-2,12%
FIXA RM 500	-	-1,26%	-0,85%	-1,50%	-2,39%	-2,03%	-1,92%
FIXA RA 1000	-	-	0,41%	-0,24%	-1,12%	-0,76%	-0,66%
FIXA RM 1000	-	-	-	-0,65%	-1,54%	-1,18%	-1,07%
MISTA 500	-	-	-	-	-0,88%	-0,52%	-0,42%
MISTA 1000	-	-		-	-	0,35%	0,46%
MISTA 1000 SEM REINICIO	-	-	-	-	-	-	0,11%

A configuração MISTA 1000 obteve a média mais alta de MQ para a maioria das instâncias avaliadas (em 13 de 18 instâncias). Também observou-se que, em uma comparação entre as médias de MQ obtidas para cada instância, esta configuração apresentou resultados favoráveis em relação a todas as outras configurações estudadas. Contudo, o melhor desempenho trouxe consigo o maior custo computacional. Este estudo privilegiou a qualidade das soluções e, por esse motivo, decidiu-se prosseguir com a configuração MISTA 1000.

4.4.2 Tempos de execução das soluções encontradas

Nesta seção exibe-se graficamente os tempos de processamento para a configuração MISTA 1000.

A Figura 4.5 exibe o tempo médio de processamento de cada instância. O eixo das abscissas está ordenado de acordo com o número de módulos em cada instância antes do pré-processamento de redução de grafos MDG. Por sua vez, a Figura 4.6 exibe o mesmo dado, porém o eixo das abscissas está ordenado de acordo com o número de módulos de cada instância após o processo de redução ser executado. Segundo Praditwong et al. [25] não existe um relacionamento óbvio entre o tamanho do problema e o esforço computacional. Tal fato pode ser observado comparando-se os gráficos. O crescimento do tempo de processamento não ocorre sempre que a instância aumenta de tamanho e tampouco é uniforme, mesmo após o pré processamento de redução. Após a execução do pré-processamento de redução das instâncias o crescimento do tempo é mais uniforme, porém, não é totalmente linear.

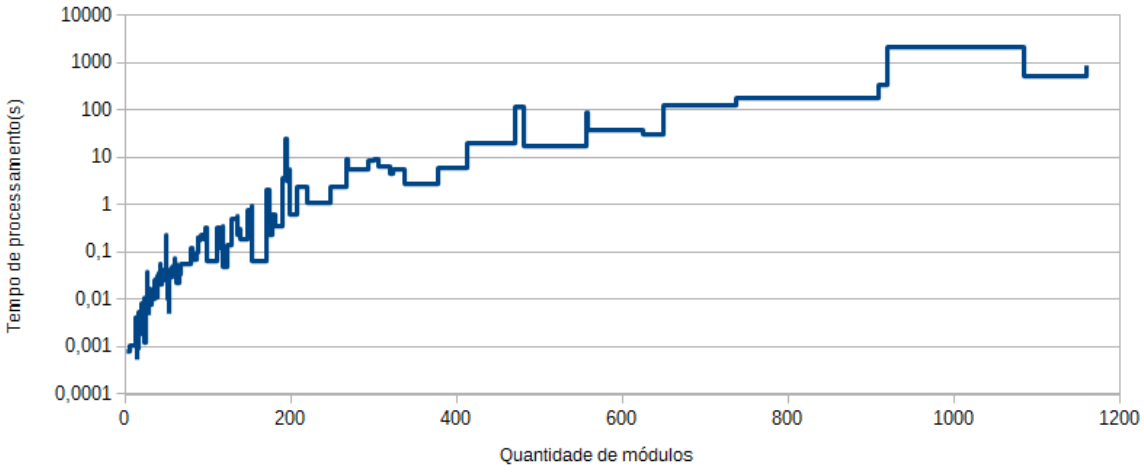


Figura 4.5: Tempo médio de processamento de cada instância em função do número de módulos antes da redução.

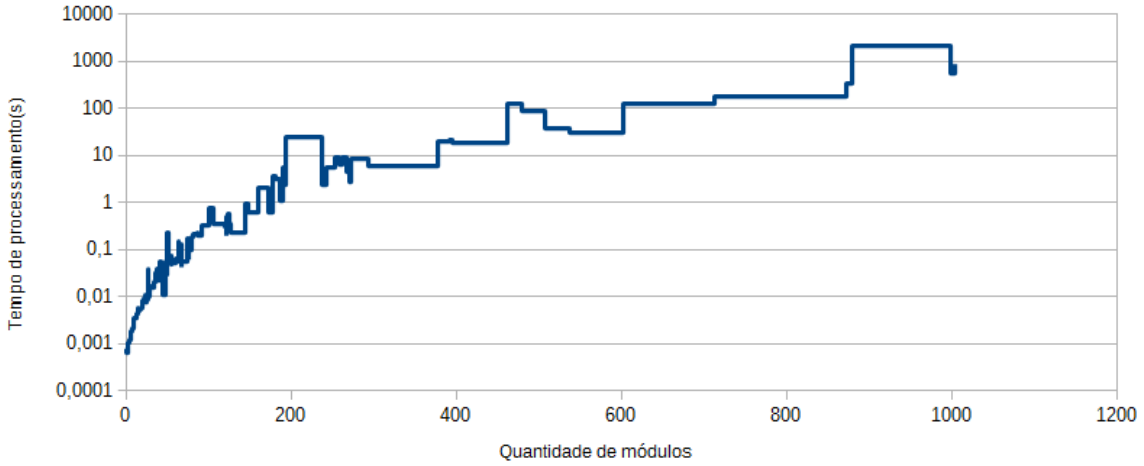


Figura 4.6: Tempo médio de processamento de cada instância em função do número de módulos depois da redução.

A configuração MISTA 1000 possui dois métodos reparadores. Ambos são executados, sendo que ao atingir 1000 iterações sem melhoria o método é trocado. Então, a busca executa no mínimo 2000 iterações, caso nenhuma melhoria seja feita. A Figura 4.7 exibe o número médio de iterações em cada instância. É possível perceber que a última iteração média nas instâncias com menos de 31 módulos fica sempre próxima a 2000 iterações, o que indica que a busca em vizinhança não conseguiu melhorar as soluções encontradas pelo método construtivo. Esta observação indica que, em trabalhos futuros, deve-se investigar uma variante da configuração MISTA 1000 que leve em conta o tamanho da instância para o critério de parada. Também é possível observar que o crescimento do

número de iterações em função do número de módulos original da instância não é linear.

Juntando-se as informações das Figuras 4.5 a 4.7, pode-se perceber que para as instâncias maiores a heurística LNS com a configuração MISTA 1000 consegue executar mais iterações. Para estas instâncias a busca foi capaz de encontrar melhores soluções ao longo das iterações. Contudo, o aumento no número de iterações traz consigo um aumento no tempo de processamento.

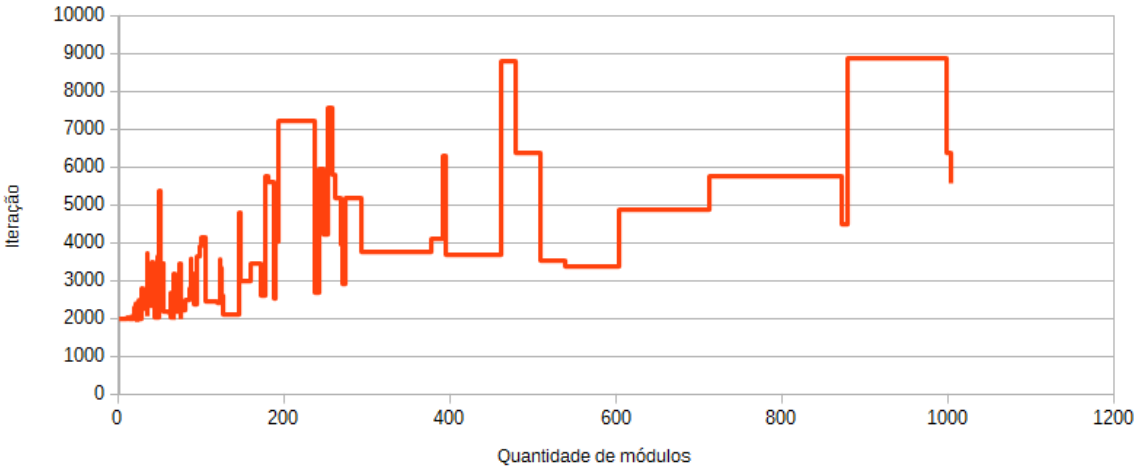


Figura 4.7: Última iteração média em cada instância.

Nas próximas seções, a configuração MISTA 1000 da heurística baseada na metaheurística Busca em Vizinhança Grande, proposta para resolver o problema CMS, será chamada de LNS_CMS.

4.5 Estudo experimental comparativo entre a heurística LNS_CMS e a heurística ILS_CMS

Com o intuito de responder a QP1, foi feito um estudo experimental comparando os resultados obtidos pela heurística LNS_CMS e os resultados obtidos pela melhor versão da heurística ILS_CMS (a versão ILS1) proposta por Pinto [22]. Foram realizadas 100 execuções independentes para cada método. Este estudo utilizou as 124 instâncias apresentadas na Tabela 4.2.

Os testes estatísticos foram aplicados com o objetivo de determinar se os resultados obtidos pelas heurísticas podem ser considerados significativamente distintos entre si e se os valores médios obtidos por uma heurística são superiores aos valores médios obtidos pela outra. Foi aplicado o teste de inferência estatística não paramétrico Wilcoxon-Mann-

Whitney [7]. Este teste compara a média de duas amostras independentes e não requer normalidade e homocedasticidade nas séries de valores comparadas. O nível de significância aplicado é de 95%. O resultado deste teste estatístico é chamado de *p-value*. Quanto menor o valor de *p-value*, maior é a significância do resultado. Isto é, as médias observadas para as amostras são distintas. Para níveis de significância de 95% é necessário que o *p-value* seja menor que 0,05.

Para se identificar a quantidade de vezes que uma heurística é melhor do que a outra, é possível utilizar uma medida de tamanho de efeito (em inglês, *effect-size* - ES). Esta comparação é feita par a par com base em cada um dos resultados obtidos no experimento. Para o cálculo do tamanho de efeito, foi utilizada a métrica não paramétrica $\hat{A}_{12}$ [31].

Utilizou-se a ferramenta estatística R na versão 3.1.0 para cálculo dos valores das médias, desvios-padrão, testes de Wilcoxon-Mann-Whitney e tamanho de efeito.

O código fonte da ILS_CMS está disponível em https://github.com/cmsp/ils.git. Originalmente, a função objetivo não aceitava MDGs com peso, e tampouco módulos com autorrelacionamento. Então, para a execução dos experimentos desta seção, foi necessária uma pequena adaptação na função objetivo utilizada pela ILS_CMS para que esta conseguisse utilizar MDGs com peso e também com autorrelacionamentos. As Tabelas 4.16 a 4.19 exibem as médias dos valores MQ obtidos pelos métodos LNS_CMS e ILS_CMS para cada uma das categorias de instância. A coluna "LNS_CMS > ILS_CMS" exibe os valores *p-value* e tamanho de efeito (ES) para uma comparação entre os valores de MQ obtidos entre os métodos para cada instância.

Tabela 4.16: Médias e desvios-padrão dos valores de MQ, *p-value* e tamanho de efeito para instâncias da categoria Pequena.

	LNS_CMS	**ILS_CMS**	**LNS_CMS > ILS_CMS**	
Instância	**Média (MQ)**	**Média (MQ)**	**PV**	**ES**
squid	1,0000±0,00	1,0000±0,00	-	50%
small	1,5238±0,00	1,8333±0,00	< 0,001	0%
compiler	1,4968±0,00	1,5065±0,00	< 0,001	0%
random	2,4409±0,00	2,4409±0,00	< 0,001	100%
regexp	2,4470±0,00	2,3016±0,00	< 0,001	100%
jstl	5,0000±0,00	5,0000±0,00	-	50%
lab4	3,4000±0,00	3,4000±0,00	< 0,001	100%
netkit-ping	1,0000±0,00	1,0000±0,00	-	50%
nss_ldap	0,9763±0,00	1,2889±0,00	< 0,001	0%
nos	1,6565±0,00	1,6743±0,00	< 0,001	0%

Continua na próxima página

Continuação da página anterior

	LNS	ILS	LNS > ILS	
Instância	**Média (MQ)**	**Média (MQ)**	**PV**	**ES**
lslayout	1,8171±0,00	1,8613±0,00	< 0,001	0%
boxer	3,1011±0,00	3,1011±0,00	< 0,001	0%
netkit-tftpd	1,1442±0,00	1,4250±0,00	< 0,001	0%
sharutils	2,5338±0,00	2,4409±0,00	< 0,001	100%
mtunis	2,2856±0,04	2,3145±0,00	< 0,001	0%
spdb	5,5897±0,00	5,5897±0,00	< 0,001	100%
xtell	2,0052±0,00	1,3998±0,00	< 0,001	100%
bunch	2,4026±0,00	2,4056±0,00	< 0,001	0%
ispell	2,3323±0,00	2,3505±0,01	< 0,001	21%
netkit-inetd	1,3121±0,00	1,5017±0,00	< 0,001	0%
nanoxml	3,8171±0,00	3,8173±0,00	< 0,001	97%
ciald	2,8459±0,00	2,8472±0,00	< 0,001	75%
jodamoney	2,7489±0,00	2,7489±0,00	< 0,001	0%
modulizer	2,7579±0,00	2,7579±0,00	< 0,001	100%
bootp	2,1985±0,00	2,4576±0,00	< 0,001	0%
jxlsreader	3,5921±0,00	3,5967±0,00	< 0,001	0%
sysklogd-1	1,6870±0,00	1,8621±0,00	< 0,001	0%
telnetd	1,8475±0,00	1,7154±0,02	< 0,001	100%
crond	2,3030±0,00	1,9721±0,00	< 0,001	100%
netkit-ftp	1,7562±0,00	1,7622±0,00	< 0,001	0%
rcs	2,2261±0,00	2,2398±0,02	0,0047	61%
seemp	4,6536±0,00	4,6536±0,00	< 0,001	0%
dhcpd-2	3,4889±0,00	3,1351±0,01	< 0,001	100%
cyrus-sasl	3,2518±0,00	3,0125±0,00	< 0,001	100%
tcsh	1,1945±0,00	1,5710±0,00	< 0,001	0%
micq	2,1329±0,00	2,4194±0,01	< 0,001	0%
apache_zip	5,7663±0,00	5,7649±0,00	< 0,001	100%
star	3,8250±0,01	3,8184±0,01	0,0098	40%
bison	2,6828±0,01	2,6886±0,01	< 0,001	27%
cia	3,7306±0,01	3,5783±0,02	< 0,001	100%
stunnel	2,5261±0,00	2,7324±0,00	< 0,001	0%
minicom	2,5758±0,00	2,2986±0,00	< 0,001	100%
mailx	3,2395±0,00	2,3455±0,01	< 0,001	100%
dot	2,8379±0,01	2,8357±0,01	0,0873	57%
screen	2,2455±0,00	2,1283±0,01	< 0,001	100%
slang	4,6790±0,00	3,7513±0,01	< 0,001	100%

Continua na próxima página

Continuação da página anterior

	LNS	ILS	LNS > ILS	
Instância	**Média (MQ)**	**Média (MQ)**	**PV**	**ES**
slrn	2,3836±0,00	2,6138±0,01	< 0,001	0%
net-tools	4,2932±0,01	4,0859±0,00	< 0,001	100%
graph10up49	1,2520±0,00	1,2530±0,00	0,0169	40%
wu-ftpd-1	2,4124±0,01	2,6627±0,02	< 0,001	0%
joe	3,3186±0,01	2,4356±0,01	< 0,001	100%
hw	8,4967±0,00	2,0000±0,00	< 0,001	100%
imapd-1	3,6250±0,00	3,4859±0,01	< 0,001	100%
wu-ftpd-3	3,3402±0,00	3,1349±0,04	< 0,001	100%
udt-java	5,2828±0,00	5,2805±0,00	< 0,001	99%
javaocr	9,0207±0,01	9,0057±0,03	< 0,001	34%
dhcpd-1	3,9608±0,00	2,9509±0,01	< 0,001	100%
icecast	2,7474±0,00	2,2994±0,01	< 0,001	100%
pfcda_base	7,3319±0,00	7,3310±0,01	< 0,001	64%
servletapi	9,7475±0,04	9,8092±0,07	< 0,001	21%
php	5,3242±0,00	4,5869±0,01	< 0,001	100%
bunch2	7,7115±0,02	7,7056±0,01	0,8559	49%
forms	8,3258±0,00	8,3258±0,00	< 0,001	0%
jscatterplot	10,7414±0,00	10,7409±0,01	< 0,001	99%

Para o total de 66 instâncias da categoria Pequena (Tabela 4.16) o método LNS_CMS atinge médias de MQ mais altas que o método ILS_CMS em 31 instâncias. Em 28 instâncias o resultado é favorável ao método ILS_CMS e em outras 5 as médias são iguais. Conclui-se que a heurística LNS_CMS obteve um desempenho muito próximo da heurística ILS_CMS para as instâncias da categoria Pequena.

Observando-se as 29 instâncias da categoria Média (Tabela 4.17) o método LNS_CMS supera o ILS_CMS em 24 instâncias e é derrotado em apenas 4 instâncias. Assim, para a categoria Média, a LNS_CMS mostra-se mais eficaz que a ILS_CMS, obtendo melhores médias de MQ em mais de 80% das instâncias.

Tabela 4.17: Médias e desvios-padrão dos valores de MQ, *p-value* e tamanho de efeito para instâncias da categoria Média.

	LNS	ILS	LNS > ILS	
Instância	**Média (MQ)**	**Média (MQ)**	**PV**	**ES**
jxlscore	9,7950±0,01	9,7908±0,01	0,0175	60%
elm-2	3,7999±0,01	4,2361±0,01	< 0,001	0%
jfluid	6,5796±0,00	6,5791±0,00	< 0,001	100%
grappa	12,7054±0,00	12,6981±0,00	< 0,001	95%
elm-1	4,3057±0,01	3,8882±0,01	< 0,001	100%
gnupg	7,1306±0,00	5,8805±0,02	< 0,001	100%
inn	8,0163±0,01	7,6344±0,01	< 0,001	100%
bash	5,8843±0,01	5,0764±0,00	< 0,001	100%
jpassword	10,6258±0,02	10,6136±0,04	< 0,001	64%
bitchx	4,3586±0,01	3,2646±0,01	< 0,001	100%
junit	11,0900±0,00	11,0901±0,00	< 0,001	0%
xntp	8,3492±0,00	8,3689±0,00	< 0,001	0%
acqcigna	16,5166±0,01	16,4550±0,03	< 0,001	99%
bunch_2	13,6076±0,01	13,5759±0,03	< 0,001	79%
exim	6,6160±0,00	6,0754±0,02	< 0,001	100%
xmldom	10,9236±0,00	10,8565±0,02	< 0,001	100%
cia++	15,4724±0,00	15,3737±0,04	< 0,001	100%
tinytim	12,5200±0,01	12,5078±0,02	< 0,001	75%
mod_ssl	10,1020±0,01	8,3891±0,01	< 0,001	100%
jkaryoscope	18,9867±0,00	18,9714±0,03	< 0,001	30%
ncurses	11,8263±0,00	11,3743±0,02	< 0,001	100%
gae_plugin_core	17,3358±0,01	17,2777±0,02	< 0,001	99%
lynx	4,9714±0,01	4,2029±0,01	< 0,001	100%
javacc	10,6930±0,03	10,6175±0,05	< 0,001	89%
lucent	59,9488±0,00	59,9377±0,00	< 0,001	100%
javageom	14,0995±0,01	14,0740±0,02	< 0,001	92%
incl	13,6122±0,01	13,6127±0,01	< 0,001	87%
jdendogram	26,0675±0,01	26,0663±0,01	0,0747	43%
xmlapi	19,0963±0,01	18,9782±0,04	< 0,001	100%

Tabela 4.18: Médias e desvios-padrão dos valores de MQ, *p-value* e tamanho de efeito para instâncias da categoria Grande.

	LNS_CMS	ILS_CMS	LNS_CMS > ILS_CMS	
Instância	**Média (MQ)**	**Média (MQ)**	**PV**	**ES**
jmetal	12,5394±0,03	12,4246±0,03	< 0,001	99%
graph10up193	2,2465±0,01	2,2314±0,01	< 0,001	94%
dom4j	19,2156±0,02	19,1278±0,02	< 0,001	100%
nmh	9,4022±0,01	7,7348±0,02	< 0,001	100%
pdf_renderer	22,3572±0,00	22,3437±0,02	< 0,001	99%
jung_graph_model	32,0257±0,01	31,9700±0,02	< 0,001	100%
jung_visualization	21,8301±0,02	21,7799±0,04	< 0,001	86%
jconsole	26,5184±0,00	26,5056±0,01	< 0,001	95%
pfcda_swing	29,0720±0,02	28,9778±0,03	< 0,001	98%
jml-1.0b4	17,5540±0,02	17,4537±0,03	< 0,001	99%
jpassword2	28,5921±0,01	28,4997±0,04	< 0,001	100%
notelab-full	29,6446±0,04	29,4582±0,05	< 0,001	100%
poormans CMS	34,2355±0,01	34,1402±0,03	< 0,001	100%
log4j	32,5039±0,02	32,4571±0,01	< 0,001	98%
jtreeview	48,1280±0,00	48,1083±0,02	< 0,001	90%
bunchall	16,9749±0,01	16,8710±0,03	< 0,001	100%
jace	26,8214±0,01	26,8159±0,01	0,2341	45%
javaws	38,3168±0,01	38,2750±0,02	< 0,001	94%

Nas 31 instâncias das categorias Grande e Muito Grande (Tabela 4.18 e Tabela 4.19), o método LNS_CMS só não supera o ILS_CMS em uma instância, na qual ocorre empate. Assim, para as maiores instâncias a heurística LNS_CMS se mostrou superior a ILS_CMS.

A Tabela 4.20 cujo nome da categoria é apontado na primeira coluna exibe o somatório dos tempos médios por instância em cada categoria para os métodos LNS_CMS e ILS_CMS na segunda e na terceira coluna, respectivamente. A quarta coluna exibe a quantidade de instâncias nas quais a média do tempo de processamento foi menor com a heurística LNS_CMS. A quinta coluna exibe esta mesma informação, porém, quando o tempo de processamento da heurística ILS_CMS é menor. Observa-se que a heurística LNS_CMS exige um esforço computacional menor nas instâncias da categoria Pequena e Média. Nestas duas categorias a LNS_CMS obtém qualidade de resultados melhores que a ILS_CMS. Neste cenário a heurística LNS_CMS consegue ser mais eficiente e eficaz que a ILS_CMS. Para as instâncias das categorias Grande e Muito Grande, a LNS_CMS obtém médias de MQ mais altas que a ILS_CMS em 30 das 31 instâncias. Contudo,

o tempo de processamento exigido pela LNS_CMS é consideravelmente maior. Então, para estas duas categorias, a busca LNS_CMS é mais eficaz, porém menos eficiente que a ILS_CMS.

Tabela 4.19: Médias e desvios-padrão dos valores de MQ, *p-value* e tamanho de efeito para instâncias da categoria Muito Grande.

	LNS_CMS	**ILS_CMS**	**LNS_CMS > ILS_CMS**	
Instância	**Média (MQ)**	**Média (MQ)**	**PV**	**ES**
swing	45,3645±0,03	44,9394±0,09	< 0,001	100%
lwjgl-2.8.4	37,1388±0,01	37,0075±0,05	< 0,001	100%
res_cobol	15,9761±0,01	15,9733±0,01	< 0,001	88%
ping_libc	51,8583±0,01	46,5815±0,02	< 0,001	100%
y_base	58,3085±0,02	58,1103±0,05	< 0,001	100%
krb5	54,5918±0,03	49,3718±0,07	< 0,001	100%
apache_ant_taskdef	66,6347±0,03	66,5245±0,02	< 0,001	99%
itextpdf	58,6549±0,03	58,3277±0,05	< 0,001	100%
apache_lucene_core	76,8792±0,04	76,5899±0,03	< 0,001	100%
eclipse_jgit	86,5484±0,04	86,2953±0,02	< 0,001	100%
linux	54,6376±0,03	54,1417±0,00	< 0,001	100%
apache_ant	102,7430±0,05	102,4097±0,00	< 0,001	100%
ylayout	111,0320±0,03	110,8846±0,00	< 0,001	100%

Tabela 4.20: Somatório dos tempos médios de processamento com ILS e LNS. Quantidade de instâncias onde o tempo de um método foi menor que o do outro.

	Tempo médio de processamento (s)			
Categoria	**LNS**	**ILS**	**LNS < ILS**	**LNS > ILS**
Pequena	1,45	3,32	50	14
Média	10,07	13,77	22	7
Grande	102,82	37,20	2	16
Muito Grande	4.543,65	232,79	0	13

A Figura 4.8 exibe 4 gráficos (um para cada categoria) com informações sobre o tempo de processamento da LNS_CMS e da ILS_CMS. Observa-se que os tempos de processamento para as instâncias das categorias Pequena e Média são muito próximos, além de não haver um método que seja sempre mais rápido ou mais lento que o outro. Contudo, para as instâncias da categoria Grande o tempo de processamento da LNS_CMS é, quase sempre, maior. Por fim, para a categoria Muito Grande, o tempo de processamento da LNS_CMS é sempre maior.

A heurística LNS_CMS consegue tempo de processamento menor do que o obtido pela heurística ILS_CMS em 74 das 124 instâncias. Na média, a heurística LNS_CMS foi mais rápida que a ILS_CMS para as categorias Pequena e Média. Porém, para as categorias Grande e Muito Grande, a heurística LNS_CMS foi muito mais lenta, chegando a ter a média de tempo de processamento quase vinte vezes maior que a ILS_CMS na categoria Muito Grande.

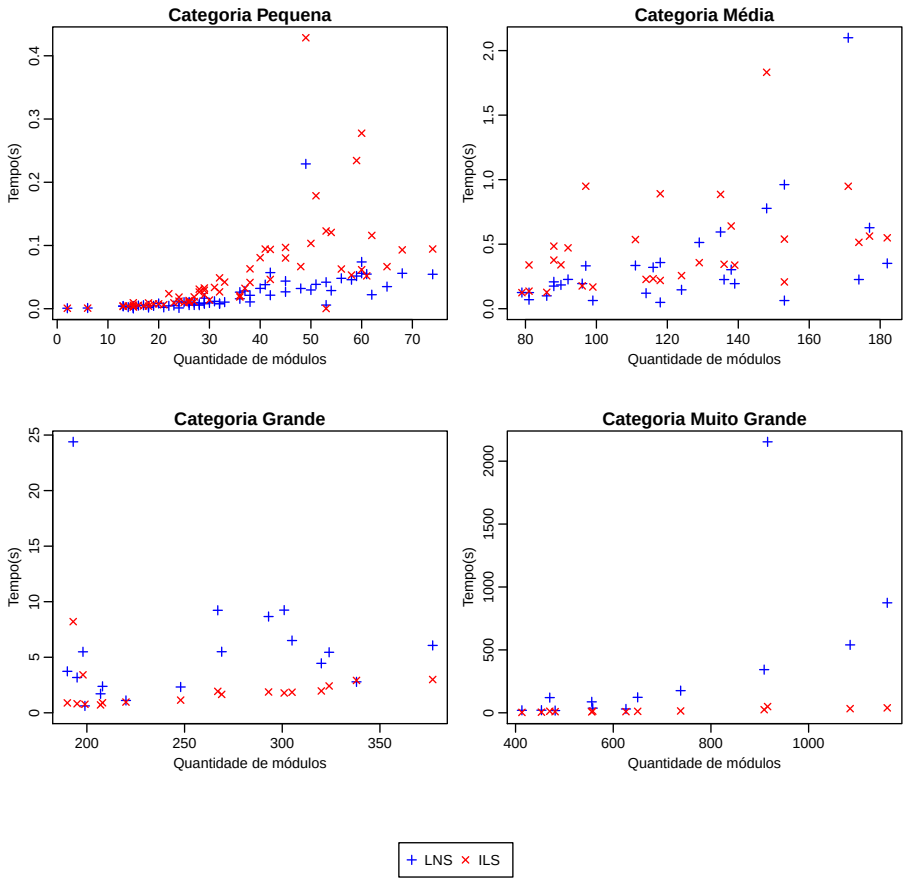


Figura 4.8: Variação do tempo(s) médio de processamento das heurísticas LNS e ILS de acordo com o número de módulos.

Considerando-se todas as instâncias, a média do tempo de processamento com a heurística LNS_CMS é aproximadamente 4.660 segundos, contra 287 segundos obtidos pela ILS_CMS. Então, sabendo-se que para cada instância a execução foi repetida 100 vezes, o tempo aproximado do experimento com a LNS_CMS é de 129 horas e para a ILS_CMS o tempo aproximado é de 8 horas.

As Figuras 4.9 e 4.10 exibem 12 *boxplots*, sendo três por cada categoria de instância. As instâncias foram selecionadas dividindo-se cada categoria em três partes e pegando-se o elemento central de cada parte. O *boxplot* é um gráfico formado pela mediana, primeiro

e terceiro quartil do conjunto de dados. Os gráficos mostram que quase não há interseção entre os dados gerados pela LNS_CMS e pela ILS_CMS, ou seja, as soluções do quartil inferior da LNS_CMS (25% piores soluções) são melhores que as soluções do quartil superior da ILS_CMS (25% melhores soluções), exceto para as instâncias da categoria Pequena.

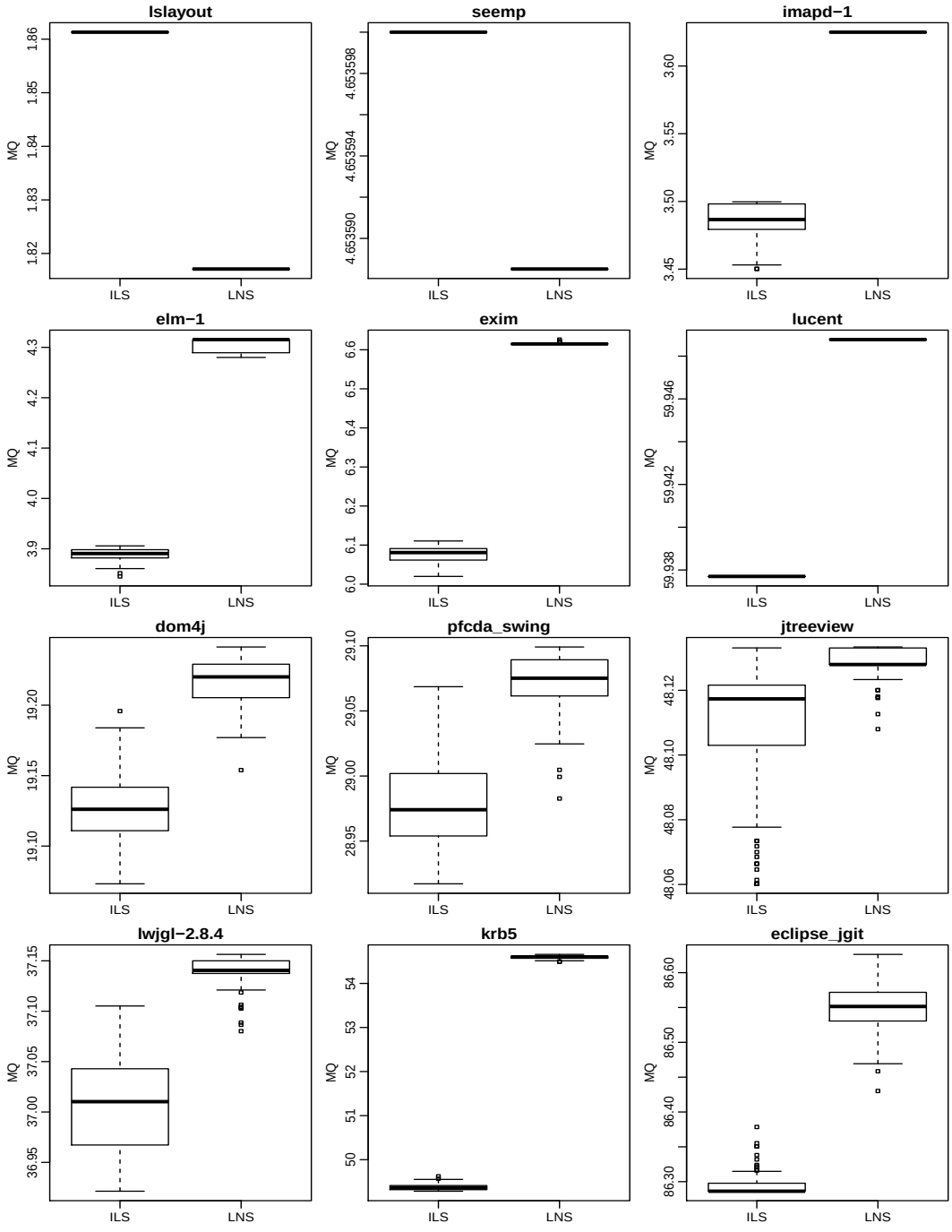


Figura 4.9: *Boxplot* de 12 instâncias com o MQ obtido pelos métodos LNS e ILS.

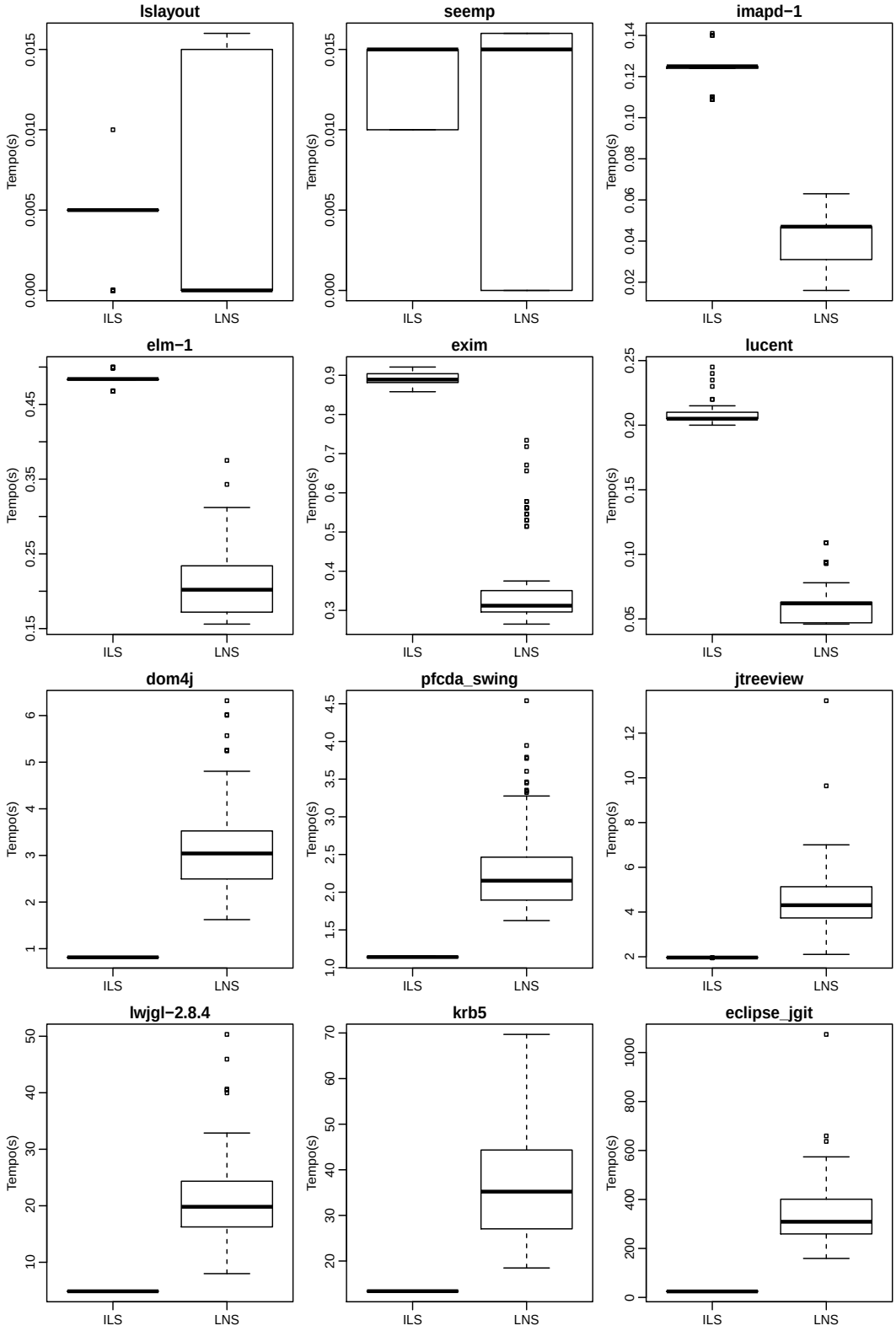


Figura 4.10: *Boxplot* de 12 instâncias com o tempo(s) médio de processamento obtido pelos métodos LNS e ILS.

4.6 Estudo experimental comparativo entre a heurística LNS_CMS e os melhores resultados obtidos na literatura

Com o intuito de responder as QP2 e QP3 , comparou-se o resultado de 100 execuções do LNS_CMS com os resultados disponíveis na literatura para as 124 instâncias disponíveis.

A Tabela 4.21 exibe os dados da execução da heurística LNS_CMS e também os melhores resultados encontrados na literatura para as instâncias estudadas. A primeira coluna da tabela exibe o nome da instância. O "*" significa que o valor ótimo já foi alcançado na literatura. A segunda coluna exibe a categoria da instância. A terceira coluna da tabela exibe o maior valor de MQ encontrado na literatura, ou a maior média, quando o método não é exato. A quarta coluna exibe o tempo gasto (normalizado) para a obtenção do MQ. A quinta coluna exibe o nome do método da literatura que conseguiu gerar o melhor MQ. Os métodos da literatura são: ECA, proposto por Praditwong [25], GGA, proposto por Praditwong [24], AG_GGA, BL, ILS_CMS, propostos por Pinto [22], $MILP_1$, $MILP_{2,1}$, $MILP^{+}_{2,1}$, propostos por Köhler et al. [12]. As colunas MQ Min., MQ Med. e MQ Max. exibem, respectivamente, o valor de MQ mínimo, a média com o desvio padrão e o valor de MQ máximo obtidos pelo método LNS_CMS. As colunas T. (s) Min., T. (s) Med. e T. (s) Max. exibem, respectivamente, o tempo de processamento mínimo, médio com o desvio padrão e o tempo máximo utilizado pelo método LNS_CMS. A última coluna exibe a diferença percentual entre o MQ obtido na literatura e o MQ obtido pela busca LNS_CMS, dado pela equação 4.1.

$$gap = \frac{\text{MQ LNS\_CMS} - \text{MQ Lit.}}{\text{MQ Lit.}} \cdot 100 \tag{4.1}$$

A comparação do tempo de processamento não pôde ser feita diretamente, pois os experimentos da literatura foram executados em máquinas com capacidades diferentes. Então, para uma comparação de tempo justa, é necessário efetuar uma normalização dos tempos obtidos. A normalização é feita descobrindo-se um fator multiplicador de um computador em relação a outro e multiplicando-se o tempo de um dos computadores pelo fator. Para descobrir o fator multiplicador entre a capacidade de dois computadores é necessário que ambos executem a mesma tarefa. Esse tipo de teste é conhecido por *benchmark*. O site https://www.spec.org/ possui dados de *benchmark* para computadores com diversas configurações distintas. Para a normalização dos tempos obtidos pela literatura foi utilizado o valor obtido no *benchmark* chamado CINT2006. O valor obtido nesse teste é o tempo que o computador levou para executar um conjunto de tarefas específicas. Nos

comparativos do site *spec.org* não havia nenhum computador com configuração idêntica ao utilizado neste trabalho. Porém, havia dados para um computador semelhante: os dados do CINT2006 são de um computador com 8GB de memória RAM, enquanto que o computador utilizado tem apenas 4GB. O resultado deste computador é 45,1. Sendo assim, esse valor foi utilizado como referência. A tabela 4.21 mostra apenas os tempos de processamento da literatura que são comparáveis aos do presente trabalho, após a aplicação do fator multiplicativo. Para aqueles trabalhos em que a informação do processador não foi disponibilizada, considera-se que os tempos não são comparáveis com os deste trabalho e, portanto, estes dados não serão exibidos.

Algumas das referências na literatura possuem o valor de MQ arrendondado para duas casas decimais. Então, para evitar diferenças devido a arredondamentos com precisões diferentes, todos os valores de MQ foram arredondados para duas casas decimais antes de serem comparados. Este arredondamento foi utilizado para o cálculo do *gap* (diferença percentual entre o melhor resultado da literatura e o resultado obtido pelo método LNS_CMS) e também para o cálculo do número de vitórias, empates e derrotas, que estão na Tabela 4.22. Valores de *gap* positivos indicam que a heurística LNS_CMS superou os resultados encontrados na literatura, enquanto que valores negativos indicam o oposto, ou seja, os resultados do LNS_CMS foram inferiores aos da literatura. Quando o valor do *gap* é 0, ou próximo, isto indica que os resultados são similares.

Tabela 4.21: Tabela comparativa dos melhores resultados da literatura e dos resultados obtidos pelo LNS com a configuração MISTA 1000.

		Literatura			LNS_CMS						
Instância	**C.**	**MQ**	**T. (s)**	**Método**	**MQ Min.**	**MQ Med.**	**MQ Max.**	**T. (s) Min.**	**T. (s) Med.**	**T. (s) Max.**	**gap**
squid	P	1	0,00[e]	ILS_CMS	1,0000	1,0000±0,0000	1,0000	0,00	0,00±0,00	0,02	0,00%
small *	P	1,8333	0,01[a]	$MILP_1$	1,5238	1,5238±0,0000	1,5238	0,00	0,00±0,00	0,02	16,94%
compiler *	P	1,5065	8,33[a]	$MILP_1$	1,4968	1,4968±0,0000	1,4968	0,00	0,00±0,01	0,02	0,66%
random *	P	2,4409	0,57[a]	$MILP^{+}_{2,1}$	2,4409	2,4409±0,0000	2,4409	0,00	0,00±0,01	0,02	0,00%
regexp *	P	2,447	0,38[a]	$MILP^{+}_{2,1}$	2,4470	2,4470±0,0000	2,4470	0,00	0,00±0,01	0,02	0,00%
jstl	P	5	0,00[e]	ILS_CMS	5,0000	5,0000±0,0000	5,0000	0,00	0,00±0,01	0,02	0,00%
lab4 *	P	3,4	0,28[a]	$MILP_1$	3,4000	3,4000±0,0000	3,4000	0,00	0,00±0,01	0,02	0,00%
netkit-ping *	P	1	0,00[a]	$MILP^{+}_{2,1}$	1,0000	1,0000±0,0000	1,0000	0,00	0,00±0,00	0,02	0,00%
nss_ldap *	P	1,2889	0,01[e]	ILS_CMS	0,9763	0,9763±0,0000	0,9763	0,00	0,00±0,00	0,02	24,03%
nos *	P	1,6775	7.755,71[a]	$MILP_1$	1,6565	1,6565±0,0000	1,6565	0,00	0,01±0,01	0,02	1,19%
lslayout *	P	1,8613	3.449,31[a]	$MILP_1$	1,8171	1,8171±0,0000	1,8171	0,00	0,01±0,01	0,02	2,15%
boxer *	P	3,1011	2,29[a]	$MILP_1$	3,1011	3,1011±0,0000	3,1011	0,00	0,00±0,01	0,02	0,00%
netkit-tftpd *	P	1,425	0,01[e]	ILS_CMS	1,1442	1,1442±0,0000	1,1442	0,00	0,00±0,01	0,02	20,28%
sharutils *	P	2,5492	0,53[a]	$MILP^{+}_{2,1}$	2,5338	2,5338±0,0000	2,5338	0,00	0,00±0,01	0,02	0,78%
Mtunis *	P	2,3145	-	$MILP_1$	2,2406	2,2856±0,0362	2,3145	0,00	0,01±0,01	0,02	0,87%
spdb *	P	5,5897	0,02[a]	$MILP_1$	5,5897	5,5897±0,0000	5,5897	0,00	0,00±0,01	0,02	0,00%
xtell *	P	2,0052	0,44[a]	$MILP^{+}_{2,1}$	2,0052	2,0052±0,0000	2,0052	0,00	0,00±0,01	0,02	0,00%
Bunch *	P	2,406	21,96[a]	$MILP_1$	2,4026	2,4026±0,0000	2,4026	0,00	0,01±0,01	0,02	0,41%
Ispell *	P	2,3639	-	$MILP_1$	2,3323	2,3323±0,0000	2,3323	0,00	0,01±0,01	0,02	1,27%
netkit-inetd *	P	1,5017	0,02[e]	ILS_CMS	1,3121	1,3121±0,0000	1,3121	0,00	0,00±0,00	0,02	12,67%
nanoxml	P	3,82	0,01[b]	ILS_CMS	3,8082	3,8171±0,0016	3,8173	0,00	0,01±0,01	0,02	0,00%
ciald *	P	2,85	-	$MILP_1$	2,8459	2,8459±0,0000	2,8459	0,00	0,01±0,01	0,02	0,00%
jodamoney	P	2,75	0,01[b]	ILS_CMS	2,7489	2,7489±0,0000	2,7489	0,00	0,01±0,01	0,03	0,00%
Modulizer *	P	2,7579	-	$MILP_1$	2,7579	2,7579±0,0000	2,7579	0,00	0,01±0,01	0,02	0,00%
bootp *	P	2,4576	0,02[e]	ILS_CMS	2,1985	2,1985±0,0000	2,1985	0,00	0,01±0,01	0,02	10,57%
jxlsreader	P	3,6	0,01[b]	ILS_CMS	3,5921	3,5921±0,0000	3,5921	0,00	0,01±0,01	0,02	0,28%
sysklogd-1 *	P	1,8621	0,03[e]	ILS_CMS	1,6870	1,6870±0,0000	1,6870	0,00	0,01±0,01	0,02	9,14%

Continua na próxima página

Continuação da página anterior

		Literatura			LNS_CMS						
Instância	**C.**	**MQ**	**T. (s)**	**Método**	**MQ Min.**	**MQ Med.**	**MQ Max.**	**T. (s) Min.**	**T. (s) Med.**	**T. (s) Max.**	**gap**
telnetd *	P	1,8475	0,73[a]	$MILP^{+}_{2,1}$	1,8475	1,8475±0,0000	1,8475	0,00	0,01±0,01	0,02	0,00%
crond *	P	2,303	4,07[a]	$MILP^{+}_{2,1}$	2,3030	2,3030±0,0000	2,3030	0,00	0,01±0,01	0,02	0,00%
netkit-ftp *	P	1,768	0,69[a]	$MILP^{+}_{2,1}$	1,7562	1,7562±0,0000	1,7562	0,00	0,01±0,01	0,02	0,56%
rcs	P	2,2775	13.554,64[a]	$MILP_{2,1}$	2,2174	2,2261±0,0010	2,2262	0,00	0,02±0,01	0,03	2,19%
seemp	P	4,6536	0,01[e]	ILS_CMS	4,6536	4,6536±0,0000	4,6536	0,00	0,01±0,01	0,02	0,00%
dhcpd-2 *	P	3,4944	20,02[a]	$MILP^{+}_{2,1}$	3,4889	3,4889±0,0000	3,4889	0,00	0,01±0,01	0,03	0,00%
cyrus-sasl *	P	3,2518	1,01[a]	$MILP^{+}_{2,1}$	3,2518	3,2518±0,0000	3,2518	0,00	0,01±0,01	0,02	0,00%
tcsh *	P	1,571	0,05[e]	ILS_CMS	1,1945	1,1945±0,0000	1,1945	0,00	0,01±0,01	0,03	24,20%
micq *	P	2,4194	0,04[e]	ILS_CMS	2,1329	2,1329±0,0000	2,1329	0,00	0,01±0,01	0,02	11,98%
apache_zip	P	5,77	0,02[b]	ILS_CMS	5,7663	5,7663±0,0000	5,7663	0,00	0,02±0,00	0,03	0,00%
Star	P	3,8321	-	$MILP_{2,1}$	3,8082	3,8250±0,0050	3,8321	0,02	0,03±0,01	0,05	0,00%
bison	P	2,7039	13.554,64[a]	$MILP_1$	2,6715	2,6828±0,0065	2,7039	0,02	0,03±0,01	0,05	0,74%
cia	P	3,5783	0,06[e]	ILS_CMS	3,7289	3,7306±0,0059	3,7507	0,02	0,02±0,01	0,03	-4,19%
stunnel *	P	2,7324	0,04[e]	ILS_CMS	2,5261	2,5261±0,0000	2,5261	0,00	0,01±0,01	0,02	7,33%
minicom *	P	2,5759	956,87[a]	$MILP^{+}_{2,1}$	2,5758	2,5758±0,0000	2,5758	0,02	0,03±0,01	0,05	0,00%
mailx	P	3,2353	-	$MILP^{+}_{2,1}$	3,2337	3,2395±0,0020	3,2402	0,02	0,04±0,01	0,05	0,00%
dot	P	2,8357	0,05[e]	ILS_CMS	2,8290	2,8379±0,0062	2,8470	0,03	0,06±0,02	0,12	0,00%
screen	P	2,1283	0,09[e]	ILS_CMS	2,2455	2,2455±0,0000	2,2455	0,02	0,02±0,01	0,03	-5,63%
slang	P	4,6794	-	$MILP^{+}_{2,1}$	4,6697	4,6790±0,0019	4,6794	0,03	0,04±0,01	0,06	0,00%
slrn	P	2,6138	0,10[e]	ILS_CMS	2,3619	2,3836±0,0046	2,3928	0,02	0,03±0,01	0,05	8,81%
net-tools	P	4,3159	13.554,64[a]	$MILP^{+}_{2,1}$	4,2869	4,2932±0,0072	4,3159	0,02	0,03±0,01	0,06	0,69%
graph10up49	P	1,253	0,43[e]	ILS_CMS	1,2485	1,2520±0,0025	1,2590	0,09	0,23±0,09	0,81	0,00%
wu-ftpd-1 *	P	2,6627	0,10[e]	ILS_CMS	2,4068	2,4124±0,0099	2,4371	0,02	0,03±0,01	0,05	9,40%
joe	P	3,2926	-	$MILP^{+}_{2,1}$	3,3025	3,3186±0,0138	3,3411	0,03	0,04±0,01	0,06	-0,91%
hw	P	2	0,00[e]	ILS_CMS	8,4967	8,4967±0,0000	8,4967	0,00	0,01±0,01	0,02	-325,00%
imapd-1 *	P	3,625	254,83[a]	$MILP^{+}_{2,1}$	3,6250	3,6250±0,0000	3,6250	0,02	0,04±0,01	0,06	0,00%
wu-ftpd-3	P	3,3953	13.554,64[a]	$MILP^{+}_{2,1}$	3,3317	3,3402±0,0015	3,3405	0,02	0,03±0,01	0,03	1,76%
udt-java	P	5,2805	0,06[e]	ILS_CMS	5,2708	5,2828±0,0012	5,2829	0,03	0,05±0,01	0,06	0,00%
javaocr	P	9,02	3,94[b]	AG_GGA	8,9657	9,0207±0,0140	9,0242	0,02	0,05±0,01	0,08	0,00%
dhcpd-1	P	3,9359	-	$MILP^{+}_{2,1}$	3,9545	3,9608±0,0006	3,9609	0,03	0,05±0,01	0,08	-0,51%

Continua na próxima página

Continuação da página anterior

Instância	C.	Literatura			LNS_CMS						gap
		MQ	T. (s)	Método	MQ Min.	MQ Med.	MQ Max.	T. (s) Min.	T. (s) Med.	T. (s) Max.	
icecast	P	2,7544	-	$MILP^{+}_{2,1}$	2,7474	2,7474±0,0001	2,7482	0,05	0,06±0,01	0,08	0,00%
pfcda_base	P	7,331	0,06[e]	ILS_CMS	7,3269	7,3319±0,0022	7,3328	0,03	0,07±0,02	0,13	0,00%
servletapi	P	9,8092	0,05[e]	ILS_CMS	9,6856	9,7475±0,0356	9,8715	0,03	0,05±0,01	0,09	0,61%
php *	P	5,3242	278,62[a]	$MILP^{+}_{2,1}$	5,3242	5,3242±0,0000	5,3242	0,02	0,02±0,01	0,05	0,00%
bunch2	P	7,7056	0,07[e]	ILS_CMS	7,6751	7,7115±0,0154	7,7251	0,02	0,04±0,01	0,08	0,00%
forms	P	8,33	0,11[b]	ILS_CMS	8,3258	8,3258±0,0000	8,3258	0,05	0,06±0,01	0,06	0,00%
jscatterplot	P	10,7409	0,09[e]	ILS_CMS	10,6934	10,7414±0,0049	10,7419	0,05	0,05±0,01	0,11	0,00%
jxlscore	M	9,7908	0,12[e]	ILS_CMS	9,7742	9,7950±0,0078	9,8050	0,08	0,13±0,05	0,33	-0,10%
elm-2	M	4,2361	0,34[e]	ILS_CMS	3,7746	3,7999±0,0070	3,8296	0,09	0,12±0,03	0,20	10,38%
jfluid	M	6,58	0,15[b]	ILS_CMS	6,5796	6,5796±0,0000	6,5796	0,06	0,07±0,01	0,11	0,00%
grappa	M	12,7	0,13[b]	ILS_CMS	12,7054	12,7054±0,0000	12,7054	0,08	0,10±0,02	0,19	-0,08%
elm-1	M	3,8882	0,48[e]	ILS_CMS	4,2800	4,3057±0,0137	4,3154	0,16	0,21±0,05	0,38	-10,80%
gnupg	M	6,905	0,00[c]	ECA	7,1273	7,1306±0,0030	7,1433	0,09	0,18±0,02	0,23	-3,18%
inn	M	7,911	-	GGA	7,9953	8,0163±0,0058	8,0180	0,14	0,18±0,04	0,30	-1,39%
bash	M	5,0764	0,47[e]	ILS_CMS	5,8707	5,8843±0,0081	5,9124	0,16	0,23±0,06	0,36	-15,75%
jpassword	M	10,6136	0,18[e]	ILS_CMS	10,5942	10,6258±0,0235	10,6877	0,11	0,19±0,07	0,52	-0,19%
bitchx	M	4,267	0,00[c]	ECA	4,3490	4,3586±0,0130	4,3860	0,27	0,33±0,09	0,78	-2,11%
junit	M	11,0901	0,17[e]	ILS_CMS	11,0894	11,0900±0,0002	11,0901	0,05	0,06±0,01	0,13	0,00%
xntp	M	8,3689	0,54[e]	ILS_CMS	8,3298	8,3492±0,0040	8,3524	0,17	0,33±0,05	0,42	0,24%
acqCIGNA	M	16,455	0,23[e]	ILS_CMS	16,5000	16,5166±0,0089	16,5971	0,08	0,12±0,02	0,17	-0,43%
bunch_2	M	13,5759	0,23[e]	ILS_CMS	13,5892	13,6076±0,0089	13,6204	0,16	0,32±0,08	0,69	-0,22%
exim	M	6,361	0,00[c]	ECA	6,6149	6,6160±0,0024	6,6260	0,27	0,36±0,11	0,73	-4,09%
xmldom	M	10,88	39,77[b]	AG_GGA	10,9236	10,9236±0,0000	10,9236	0,05	0,05±0,01	0,06	-0,37%
cia++	M	15,3737	0,26[e]	ILS_CMS	15,4724	15,4724±0,0000	15,4724	0,09	0,15±0,04	0,23	-0,65%
tinytim	M	12,51	0,38[b]	ILS_CMS	12,4937	12,5200±0,0063	12,5415	0,30	0,51±0,05	0,64	-0,08%
mod_ssl	M	9,831	-	GGA	10,0821	10,1020±0,0060	10,1118	0,33	0,59±0,18	1,01	-2,75%
jkaryoscope	M	18,98	0,35[b]	ILS_CMS	18,9783	18,9867±0,0017	18,9871	0,20	0,23±0,02	0,28	-0,05%
ncurses	M	11,452	-	GGA	11,8245	11,8263±0,0031	11,8318	0,23	0,30±0,10	0,69	-3,32%
gae_plugin_core	M	17,32	0,25[b]	BL	17,2929	17,3358±0,0070	17,3370	0,09	0,19±0,02	0,25	-0,12%
lynx	M	4,694	0,00[c]	ECA	4,9588	4,9714±0,0052	4,9814	0,36	0,78±0,32	2,53	-5,97%

Continua na próxima página

Continuação da página anterior

Instância	C.	Literatura		Método	LNS_CMS						gap
		MQ	T. (s)		MQ Min.	MQ Med.	MQ Max.	T. (s) Min.	T. (s) Med.	T. (s) Max.	
javacc	M	10,62	107,46[b]	AG_GGA	10,5550	10,6930±0,0344	10,7216	0,41	0,96±0,38	3,01	-0,66%
lucent	M	59,9377	0,21[e]	ILS_CMS	59,9488	59,9488±0,0000	59,9488	0,05	0,06±0,02	0,11	-0,02%
javaGeom	M	14,074	0,95[e]	ILS_CMS	14,0835	14,0995±0,0073	14,1148	1,16	2,10±0,83	6,38	-0,21%
incl	M	13,6127	0,51[e]	ILS_CMS	13,5964	13,6122±0,0062	13,6146	0,16	0,23±0,05	0,34	0,00%
jdendogram	M	26,07	0,57[b]	ILS_CMS	26,0569	26,0675±0,0117	26,0843	0,39	0,63±0,29	1,93	0,00%
xmlapi	M	18,99	0,55[b]	ILS_CMS	19,0745	19,0963±0,0054	19,0980	0,27	0,35±0,08	0,61	-0,58%
jmetal	G	12,4246	0,89[e]	ILS_CMS	12,4589	12,5394±0,0338	12,6282	1,48	3,73±1,87	13,54	-0,97%
graph10up193	G	2,2314	8,21[e]	ILS_CMS	2,2309	2,2465±0,0066	2,2599	8,07	24,39±12,88	86,03	-0,90%
dom4j	G	19,1278	0,82[e]	ILS_CMS	19,1539	19,2156±0,0176	19,2413	1,62	3,18±0,95	6,32	-0,47%
nmh	G	8,77	-	GGA	9,3863	9,4022±0,0069	9,4148	2,40	5,49±1,42	11,45	-7,18%
pdf_renderer	G	22,3437	0,77[e]	ILS_CMS	22,3436	22,3572±0,0014	22,3573	0,44	0,62±0,15	0,98	-0,09%
Jung_graph_model	G	31,97	0,71[e]	ILS_CMS	31,9839	32,0257±0,0126	32,0325	0,91	1,71±0,65	5,55	-0,19%
jung_visualization	G	21,7799	0,90[e]	ILS_CMS	21,7716	21,8301±0,0238	21,8614	1,11	2,38±0,82	5,27	-0,23%
jconsole	G	26,51	0,95[b]	ILS_CMS	26,5003	26,5184±0,0036	26,5195	0,81	1,11±0,28	2,11	-0,04%
pfcda_swing	G	28,99	1,15[b]	ILS_CMS	28,9827	29,0720±0,0217	29,0991	1,62	2,32±0,58	4,54	-0,28%
jml-1.0b4	G	17,4537	1,93[e]	ILS_CMS	17,4791	17,5540±0,0214	17,6074	4,12	9,23±2,82	20,56	-0,57%
jpassword2	G	28,4997	1,66[e]	ILS_CMS	28,5528	28,5921±0,0145	28,6251	2,53	5,50±1,76	16,99	-0,32%
notelab-full	G	29,49	1.268,55[b]	AG_GGA	29,5428	29,6446±0,0366	29,7388	3,76	8,66±2,61	16,33	-0,51%
poormans CMS	G	34,1402	1,79[e]	ILS_CMS	34,2008	34,2355±0,0130	34,2540	3,53	9,24±3,61	21,03	-0,29%
log4j	G	32,4571	1,85[e]	ILS_CMS	32,4501	32,5039±0,0178	32,5186	3,01	6,50±2,50	19,28	-0,12%
jtreeview	G	48,1083	1,97[e]	ILS_CMS	48,1080	48,1280±0,0047	48,1334	2,11	4,45±1,52	13,45	-0,04%
bunchall	G	16,871	2,41[e]	ILS_CMS	16,9356	16,9749±0,0140	16,9948	2,14	5,45±2,25	17,11	-0,59%
JACE	G	26,8159	2,91[e]	ILS_CMS	26,7758	26,8214±0,0099	26,8403	1,75	2,79±0,79	4,68	0,00%
javaws	G	38,275	2,99[e]	ILS_CMS	38,2648	38,3168±0,0090	38,3388	2,98	6,06±2,27	15,77	-0,10%
swing	MG	44,9394	3,22[e]	ILS_CMS	45,2718	45,3645±0,0342	45,4527	9,67	19,91±8,14	70,64	-0,93%
lwjgl-2.8.4	MG	37,0075	4,87[e]	ILS_CMS	37,0802	37,1388±0,0146	37,1564	7,99	20,80±7,64	50,33	-0,35%
res_cobol	MG	15,9733	12,45[e]	ILS_CMS	15,9510	15,9761±0,0090	16,0083	29,92	120,76±60,12	360,33	-0,06%
ping_libc	MG	46,5815	9,00[e]	ILS_CMS	51,8231	51,8583±0,0128	51,8938	9,49	17,86±7,42	50,06	-11,34%
y_base	MG	58,1103	7,32[e]	ILS_CMS	58,2228	58,3085±0,0245	58,3474	31,06	87,04±44,31	320,62	-0,34%
krb5	MG	49,3718	13,39[e]	ILS_CMS	54,4895	54,5918±0,0349	54,6586	18,47	37,07±12,05	69,69	-10,57%

Continua na próxima página

Continuação da página anterior

Instância	C.	Literatura MQ	Literatura T. (s)	Método	LNS_CMS MQ Min.	LNS_CMS MQ Med.	LNS_CMS MQ Max.	LNS_CMS T. (s) Min.	LNS_CMS T. (s) Med.	LNS_CMS T. (s) Max.	gap
apache_ant_taskdef	MG	66,5245	9,37[e]	ILS_CMS	66,5287	66,6347±0,0258	66,6626	15,58	30,46±12,20	94,82	-0,17%
itextpdf	MG	58,3277	11,68[e]	ILS_CMS	58,5555	58,6549±0,0348	58,7145	50,44	122,89±45,60	270,18	-0,55%
apache_lucene_core	MG	76,5899	14,68[e]	ILS_CMS	76,7611	76,8792±0,0440	76,9618	69,75	176,11±95,53	590,96	-0,38%
eclipse_jgit	MG	86,2953	24,87[e]	ILS_CMS	86,4302	86,5484±0,0373	86,6262	159,17	343,10±129,01	1073,98	-0,29%
linux	MG	54,1417	49,51[e]	ILS_CMS	54,5519	54,6376±0,0349	54,7275	882,07	2153,65±790,07	4536,69	-0,92%
apache_ant	MG	102,4097	33,06[e]	ILS_CMS	102,5824	102,7430±0,0477	102,8400	218,31	539,80±202,95	1295,19	-0,32%
ylayout	MG	110,8846	39,37[e]	ILS_CMS	110,9509	111,0320±0,0318	111,0931	353,36	874,21±285,89	1681,43	-0,14%

[a] Tempo original em um computador com 24 GB RAM, 2.67 GHz Intel Xeon.
Resultado CINT2006=36,6. Fator de multiplicação=1,2322 [12].

[b] Tempo original em um computador com 4GB RAM, 3.40 GHz Intel Core i7-2600.
Resultado CINT2006=45,1. Fator de multiplicação=1 [22].

[c] Tempo original em um computador desconhecido.
Resultado CINT2006 desconhecido [25].

[d] Tempo original em um computador desconhecido.
Resultado CINT2006 desconhecido [24].

[e] Experimento executado no presente trabalho.
Resultado CINT2006=45,1. Fator de multiplicação=1.

A Tabela 4.22 exibe um resumo dos resultados do experimento comparativo. Os dados na tabela mostram, para cada categoria de instâncias, a quantidade de vezes que o método LNS_CMS conseguiu média de qualidade de soluções melhores, piores ou iguais às encontradas na literatura. A busca LNS_CMS consegue melhores resultados que os encontrados na literatura em 58 instâncias.

Tabela 4.22: Comparativo entre as qualidades das soluções encontradas pela heurística LNS_CMS e os melhores resultados da literatura.

	Qualidade das soluções		
Categoria	**LNS_CMS > Lit.**	**LNS_CMS < Lit.**	**LNS_CMS=Lit.**
Pequena	5	26	33
Média	23	2	4
Grande	17	0	1
Muito Grande	13	0	0

A Tabela 4.23 exibe um resumo dos resultados da comparação entre os tempos de execução da busca LNS_CMS e a literatura. A segunda coluna exibe a quantidade de instâncias nas quais a heurística LNS_CMS terminou sua execução mais rapidamente que o método da literatura que gerou a melhor solução. A terceira coluna exibe a informação da quantidade de vezes em que a heurística LNS_CMS demorou mais do que o método da literatura. A coluna "Lit. s/tempo" exibe a quantidade de instâncias onde o tempo da literatura não foi informado ou foi descartado, pois não é comparável com este trabalho.

Tabela 4.23: Comparativo entre os tempos de execução da busca LNS_CMS e a literatura em segundos.

	Tempo de processamento		
	LNS_CMS < Lit.	**LNS_CMS > Lit.**	**Lit. s/tempo**
Pequena	45	9	10
Média	16	6	7
Grande	3	14	1
Muito Grande	0	13	0

A busca LNS_CMS conseguiu maior número de vitórias nas categorias Média, Grande e Muito Grande. O que a torna mais eficaz em encontrar boas soluções, em especial para as instâncias maiores.

4.7 Caracterização das soluções encontradas pelo método LNS_CMS

Nesta seção apresentam-se, para cada uma das 124 instâncias, as características da melhor solução encontrada pelo método LNS_CMS, considerando-se 100 execuções de cada instância.

A Tabela 4.24 exibe, na primeira coluna, o nome da instância. Na coluna C. é exibido a categoria da instância (P, M, G ou MG). Na terceira coluna é exibido o maior valor de MQ obtido pelo método LNS_CMS. A quarta coluna possui a informação da iteração na qual a melhor solução foi encontrada. A coluna Tot. *cl.* exibe a quantidade de *clusters* da melhor solução encontrada. As colunas > *cl.* e < *cl.* exibem a quantidade de módulos no maior e no menor *cluster* da solução. A coluna *Cl.* iso. exibe o total de *clusters* com apenas um módulo existentes na solução, mesmo quando este possui autorrelacionamento. A coluna $\frac{k}{n}\%$ exibe a razão percentual entre o número de clusters da solução obtida e o número de módulos da instância.

Observa-se que em 21 instâncias da categoria Pequena a busca LNS_CMS não conseguiu melhorar a solução inicial recebida (Iteração igual a zero). Este fato mostra que o algoritmo CAMQ, responsável pela construção da solução inicial, consegue gerar soluções de boa qualidade para esta classe de instâncias. Por outro lado, isto também mostra que a condição de parada da busca (1.000 mil iterações sem melhoria para cada algoritmo de reparação utilizado) é muito elevada. Além disso, observa-se que, em mais de 70% das instâncias, as soluções geradas possuem ao menos um *cluster* isolado. Isto é um indicativo que a função objetivo utilizada produz soluções incomuns para projetos de software, visto que, segundo Lanza et al. [15] o número de módulos dividido pelo número de *clusters* não deve ser um número muito pequeno nem muito grande. Os autores expõem que as médias observadas são de 17 e 19 para os softwares avaliados em Java e em C++, respectivamente. Este número pode ser sensível ao tamanho da instância, uma vez que o limite superior de seu valor é a quantidade de módulos da instância. Então, para se normalizar os números utilizou-se a razão $\frac{k}{n}$, onde k é o número de *clusters* obtido na melhor solução encontrada e n é número de módulos da instância após o pré-processamento de redução. Desta maneira os números gerados pertencem ao intervalo]0, 1], independente do tamanho da instância. Utilizando esta abordagem, os números obtidos por Lanza et al. são 5,88% ($\frac{1}{17} \cdot 100$) e 5,26% ($\frac{1}{19} \cdot 100$). As instâncias avaliadas pelo LNS_CMS possuem um total 16.313 módulos (após o pré-processamento de redução) e as soluções geradas possuem um total de 6.061 *clusters*. Assim, a razão entre os valores é 37,15%. Valor bem discrepante do observado pelos autores. A Figura 4.11 exibe um *boxplot* contendo os valores obtidos em cada instância. Observa-se que para todas as categorias os valores obtidos ficam próximos

aos 37%. Outra observação é que a medida que a categoria contém instâncias maiores, os primeiro e terceiro quartis ficam mais próximos, mostrando uma maior concentração dos valores. Para 80% das instâncias o valor de $\frac{n}{k}$ encontra-se no intervalo [30, 50]. A média e mediana obtidas são 39,23% e 36,21%, respectivamente.

Tabela 4.24: Características das melhores soluções obtidas pela heurística LNS_CMS.

Instância	**C.**	**Maior MQ**	**It.**	**Tot. *cl.***	**> *cl.***	**< *cl.***	***Cl.* iso.**	$\frac{k}{n} \cdot 100\%$
squid	P	1,0000	0	1	1	1	1	100,00%
small	P	1,5238	0	2	2	1	1	66,67%
compiler	P	1,4968	0	4	4	2	0	30,77%
random	P	2,4409	0	5	3	1	1	41,67%
regexp	P	2,4470	0	4	3	2	0	44,44%
jstl	P	5,0000	0	5	4	1	1	35,71%
lab4	P	3,4000	4	5	4	1	3	50,00%
netkit-ping	P	1,0000	0	1	1	1	1	100,00%
nss_ldap	P	0,9763	0	2	2	1	1	66,67%
nos	P	1,6565	43	5	5	1	1	33,33%
lslayout	P	1,8171	20	7	4	2	0	41,18%
boxer	P	3,1011	6	7	2	1	2	58,33%
netkit-tftpd	P	1,1442	0	2	3	3	0	33,33%
sharutils	P	2,5338	0	5	4	2	0	35,71%
mtunis	P	2,3145	302	5	5	3	0	25,00%
spdb	P	5,5897	0	6	2	1	5	85,71%
xtell	P	2,0052	0	5	5	2	0	35,71%
bunch	P	2,4026	0	7	3	1	2	46,67%
ispell	P	2,3323	493	8	4	2	0	34,78%
netkit-inetd	P	1,3121	0	2	2	2	0	50,00%
nanoxml	P	3,8173	12	9	4	1	2	39,13%
ciald	P	2,8459	134	9	4	2	0	40,91%
jodamoney	P	2,7489	28	9	5	2	0	34,62%
modulizer	P	2,7579	14	7	4	1	2	38,89%
bootp	P	2,1985	0	7	4	2	0	36,84%
jxlsreader	P	3,5921	0	10	4	1	2	40,00%
sysklogd-1	P	1,6870	0	5	12	2	0	22,73%
telnetd	P	1,8475	0	4	7	2	0	21,05%
crond	P	2,3030	0	7	6	3	0	28,00%

Continua na próxima página

Continuação da página anterior

Instância	C.	Maior MQ	It.	Tot. *cl.*	> *cl.*	< *cl.*	*Cl.* iso.	$\frac{k}{n} \cdot 100\%$
netkit-ftp	P	1,7562	46	5	8	3	0	20,83%
rcs	P	2,2262	371	9	6	1	1	32,14%
seemp	P	4,6536	1069	8	5	1	2	38,10%
dhcpd-2	P	3,4889	5	10	5	1	1	38,46%
cyrus-sasl	P	3,2518	9	7	6	1	2	33,33%
tcsh	P	1,1945	0	3	16	3	0	13,04%
micq	P	2,1329	0	8	7	2	0	30,77%
apache_zip	P	5,7663	141	13	4	1	2	43,33%
star	P	3,8321	3431	9	7	2	0	25,00%
bison	P	2,7039	1212	12	5	2	0	33,33%
cia	P	3,7507	2008	13	6	1	1	38,24%
stunnel	P	2,5261	103	7	6	2	0	28,00%
minicom	P	2,5759	1990	10	6	2	0	28,57%
mailx	P	3,2402	1762	13	5	1	1	34,21%
dot	P	2,8470	2023	14	6	1	2	35,00%
screen	P	2,2455	272	11	5	2	0	31,43%
slang	P	4,6794	1188	14	6	1	1	33,33%
slrn	P	2,3928	1795	12	5	1	2	32,43%
net-tools	P	4,3159	3302	15	5	1	1	34,09%
graph10up49	P	1,2590	5152	18	6	1	3	36,73%
wu-ftpd-1	P	2,4371	2651	11	12	2	0	25,00%
joe	P	3,3411	384	15	4	2	0	34,09%
hw	P	8,4967	45	15	1	1	15	100,00%
imapd-1	P	3,6250	759	13	6	1	2	32,50%
wu-ftpd-3	P	3,3405	24	15	10	1	1	31,25%
udt-java	P	5,2829	195	20	6	2	0	37,04%
javaocr	P	9,0242	1662	20	7	1	5	46,51%
dhcpd-1	P	3,9609	145	14	7	2	0	25,45%
icecast	P	2,7482	558	15	7	1	1	29,41%
pfcda_base	P	7,3328	3011	24	3	1	3	44,44%
servletapi	P	9,8715	1187	22	4	1	9	46,81%
php	P	5,3242	374	14	6	1	2	35,90%
bunch2	P	7,7251	1899	18	6	1	4	42,86%
forms	P	8,3258	15	23	6	1	3	35,94%
jscatterplot	P	10,7419	214	25	6	1	1	36,76%
jxlscore	M	9,8050	2775	28	5	1	7	41,79%

Continua na próxima página

Continuação da página anterior

Instância	C.	Maior MQ	It.	Tot. *cl.*	> *cl.*	< *cl.*	*Cl.* iso.	$\frac{k}{n} \cdot 100\%$
elm-2	M	3,8296	1501	24	8	2	0	32,00%
jfluid	M	6,5796	15	24	13	1	1	32,00%
grappa	M	12,7054	74	34	4	1	5	44,74%
elm-1	M	4,3154	2003	26	8	2	0	32,10%
gnupg	M	7,1433	1294	27	5	1	1	36,49%
inn	M	8,0180	366	32	6	1	4	40,00%
bash	M	5,9124	209	30	10	1	1	34,88%
jpassword	M	10,6877	5041	32	4	1	5	36,78%
bitchx	M	4,3860	355	29	6	2	0	31,52%
junit	M	11,0901	11	29	4	1	5	47,54%
xntp	M	8,3524	1471	33	8	1	3	34,74%
acqcigna	M	16,5971	2478	39	3	1	10	52,00%
bunch_2	M	13,6204	2858	39	5	1	11	39,80%
exim	M	6,6261	2115	35	8	1	3	33,33%
xmldom	M	10,9236	29	27	10	1	7	40,30%
cia++	M	15,4724	554	44	3	1	29	69,84%
tinytim	M	12,5415	1358	40	8	1	3	32,79%
mod_ssl	M	10,1118	1462	42	9	1	4	34,15%
jkaryoscope	M	18,9871	38	43	6	1	5	33,86%
ncurses	M	11,8318	1886	44	6	1	1	36,67%
gae_plugin_core	M	17,3370	2191	39	5	1	10	44,83%
lynx	M	4,9814	3027	31	7	1	1	31,00%
javacc	M	10,7216	4754	43	25	1	4	29,66%
lucent	M	59,9488	997	63	2	1	60	95,45%
javageom	M	14,1148	4149	54	7	1	5	33,75%
incl	M	13,6146	837	37	22	1	10	30,33%
jdendogram	M	26,0843	3270	60	5	1	8	40,54%
xmlapi	M	19,0980	1027	52	10	1	12	41,60%
jmetal	G	12,6282	10821	70	5	1	6	39,33%
graph10up193	G	2,2599	8500	67	5	1	3	34,72%
dom4j	G	19,2413	2833	70	5	1	6	38,67%
nmh	G	9,4148	2179	67	8	2	0	35,26%
pdf_renderer	G	22,3573	317	53	41	1	14	30,81%
jung_graph_model	G	32,0325	880	80	5	1	8	42,78%
jung_visualization	G	21,8614	3916	73	7	1	7	38,02%
jconsole	G	26,5195	370	75	5	1	10	40,11%

Continua na próxima página

Continuação da página anterior

Instância	**C.**	**Maior MQ**	**It.**	**Tot. *cl.***	**> *cl.***	**< *cl.***	***Cl.* iso.**	$\frac{k}{n} \cdot 100\%$
pfcda_swing	G	29,0991	284	91	8	1	4	38,40%
jml-1.0b4	G	17,6074	2685	89	11	1	4	33,97%
jpassword2	G	28,6251	3596	91	5	1	8	36,69%
notelab-full	G	29,7388	3982	99	8	1	6	36,26%
poormans CMS	G	34,2540	10241	105	7	1	16	41,50%
log4j	G	32,5186	4319	99	7	1	17	38,37%
jtreeview	G	48,1334	4236	106	6	1	25	39,55%
bunchall	G	16,9948	6013	80	10	1	8	32,92%
jace	G	26,8403	1681	72	51	1	14	26,57%
javaws	G	38,3388	7077	110	9	1	15	37,54%
swing	MG	45,4527	6749	160	6	1	16	42,44%
lwjgl-2.8.4	MG	37,1564	6583	121	21	1	19	30,87%
res_cobol	MG	16,0083	3835	112	32	1	3	24,30%
ping_libc	MG	51,8938	2542	163	8	1	18	41,27%
y_base	MG	58,3474	5110	187	7	1	23	39,04%
krb5	MG	54,6586	2938	187	9	1	10	36,81%
apache_ant_taskdef	MG	66,6626	1805	183	13	1	26	34,01%
itextpdf	MG	58,7145	1703	224	10	1	16	37,15%
apache_lucene_core	MG	76,9618	6078	256	9	1	13	35,96%
eclipse_jgit	MG	86,6262	3142	311	13	1	14	35,67%
linux	MG	54,7162	3591	421	7	1	140	47,90%
apache_ant	MG	102,8400	6856	348	8	1	31	34,83%
ylayout	MG	111,0931	8943	363	16	1	50	36,16%

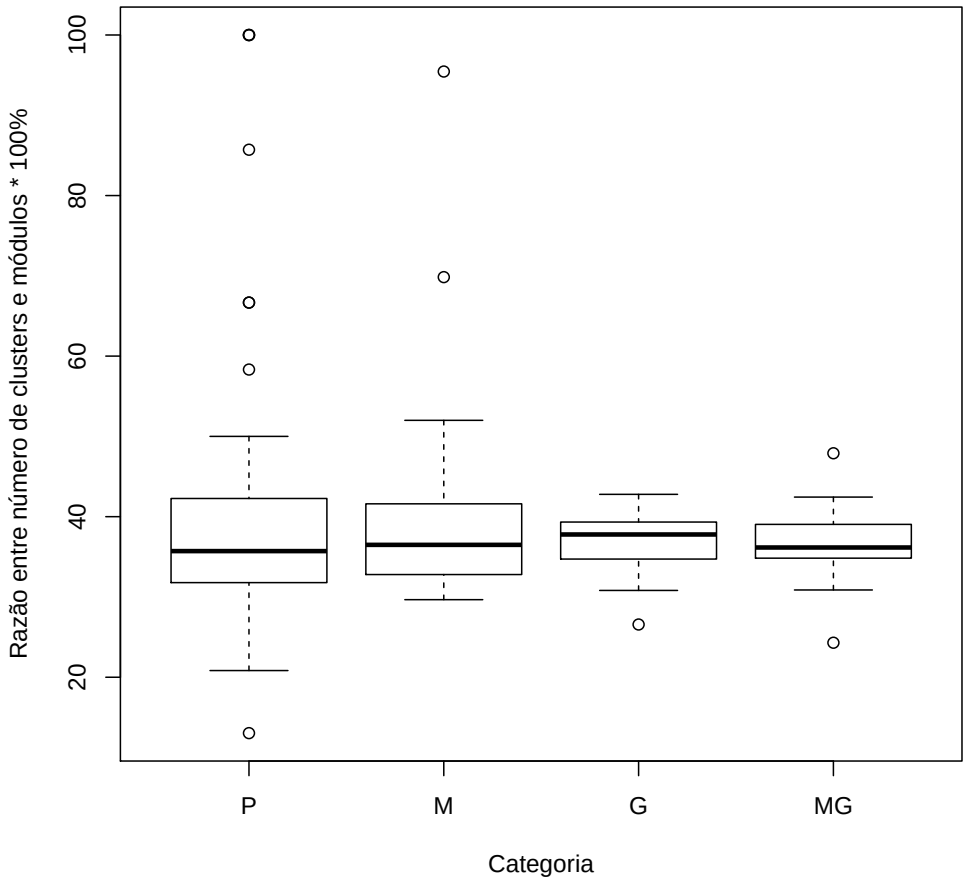


Figura 4.11: *Boxplot* por categoria da razão entre o número de *clusters* e módulos das melhores soluções obtidas.

Praditwong et al. [25] definem duas abordagens multiobjetivo (Seção 2.3.4) chamadas MCA e ECA. A abordagem MCA considera, dentre outros fatores, o número de *clusters* isolados, e a abordagem ECA considera, dentre outros fatores, a diferença entre o maior e o menor *cluster* da solução. A heurística LNS_CMS não considerou nenhum destes fatores. Contudo, é possível obter estes dados das melhores soluções encontradas. A Tabela 4.25 exibe estes dados. A primeira coluna indica o nome da categoria. A segunda coluna exibe o percentual de instâncias cujas soluções possuem ao menos um *cluster* isolado. A terceira coluna exibe o percentual de *clusters* isolados nas soluções encontradas. Por fim, a quarta coluna exibe o somatório da diferença entre o maior e menor *cluster* das soluções encontradas. Observa-se que mais de 73% das instâncias possuem ao menos um *clusters* isolado. Além disso, o percentual de instâncias com *clusters* isolados aumenta quando a categoria contém instâncias maiores, chegando a ser 100% para as instâncias da categoria MG. Outro fator observado é a alta quantidade de *clusters* isolados. O total de *clusters*

isolados é 13,63%.

Tabela 4.25: Resumo das características das melhores soluções obtidas por categoria.

Categ.	Percentual de instâncias com *clusters* isolados	Percentual de *clusters* isolados	Somatório da diferença entre maior e menor *cluster*
P	54,69%	14,78%	223
M	89,66%	19,59%	187
G	94,44%	11,35%	139
MG	100,00%	12,48%	146
TOTAL	**73,39%**	**13,63%**	**695**

4.8 Ameaças à validade do estudo

De acordo com Wohlin et al. [32] existem quatro categorias de ameaças à validade do estudo: Construção, Interna, Externa e Conclusão.

A validade de Construção trata da relação entre teoria e observação [22]. Para superar esta ameaça a teoria foi tratada no Capítulo 1 e a parte de observação foi feita por meio de experimentos abordados no Capítulo 4.

A validade Interna de um estudo é definida como a capacidade de um novo estudo replicar o comportamento do estudo atual [2]. Para garantir a reprodutibilidade do estudo atual, os experimentos utilizaram instâncias disponíveis em outros trabalhos [3, 12, 14, 20, 21, 22, 24, 25]. A função objetivo utilizada (MQ) já havia sido empregada em outros trabalhos, com isso os resultado obtidos puderam ser comparados com eles. Como um facilitador, os códigos-fonte e instâncias utilizados estão disponíveis no endereço https://bitbucket.org/marlonmoncores/unirio_lns_cms.

A validade Externa é definida como a capacidade de generalização do estudo e se o estudo pode ser utilizado em um escopo diferente do definido [22]. O escopo do atual estudo pode ser alterado, escolhendo-se outras instâncias para serem trabalhadas. Para isto, pode-se criar novas instâncias a partir de softwares reais ou com características específicas. A instância é um arquivo texto simples, descrito na Seção 4.1.

A validade de Conclusão mede a relação entre os tratamentos e os resultados, determinando a capacidade do estudo em gerar alguma conclusão [2]. Para evitar as ameaças

à conclusão os experimentos foram repetidos diversas vezes para cada instância (2.000 valores aleatórios para o primeiro experimento, e 100 repetições para os demais experimentos), visto que as heurísticas possuem componentes estocásticos. Somado a isto, a heurística ILS criada por Pinto [22] também foi executada 100 vezes. Os resultados obtidos por ambos métodos foram comparados utilizando testes estatísticos. O nível de significância de distinção entre as amostras foi calculado pelo teste não paramétrico Wilcoxon-Mann-Whitney [7]. Para verificar a superioridade de uma amostra em relação à outra, foi calculado o tamanho de efeito utilizando a métrica não paramétrica $\hat{A}_{12}$ [31].

4.9 Considerações finais

Este capítulo apresentou os resultados dos experimentos computacionais realizados para avaliar a heurística LNS_CMS para o problema CMS. Também foram expostas as avaliações feitas em cada experimento. O próximo capítulo contém as contribuições, limitações e ideias de melhorias futuras.

5. Conclusão

A meta-heurística Busca em Vizinhança Grande caracteriza-se por usar uma vizinhança que cresce exponencialmente em função do tamanho do problema, ou simplesmente por usar uma vizinhança muito grande. No processo de busca, a vizinhança é percorrida de forma heurística, ou seja, apenas alguns vizinhos são observados. A cada iteração parte da solução é destruída e, então, reconstruída. Ao se criar uma heurística baseada em Busca em Vizinhança Grande os algoritmos responsáveis pela geração de solução inicial, destruição e reconstrução da solução têm que ser desenvolvidos. O Capítulo 3 descreve cada um dos algoritmos criados.

Este capítulo está dividido em duas seções. Na Seção 5.1 são apresentadas as principais contribuições. Na Seção 5.2 são apresentas as limitações do trabalho desenvolvido e algumas perspectivas futuras do trabalho.

5.1 Contribuições

O Capítulo 4 apresentou experimentos computacionais, seus resultados e análise dos dados. Os resultados obtidos indicam que a heurística proposta para resolver o problema CMS, baseada na meta-heurística Busca em Vizinhança Grande, chamada de LNS_CMS, é uma heurística eficiente e eficaz para o problema CMS, quando comparada a outras heurísticas e a algoritmos exatos aplicados ao mesmo problema.

O primeiro estudo experimental teve como objetivo avaliar diferentes escolhas para os componentes livres da meta-heurística Busca em Vizinhança Grande. Foram avaliados os métodos de construção de solução inicial, reparação da solução, destruição da solução e o parâmetro grau de destruição. Com base nos experimentos conclui-se que o método de geração de solução inicial e de reparação da solução funcionam melhor quando não possuem componentes aleatórios. Contudo, para o método de destruição, o melhor resultado

foi obtido com o método totalmente aleatório. Isto ocorre pela natureza de cada um dos processos. Os métodos de construção e reparação buscam a intensificação da solução, enquanto que o método destruidor busca a geração de diversidade entre as soluções geradas. Outro fator estudado foi o grau de destruição, que obteve o pior desempenho no experimento realizado quando apenas 5% foi destruído. Isto revela que destruir 5% da solução não é suficiente e a busca fica presa, sem conseguir alcançar boas soluções. Por outro lado, os percentuais de destruição que obtiveram melhores resultados foram 15%, 10% e 20%, nesta ordem. Então, conclui-se que um bom intervalo para o grau de destruição são valores entre 10% e 20%.

O segundo estudo experimental teve como objetivo aprofundar a análise feita no estudo anterior, além de incluir a investigação do componente critério de parada. Este segundo experimento analisou configurações específicas de todos os componentes livres do método. A configuração escolhida como vencedora foi aquela que usa dois algoritmos de reparação alternadamente enquanto houver melhoria na qualidade da solução. Esta configuração alcançou a melhor média na geração de MQ para as 18 instâncias utilizas no experimento. Contudo, a configuração escolhida possui um elevado tempo de processamento.

Um estudo experimental comparativo foi feito entre a heurística LNS_CMS e a heurística ILS_CMS, proposta por Pinto [22], baseada na meta-heurística *Iterated Local Search*. Para cada heurística, a busca foi executada 100 vezes com cada uma das 124 instâncias trabalhadas. Observando a eficácia, percebe-se que a LNS_CMS alcança melhores médias de resultados que a ILS_CMS em todas as quatro categorias de instância. Na categoria Pequena a heurística LNS_CMS obtém 31 vitórias contra 28 da heurística ILS_CMS. Para a categoria Média o resultado também é favorável a LNS_CMS, sendo, 24 vitórias e apenas 4 derrotas. Nas categorias Grande e Muito Grande a heurística LNS_CMS supera a ILS_CMS em 30 instâncias, e em uma instância o *p-value* é maior que 0,05 e então é considerado o empate. Ao se comparar a eficiência das heurísticas, os resultados mostram que a LNS_CMS possui média de tempo de processamento menor que a ILS_CMS para as categorias Pequena e Média. Contudo, para as categorias Grande e Muito Grande o tempo de processamento da LNS_CMS é maior que o da ILS_CMS. No total, o tempo de processamento aproximado com a LNS_CMS foi de 129 horas enquanto que a ILS_CMS levou aproximadamente 8 horas.

Um segundo estudo experimental comparou a heurística LNS_CMS e os melhores resultados encontrados na literatura. Também foram incluídos os resultados obtidos pela heurística ILS_CMS. Quanto à eficácia, os resultados mostram que a LNS_CMS consegue superar os resultados da literatura para as categorias de instâncias Média, Grande e Muito

Grande. Considerando o *gap* (diferença percentual entre a qualidade das soluções obtidas) obtido em cada instância, os resultados indicam que a busca LNS_CMS supera os demais métodos da literatura em 58 instâncias, ocorrem 39 empates e a LNS_CMS é derrotada em outras 27 instâncias. Observando-se a eficiência, a LNS_CMS consegue ter a média de tempo de processamento menor que a média da literatura. Isto ocorre porque alguns dos métodos encontrados na literatura são exatos (conseguem resolver para o resultado ótimo) e possuem custo computacional muito elevado.

Além destas contribuições, o presente trabalho reuniu um total de 124 instâncias. Em nenhum outro trabalho encontrado na literatura foi utilizada uma quantidade tão grande de instâncias. A maior quantidade de instâncias utilizadas permitiu definir 4 grupos de instâncias. Cada grupo foi definido utilizando-se o método de clusterização *k-means*. Os experimentos também permitem que o trabalho seja utilizado como referência para outros que tratem do problema CMS. O método ILS_CMS foi adaptado para aceitar instâncias com peso e com autorrelacionamento entre os módulos. Documentou-se, na presente dissertação, o resultado da execução do ILS_CMS adaptado para as 124 instâncias da literatura, de forma a permitir que trabalhos futuros utilizem também estes dados.

Observando-se as características das melhores soluções geradas (Seção 4.7) é possível perceber que estas possuem uma quantidade de *clusters* em torno de 37% da quantidade de módulos. Enquanto que, em clusterizações feitas manualmente este mesmo fator é menor que 6% [15]. Assim, soluções geradas com uma quantidade grande de *clusters* podem não suprir as necessidades da engenharia de software. Desta forma, pode-se concluir que a utilização da função objetivo MQ para o problema CMS é questionável.

5.2 Limitações e perspectivas futuras

Nesta seção destaca-se as principais limitações observadas analisando-se os resultados dos experimento e apresenta-se as perspectivas de trabalhos futuros.

O estudo limitou-se a testar valores para o grau de destruição entre 5% e 50%, com intervalo de 5%. Contudo, após realização dos experimentos, concluiu-se que o intervalo de valores mais promissor está entre 10% e 20%. Então, em um experimento futuro, podem ser testados mais valores neste intervalo.

Em diversas execuções o resultado de saída da busca foi a mesma solução que a heurística recebeu como entrada. Assim, um estudo mais detalhado das condições de parada pode ser feito com o intuito de detectar situações onde a busca fica estagnada. Pois, com

esta detecção, o processamento poderá ser interrompido antes, tornando a busca mais eficiente (diminuindo o tempo de processamento).

A heurística desenvolvida limitou-se em aceitar, a cada iteração, apenas soluções melhores. Contudo, Pisinger et al. [23] sugere a utilização de um critério de aceitação baseado em um processo de busca que use critérios de aceitação similares aos usados pela metaheurística *Simmulated Annealing*. Neste processo de busca são aceitas soluções de pior qualidade com uma determinada probabilidade. A cada iteração a probabilidade de aceitar soluções piores diminui. O objetivo é permitir que a busca possua mais liberdade para explorar o espaço de busca nas primeiras iterações.

Outra perspectiva futura é a utilização de uma Busca em Vizinhança Grande Adaptativa [26]. Neste cenário diversos métodos de destruição e de construção são utilizados durante a busca. A cada iteração ocorre um sorteio para a seleção dos métodos que serão utilizados. Também é possível avaliar o desempenho dos métodos e aumentar a chance dos que alcançam bons resultados serem sorteados. Ropke et al. [26] ainda sugerem a utilização de um ruído aleatório na função objetivo, com o intuito de aumentar a diversificação do método.

O atual trabalho também se limitou em utilizar apenas a função objetivo MQ. Uma possibilidade de continuação do trabalho é aplicar a função objetivo EVM e comparar com outros trabalhos encontrados na literatura que também tenham utilizado a função objetivo EVM.

Para finalizar, por ser uma heurística simples de ser implementada e adaptada a novos cenários, a LNS também pode ser aplicada em outros problemas. Em especial, problemas da área de Engenharia de Software Baseada em Buscas.

Referências Bibliográficas

[1] AHUJA, R. K., ERGUN, O., ORLIN, J. B., et al. "A survey of very large-scale neighborhood search techniques", *Discrete Appl. Math.* v. 123, n. 1-3, pp. 75–102, Nov. 2002.

[2] BARROS, M. O. *Gerenciamento de projetos baseado em cenários: uma abordagem de modelagem dinâmica e simulação*. Tese de D.Sc., COOPE/UFRJ, 2001.

[3] BARROS, M. O. "An analysis of the effects of composite objectives in multiobjective software module clustering". In: *GECCO '12*, pp. 1205–1212, Philadelphia, Jul. 2012.

[4] BARROS, M. O. "An experimental study on incremental search-based software engineering". In: *Search Based Software Engineering - 5th International Symposium, SSBSE 2013, St. Petersburg, Russia, August 24-26, 2013. Proceedings*, pp. 34–49, St.Petersburg, Ago. 2013.

[5] CHEN, D. Z., DAESCU, O., DAI, Y., et al. "Efficient algorithms and implementations for optimizing the sum of linear fractional functions, with applications", *Journal of Combinatorial Optimization* v. 9, n. 1, pp. 69–90, Fev. 2005.

[6] DOVAL, D., MANCORIDIS, S., MITCHELL, B. S. "Automatic clustering of software systems using a genetic algorithm". In: *Software Technology and Engineering Practice, 1999. STEP '99. Proceedings*, pp. 73–81, Pittsburgh, Ago. 1999.

[7] FELTOVICH, N. "Nonparametric tests of differences in medians: Comparison of the wilcoxon–mann–whitney and robust rank-order tests", *Experimental Economics* v. 6, n. 3, pp. 273–297, Nov. 2003.

[8] GLOVER, F., LAGUNA, M., MARTÍ, R. "Fundamentals of scatter search and path relinking", *Control and Cybernetics* v. 39, n. 3, pp. 653–684, Jul. 2000.

[9] HANSEN, P., JAUMARD, B. "Cluster analysis and mathematical programming", *Math. Program.* v. 79, n. 1-3, pp. 191–215, Out. 1997.

[10] HARMAN, M., SWIFT, S., MAHDAVI, K. "An empirical study of the robustness of two module clustering fitness functions". In: *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, pp. 1029–1036, Washington, Jun. 2005.

[11] IEEE. "Ieee standard glossary of software engineering terminology", *IEEE Std 610.12-1990*, pp. 1–84, Dez. 1990.

[12] KÖHLER, V., FAMPA, M., ARAÚJO, O. "Mixed-integer linear programming formulations for the software clustering problem", *Computational Optimization and Applications* v. 55, n. 1, pp. 113–135, Mai. 2013.

[13] KRAMER, H. H., UCHOA, E., FAMPA, M., et al. "Column generation approaches for the software clustering problem". In: *XLVI Simpósio Brasileiro de Pesquisa Operacional*, pp. 2639–2650, Salvador, Set. 2014.

[14] KUMARI, A. C., SRINIVAS, K. "Article: Software module clustering using a fast multi-objective hyper-heuristic evolutionary algorithm", *International Journal of Applied Information Systems* v. 5, n. 6, pp. 12–18, April. 2013.

[15] LANZA, M., MARINESCU, R., *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. 1 ed. Berlin, Springer-Verlag, 2010.

[16] LI, H.-L. "A global approach for general 0–1 fractional programming", *European Journal of Operational Research* v. 73, n. 3, pp. 590–596, Mar. 1994.

[17] LOURENÇO, H. R., MARTIN, O. C., STÜTZLE, T. "Iterated local search: Framework and applications". In: Gendreau, M., Potvin, J.-Y., (eds), *Handbook of Metaheuristics*, 2 ed., chapter 12, pp. 363–397, New York, USA, Springer US, 2010.

[18] MAHDAVI, K., HARMAN, M., HIERONS, R. M. "A multiple hill climbing approach to software module clustering". In: *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pp. 315–324, Amsterdam, Sept. 2003.

[19] MANCORIDIS, S., MITCHELL, B. S., CHEN, Y., et al. "Bunch: a clustering tool for the recovery and maintenance of software system structures". In: *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pp. 50–59, Oxford, Ago. 1999.

[20] MANCORIDIS, S., MITCHELL, B. S., RORRES, C., et al. "Using automatic clustering to produce high-level system organizations of source code". In: *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pp. 45–52, Ischia, Jun. 1998.

[21] MITCHELL, B. S. *A Heuristic Search Approach to Solving the Software Clustering Problem*. Ph.D. thesis, Drexel University, Philadelphia, PA, USA, 2002.

[22] PINTO, A. F. *Uma heurística baseada em busca local iterada para o problema de clusterização de módulos de software*. Dissertação de M.Sc., PPGI/UNIRIO, Rio de Janeiro, RJ, Brasil, 2014.

[23] PISINGER, D., ROPKE, S. "Large neighborhood search". In: Gendreau, M., Potvin, J.-Y., (eds), *Handbook of Metaheuristics*, 2 ed., chapter 12, pp. 399–419, New York, USA, Springer US, 2010.

[24] PRADITWONG, K. "Solving software module clustering problem by evolutionary algorithms". In: *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*, pp. 154–159, Nakhon Pathom, Mai. 2011.

[25] PRADITWONG, K., HARMAN, M., YAO, X. "Software module clustering as a multi-objective search problem", *IEEE Transactions on Software Engineering* v. 37, n. 2, pp. 264–282, Mar. 2011.

[26] ROPKE, S., PISINGER, D. "An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows", *Transportation Science* v. 40, n. 4, pp. 455–472, Nov. 2006.

[27] SEMAAN, G. S., OCHI, L. S. "Algoritmo evolutivo para o problema de clusterização em grafos orientados".

[28] SHAW, P. "Using constraint programming and local search methods to solve vehicle routing problems". In: *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, pp. 417–431, Pisa, Out. 1998.

[29] ŠÍMA, J., SCHAEFFER, S. E. "On the np-completeness of some graph cluster measures". In: *Proceedings of the 32Nd Conference on Current Trends in Theory and Practice of Computer Science*, pp. 530–537, Měřín, Jan. 2006.

[30] TUCKER, A., SWIFT, S., LIU, X. "Variable grouping in multivariate time series via correlation", *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on* v. 31, n. 2, pp. 235–245, Abr. 2001.

[31] VARGHA, A., DELANEY, H. D. "A critique and improvement of the cl common language effect size statistics of mcgraw and wong", *Journal of Educational and Behavioral Statistics* v. 25, n. 2, pp. 101–132, Jun. 2000.

[32] WOHLIN, C., RUNESON, P., HÖST, M., et al., *Experimentation in Software Engineering: An Introduction*. 1 ed. Boston, Kluwer Academic Publishers, 2000.